Modern Operators, Loops, and Timers

> ## Goals
>
> By the end of this lesson, you should understand:
>
> - What the **spread/rest** operator is and it's different contexts
> - **for...of** and **for...in** loops and when to use each of them
> - The **setInterval** and **setTimeout** timer methods
>
> ## Introduction
>
> Today, we will be delving into some more building blocks of modern JavaScript: we'll be looking at the **spread/rest operator**, **for...of** loops, **for...in** loops, **setInterval**, and **setTimeout**. The **spread operator** allows you to spread an array or object literal into individual elements, making it easy to concatenate and manipulate data. **For...of** loops are ideal for iterating over an array or any other iterable object, allowing you to execute code on each item. **For...in** loops, on the other hand, allow you to loop through the properties of an object, which is often very helpful for dynamically accessing and modifying those properties. Finally, **setInterval** and **setTimeout** are essential tools for scheduling or delaying code execution. Throughout this lesson, we will explore each of these concepts in detail, providing you with the knowledge and skills needed to work with these important JavaScript language features.

Spread/Rest Operator

> The spread/rest operator is a powerful feature of JavaScript that allows you to manipulate arrays and objects in a flexible and efficient way. The operator is denoted by three dots (...) and can be used in two different contexts: as a spread operator or as a rest parameter.

One common use case for the **spread operator** is when combining objects and concatenating arrays. For example:

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combinedArr = [...arr1, ...arr2];
console.log(combinedArr); // Output: [1, 2, 3, 4, 5, 6]
```

You will see this code already in the IDE. For the rest of the examples, try writing them into the IDE and check your output.

```
const obj1 = {a: 1, b: 2};
const obj2 = {c: 3, d: 4};
const newObj = {...obj1, ...obj2};
console.log(newObj); // Output: {a: 1, b: 2, c: 3, d: 4}
```

In the above two examples, the spread operator is used first to “spread” the contents of **arr1** and **arr2** into **combinedArr**, resulting in a new array with all six elements. It is then used to create a new object containing all the properties of both **obj1** and **obj2**.
>
The spread operator can also be used simply to create a copy of an array or object, while adding additional elements or properties:

```
const originalArr = [1, 2, 3];
const arrCopy = [...originalArr, 4, 5, 6, 7];
console.log(arrCopy); // Output: [1, 2, 3, 4, 5, 6, 7]

const originalObj = {a:1,b:2,c:3,d:4};
const objCopy = {...originalObj, e:5, f:6, g:7};

console.log(objCopy); // Output: {a:1, b:2, c:3, d:4, e:5, f:6,
        g:7}
```

In this example, the spread operator is used to create copies of **originalArr** and **objCopy**, both retaining the same contents plus adding in some new ones.
>
The **rest operator**, on the other hand, while operating in a similar way, is instead used in function parameters to capture a variable number of arguments into an array:

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num);
}

console.log(sum(1, 2, 3)); // 6
console.log(sum(4, 5, 6, 7)); // 22
```

In this example, the rest operator takes any and all arguments passed into the **sum** function and stores them in an array called **numbers**. The **reduce** method is then used to sum up all the numbers in the array and return the total.
>
Practice Exercise:
Write a function that takes two arrays as arguments and returns a new array that contains all the elements from both arrays using the spread operator.

For...of Loops

> The **for...of** loop was one of the many new features introduced in ES6 and allows you to iterate over the elements of an array, string, or any other iterable object. The syntax for the **for...of** loop is as follows:

```
for (let element of iterable) {
  // code to execute for each element
}
```

Here, **element** is the current element being iterated over and **iterable** is the array or iterable object being iterated. Let's look at a couple examples to see this in action.
>
Here is an example of using **for...of** loop with an array:

```
const colors = ['red', 'green', 'blue'];


for (const color of colors) {
  console.log(color);
} // Output: red green blue
```

You can also use **for...of** loops with nested arrays:

```
const matrix = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ];


for (const row of matrix) {
  for (const cell of row) {
    console.log(cell);
  }
} // Output: 1 2 3 4 5 6 7 8 9
```

(Note that the output in the preview window has been vertical, rather than horizontal.)
>
In addition to simple arrays and nested arrays, you can even use **for...of** loop with other iterable objects such as strings, maps, and sets (we'll learn about maps and sets next week). Let's see an example using a string:

```javascript
const sentence = 'The quick brown fox jumps over the lazy dog.';



for (const character of sentence) {
  console.log(character);
} // Output: T h e q u i c k b r o w n f o x j u m p s o v e r t
      h e l a z y d o g .
```

One advantage of using **for...of** loops is that they are more concise and readable than traditional **for** loops.
>
Practice Exercise:
Write a function that takes an array of numbers as an argument and returns the sum of all the numbers using a **for...of** loop.

For...in Loops

> The **for...in** loop is another type of loop that allows you to iterate over the properties of an object. Unlike the **for...of** loop, which iterates over the values of an array, the **for...in** loop iterates over the keys or property names of an object.
>
> The syntax for the **for...in** loop is as follows:

```
for (let key in obj) {
  // code to execute for each property
}
```

Here, **key** is the name of the current property being iterated over and obj is the object being iterated.
>
Let's see a couple examples:

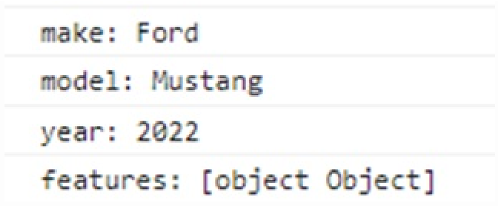
```
// using for...in loop with an object literal:
const person = {
  firstName: 'John',
  lastName: 'Doe',
  age: 30
};


for (const property in person) {
  console.log(property + ": " + person[property]);
}


/* Output:

  firstName: John
  lastName: Doe
  age: 30

*/
```

In this example, the loop iterates over the properties of the person object and logs each property name and its value.
>

In addition to simple objects, you can use **for...in** loop with more complex objects such as arrays or objects with nested properties:

```
const car = {
  make: 'Ford',
  model: 'Mustang',
  year: 2022,
  features: {
    engine: 'V8',
    transmission: 'manual',
    color: 'red'
  }
};

for (const property in car) {
  if (typeof car[property] === 'object') {
    console.log(property + ": ", car[property]);
  } else {
    console.log(property + ": " + car[property]);
  }
}


/* Output:

  make: Ford,
  model: Mustang,
  year: 2022,
  features: {
    engine: 'V8',
    transmission: 'manual',
    color: 'red'
  }

*/
```

Here we created a **car** object with various properties, including a features property which it, itself, and object. Now, when creating our **for...in** loop, we're adding an if check inside the loop to see whether the given property's value is an object; if it is, we modify slightly how we log it to the console by adding a comma after the ":". Without doing something like this, what would get logged to the console would look similar to the following:



```
make: Ford
model: Mustang
year: 2022
features: [object Object]
```

It’s also important to note that the order in which the properties are iterated over is not guaranteed, so if you need to iterate over the properties in a very specific order, you should use an array of property names instead.
>
Practice Exercise:
Write a function that takes an object as an argument and returns an array of all the property names using a **for...in** loop.

setInterval Timer Method

> The `setInterval` function is a built-in JavaScript function that allows you to execute a specified function at specified intervals. The function typically takes two arguments: the first argument is the function to execute, and the second argument is the time interval in milliseconds. Here's what the syntax looks like:
>
> `setInterval(function, milliseconds)`
>
> The **function** parameter is the function that needs to be executed after each interval. The **milliseconds** parameter is the time interval in milliseconds after which the function needs to be called.
>
> Here's an example:

```
function logMessage() {
  console.log("Hello, world!");
}


setInterval(logMessage, 1000);
```

This code will log the message "Hello, world!" to the console every second.
>
Note that the `setInterval` function returns a unique ID that can be used to stop the interval using the `clearInterval` function.

```
const intervalID = setInterval(logMessage, 1000);
clearInterval(intervalID);
```

Oftentimes, you would call the `clearInterval` method inside the `setTimeout` method, which we will look at in the next section.
>

```
function logMessage() {
  console.log("Hello, world!");
}

const intervalID = setInterval(logMessage, 1000);
console.log("Interval started with ID: " + intervalID);

setTimeout(function() {
  clearInterval(intervalID);
  console.log("Interval stopped with ID: " + intervalID);
}, 5000);
```

Practice Exercise:

Write a function that logs a message to the console every 5 seconds using setInterval.

setTimeout Timer Method

> The `setTimeout` function is another built-in JavaScript function that allows you to execute a specified function after a specified delay. The function takes two arguments: the first argument is the function to execute, and the second argument is the delay in milliseconds.

For example:

```
function logMessage() {
  console.log("Hello, world!");
}


setTimeout(logMessage, 1000);
```

This code will log the message “Hello, world!” to the console after a delay of one second.
>
Note that like the `setInterval` function, the `setTimeout` function also returns a unique ID that can be used to cancel the timeout using the `clearTimeout` function.
>

> ### Practice Exercise:
>
> Write a function that logs a message to the console after a random delay between 1 and 5 seconds using the `setTimeout` function.

Conclusion and Takeaway

> Throughout this lesson, we went over the spread/rest operator, for…of loops, for…in loops, setInterval, and setTimeout, which are all very useful tools for any JavaScript developer. The **spread/rest operator** allows for more concise and readable code when working with arrays and objects, while the **for…of** and **for…in** loops provide flexible and efficient ways to iterate over array and object data. The **setInterval** and **setTimeout** functions provide valuable control over the timing and execution of certain tasks. By mastering these features of JavaScript, you can greatly enhance the quality and functionality of your code.

Destructuring

> ## Goals
>
> By the end of this lesson, you should understand:
>
> - Object destructuring and how to use it
> - Array destructuring and how to use it
>
> ## Introduction
>
> In JavaScript, destructuring is a powerful feature that allows you to extract values from arrays or objects and assign them to variables using a concise syntax. This can make your code more readable and easier to work with. There are two types of destructuring in JavaScript: object destructuring and array destructuring.

Object Destructuring

> Object destructuring allows you to extract properties from an object and assign them to variables with the same name. This can be useful when you need to access multiple properties of an object and don't want to repeat the object name every time.

Syntax

Let's have a look at the syntax for object destructuring:
>
`const {property1, property2, ...} = object;`
>
The above may look a little strange at first glance, but what is happening here is that we are extracting some (or all) of a given object's properties (or methods), which then effectively become variables of the same name which we can use throughout our code.

Example

Here's an example of object destructuring:

```
const person = {
  firstName: 'John',
  lastName: 'Doe',
  age: 30
};


const {firstName, lastName, age} = person;


console.log(firstName); // Output: 'John'
console.log(lastName); // Output: 'Doe'
console.log(age); // Output: 30
```

In this example, we are once again creating an object called "person". Inside our person object, we are defining three properties: `firstName`, `lastName`, and `age`. We then use object destructuring so that we can use these properties elsewhere in our code without having to utilize dot or bracket notation (e.g., person.firstName or person[age]). We can then log these variables to the console to verify that they contain the correct values.
>

Default Values
You can also specify default values in object destructuring in case the property doesn’t exist in the object.
>
Here’s an example:

```
const person = {
  firstName: 'John',
  lastName: 'Doe'
};


const {firstName, lastName, age = 30} = person;


console.log(firstName); // Output: 'John'
console.log(lastName); // Output: 'Doe'
console.log(age); // Output: 30
```

Here, we add a default value of 30 for the “age” variable in case it doesn’t exist in the object. To make sure this works, try removing the age property from our **person** object, then console log the age property. We will still get **30** because we defined it as a default.
>
Renaming Variables
You can rename the variables generated from destructured properties by using the following syntax:
`const {property1: newVariableName1, property2: newVariableName2, ...} = object;`
>
Let’s continue using our **person** object as an example:

```
const person = {
  firstName: 'John',
  lastName: 'Doe'
};


const {firstName: fName, lastName: lName} = person;


console.log(fName); // Output: 'John'
console.log(lName); // Output: 'Doe'
```

Here we have renamed the **firstName** and **lastName** variables to **fName** and **lName**, respectively.
>
Nested Objects
You can destructure nested objects using the same syntax. You just need to use another set of curly braces around the nested properties:

```
const person = {
  name: {
    firstName: 'John',
    lastName: 'Doe'
  },
  age: 30
};



const {name: {firstName, lastName}, age} = person;



console.log(firstName); // Output: 'John'
console.log(lastName); // Output: 'Doe'
console.log(age); // Output: 30
```

Revamping the structure of our **person** object a little, to now include a nested object called **name** containing the properties **firstName** and **lastName**, we can then use the same object destructuring syntax to extract these properties from the **name** object, as well as the **age** property.

Array Destructuring

> Array destructuring works in pretty much the same way as object destructuring. We are still extracting values which become variables, it's just that we're taking them from an array now and the variable names don't really come from the elements. This can be useful when you need to access multiple elements of an array and don't want to repeat the array name every time.

Syntax

The syntax is virtually identical, except we use brackets instead of curly braces:
>
`const [element1, element2, ...] = array;`
>
We use square brackets to indicate the elements we want to extract from the array. These elements are then assigned to variables with the same name, which we can use throughout our code.
>
Example

Let's look at an example:

```
const numbers = [1, 2, 3];
const [firstNumber, secondNumber, thirdNumber] = numbers;



console.log(firstNumber); // Output: 1
console.log(secondNumber); // Output: 2
console.log(thirdNumber); // Output: 3
```

Here we are creating an array called "numbers" with three elements. We then use array destructuring to extract these elements and assign them to variables with the same names. We can then use these variables to access the corresponding values from the array.

It's important to note that the order of the variables matters. When you destructure an array, each variable you assign represents an element in the array, in the order that they appear. Naming variables to reflect their position in the array, will make it easier to understand what each variable represents.

As before, we then log these variables to the console to verify that they contain the correct values.
>
Default Values
Like with object destructuring, you can specify default values for the variables in array destructuring as well:

```
const numbers = [1, 2];
const [firstNumber, secondNumber, thirdNumber = 3] = numbers;


console.log(firstNumber); // Output: 1
console.log(secondNumber); // Output: 2
console.log(thirdNumber); // Output: 3
```

In this example, we add a default value of 3 for the “thirdNumber” variable in case the array doesn’t have a third element.
>
Skipping Elements
You can skip elements in an array by using commas in the destructuring syntax:

```
const numbers = [1, 2, 3];
const [firstNumber, , thirdNumber] = numbers;


console.log(firstNumber); // Output: 1
console.log(thirdNumber); // Output: 3
```

In this example, we skip the second element of the **numbers** array by leaving a blank space in between the two commas.
>
Swapping Values
Additionally, array destructuring can be used to swap the values of two variables:

```
let a = 1;
let b = 2;


[a, b] = [b, a];


console.log(a); // Output: 2
console.log(b); // Output: 1
```

In this example, we use array destructuring to swap the values of “a” and “b”.
>
Nested Arrays
You can also destructure nested arrays using the same syntax:

```
const numbers = [1, [2, 3], 4];
const [firstNumber, [secondNumber, thirdNumber], fourthNumber] =
numbers;



console.log(firstNumber); // Output: 1
console.log(secondNumber); // Output: 2
console.log(thirdNumber); // Output: 3
console.log(fourthNumber); // Output: 4
```

In this example, we create an array called “numbers” with a nested array. We then use array destructuring to extract the elements from both the outer and inner arrays.

Conclusion and Takeaway

> In this lesson, we went over the ins and outs of object and array destructuring. Destructuring can make your code more concise and readable by allowing you to easily extract values from objects and arrays.

By using default values, skipping elements, and swapping values, you can further customize the destructuring process to suit your needs. As with any new concept you learn, it’s a good idea to keep practicing using object and array destructuring in your own code to become more and more comfortable with this very useful JavaScript feature.

Practice Time

> Work alone or with a partner to answer the following practice questions. Try to come up with solutions first, but If you get stuck you can always refer back to the material in this lesson or by asking an instructor or TA for help. Happy coding!
>
> 1. Create an object called “person” with the following properties: firstName, lastName, age, and address. Use object destructuring to extract the firstName and address properties and assign them to variables with the same name. Log these variables to the console.
>
> 2. Create an array called “colors” with the following elements: red, green, blue. Use array destructuring to extract the first and last elements and assign them to variables with the same name. Log these variables to the console.
>
> 3. Create an object called “person2” with the following properties: firstName, lastName, and age. Use object destructuring to extract the firstName and lastName properties and assign them to variables with different names. Log these variables to the console.
>
> 4. Create an array called “numbers” with the following elements: 1, 2, 3, 4, 5. Use array destructuring to extract the first three elements and assign them to variables with the names `one`, `two`, and `three`. Log these variables to the console.
>
> 5. Create an array called “students” with the following elements: John, Jane, and Jim. Use array destructuring to swap the values of the first and last elements. Log the “students” array to the console to verify that the values have been swapped.
>
> When completed, share the code you developed for all problems.

Math Methods

> ## Goals
>
> By the end of this lesson, you should understand:
>
> - Various math methods in JavaScript
> - How to use these methods individually and how to combine them
>
> ## Introduction
>
> In JavaScript, the Math object provides a collection of methods that allow developers to perform several different mathematical operations easily. Math is a built-in object in JavaScript that provides a range of methods to work with numbers. Math methods can be used for performing basic arithmetic operations, rounding, generating random numbers, and more. Three of the most commonly used Math methods in JavaScript are **Math.floor()**, **Math.ceil()**, and **Math.random()**. This lesson will cover these three methods in detail, along with examples and (of course) some practice exercises.

Math.floor()

> The **Math.floor()** method returns the largest integer less than or equal to a given number. The returned value is rounded **down** (hence, floor) to the nearest integer.
>
> Here’s an example:

```
const num = 5.9;
const roundedDown = Math.floor(num);


console.log(roundedDown); // Output: 5
```

In this example, the variable **num** is assigned the value of 5.9. When we call the Math.floor() method with **num**, it returns the largest integer that is less than or equal to 5.9, which is 5. The value of **roundedDown** is then assigned the value of 5, which is logged to the console.

The Math.floor() method can be useful when you need to round a number down to the nearest integer. For example, if you have a shopping cart with items that cost a certain amount, you might want to round the total price down to the nearest dollar.
>

Math.ceil()

> The **Math.ceil()** (short for ceiling) method is similar to **Math.floor()** except that it returns the smallest integer greater than or equal to a given number, and the returned value is rounded up to the nearest integer.

Here’s an example:
>

```
const num = 5.1;
const roundedUp = Math.ceil(num);


console.log(roundedUp); // Output: 6
```

In this example, the variable **num** is assigned the value of 5.1. When we call the Math.ceil() method with **num**, it returns the smallest integer that is greater than or equal to 5.1, which is 6. The value of **roundedUp** is then assigned the value of 6, which is logged to the console.
>
The Math.ceil() method can be useful when you need to round a number up to the nearest integer. For example, if you have a recipe that calls for a certain number of cups of flour, but you only have a half-full bag of flour, you might need to round the number up to make sure you have enough.
>

Math.random()

> The Math.random() method returns a random number between 0 and 1. This means that the returned value can be any decimal number between 0 and 1, but it will never be exactly 0 or 1.
> Here’s an example:

```
const randomNum = Math.random();


console.log(randomNum); // Output: a random decimal number
        between 0 and 1
```

In this example, the Math.random() method returns a random decimal number between 0 and 1. The value of **randomNum** is then assigned the value of that random number, which is logged to the console.
>
The Math.random() method can be useful when you need to generate a random number for use in your code. For example, if you’re building a game and need to generate a random number for the player’s starting position, you can use the Math.random() method to generate a random number between two given values.
>

Math.abs()

> This is another useful Math method which returns the absolute value of a number (the distance of the number from zero). If the number is positive, the absolute value is the same as the number itself. If the number is negative, the absolute value is the opposite of the number.
>
> The **Math.abs()** method takes one argument, which is the number whose absolute value you want to find. It's important to note that this argument can be any valid JavaScript expression that evaluates to a number.
>
> Here's an example of using the **Math.abs()** method:

```
// Absolute value of a number
const num = -10;
const absNum = Math.abs(num);
console.log(absNum); // Output: 10
```

In this example, we have a variables **num**, which contains the number -10. We then use the **Math.abs()** method to find the absolute value of the number, and store the results in **absNum**.
>
When we log **absNum** to the console, we get the value 10. This is because the absolute value of **num** is 10 (which is the opposite of the number -10).
>
Here's another example that shows how the Math.abs() method can be used with an expression:

```
// Absolute difference between two numbers
const num1 = 5;
const num2 = 8;
const absoluteDiff = Math.abs(num1 - num2);
console.log(absoluteDiff); // Output: 3
```

In this example, we have two variables, **num1** and **num2**, which contain the numbers 5 and 8, respectively. We then subtract **num1** from **num2** to get the difference between the two numbers, which is -3. However, since we want the distance between the two numbers (i.e., the absolute value of the difference), we use the **Math.abs()** method which gives us the value 3.
>

In summary, the **Math.abs()** method in JavaScript is a useful tool for finding the absolute value of a number or expression, and can be used in a wide variety of contexts.

Combining Math methods

> You can also combine Math methods to perform more complex mathematical operations. For example, if you need to generate a random number between two given values, you can use the **Math.random()** method in combination with **Math.floor()**.
>
> Here’s an example:

```
const min = 1;
const max = 10;
const randomNum = Math.floor(Math.random() * (max - min + 1)) +
        min;



console.log(randomNum); // Output: a random integer between 1
        and 10 (inclusive)
```

In this example, the **min** variable is assigned a value of 1, and the **max** variable is assigned a value of 10. The **Math.random()** method is used to generate a random decimal number between 0 and 1, which is then multiplied by the difference between **max** and **min** plus one (in this case, 10 - 1 + 1 = 10). The result of this operation is a random decimal number between 0 and 10, which is then rounded down to the nearest integer using the **Math.floor()** method. Finally, 1 is added to the rounded down value to ensure that the result is between 1 and 10, inclusive. The value of **randomNum** is then logged to the console.
>

Additional Resources

Additional Resources for Day 1

This page provides a list of external resources that you can use to enhance your understanding of concepts learned in Day 1.

Modern Operators, Loops, and Timers

MDN Web Docs: Spread syntax
MDN Web Docs: Rest parameters
MDN Web Docs: for…of loop
MDN Web Docs: for…in loop:
MDN Web Docs: setInterval
MDN Web Docs: setTimeout:

Destructuring

JS Destructuring in 100 Seconds

Destructuring Assignment

JS Math

JavaScript Math Object

Conclusion & Takeaway

> In this lesson, we’ve covered three commonly used Math methods in JavaScript: **Math.floor()**, **Math.ceil()**, and **Math.random()**. **Math.floor()** can be used to round a number down to the nearest integer, **Math.ceil()** can be used to round a number up to the nearest integer, and **Math.random()** can be used to generate a random number. We’ve also learned how to combine these methods to perform more complex operations.

Practice Time

Work alone or with a partner to complete the following practice exercises. Try to come up with solutions first, but If you get stuck you can always refer back to the material in this lesson or ask an instructor or TA for help. Happy coding!

Practice Exercise 1:

1. Declare a variable **price** and assign it a value of 7.99.
2. Use the Math.floor() method to round the value of **price** down to the nearest integer.
3. Log the rounded down value of **price** to the console.

Practice Exercise 2:

1. Declare a variable **cupsOfFlour** and assign it a value of 2.5.
2. Use the Math.ceil() method to round the value of **cupsOfFlour** up to the nearest integer.
3. Log the rounded up value of **cupsOfFlour** to the console.

Practice Exercise 3:

1. Declare a variable **randomNum** and assign it a value using the Math.random() method.
2. Use the Math.floor() method to round the value of **randomNum** down to the nearest integer between 1 and 10.
3. Log the rounded down value of **randomNum** to the console.

Practice Exercise 4:

1. Write a JavaScript function that takes two arguments, **a** and **b**, and returns the absolute difference.
2. The function should calculate the absolute difference between **a** and **b** using the **Math.abs() method**.

Practice Exercise 5:

1. Declare a variable **min** and assign it a value of 5.
2. Declare a variable **max** and assign it a value of 15.
3. Use the Math.random() method in combination with Math.floor() and some basic arithmetic operations to generate a random integer between **min** and **max** (inclusive).
4. Log the random integer to the console.

Submit your code for all exercises:

<!–
Practice Exercises - Possible Solutions
Practice Exercise 1:

```
const price = 7.99;
const roundedDown = Math.floor(price);


console.log(roundedDown); // Output: 7
```

Practice Exercise 2:

```
const cupsOfFlour = 2.5;
const roundedUp = Math.ceil(cupsOfFlour);


console.log(roundedUp); // Output: 3
```

Practice Exercise 3:

```
const randomNum = Math.random();
const roundedDown = Math.floor(randomNum * 10) + 1;


console.log(roundedDown); // Output: a random integer between 1
and 10 (inclusive)
```

Practice Exercise 4:

```
const min = 5;
const max = 15;
const randomNum = Math.floor(Math.random() * (max - min + 1)) +
min;


console.log(randomNum); // Output: a random integer between 5
and 15 (inclusive)
```

–>

Mini Project: Random Quote Generator

> ## Goals
>
> By the end of this lesson, you will:
>
> - Get more practice with JavaScript concepts you've learned so far
> - Build a simple random quote generator
>
> ## Getting Started
>
> You will see three files open in Codio: **index.html**, **style.css**, and **script.js**.
>
> As per usual the HTML and CSS files are already complete, and we'll just be focusing on writing the JavaScript.
>
> ## Create the functionality
>
> Let's start off by getting the DOM elements we need:

```
const button = document.getElementById("new-quote-btn");
const quoteDiv = document.getElementById("quote-output");
const authorDiv = document.getElementById("author-output");
```

Now we need to create an array of predefined quote objects which we will select from randomly:

```
const quotes = [
  {
    quote: "The only way to do great work is to love what you
        do.",
    author: "— Steve Jobs"
  },
  {
    quote: "Believe you can and you're halfway there.",
    author: "— Theodore Roosevelt"
  },
  {
    quote: "I have not failed. I've just found 10,000 ways that
        won't work.",
    author: "— Thomas Edison"
  },
  {
    quote: "It does not matter how slowly you go as long as you
        do not stop.",
    author: "— Confucius"
  },
  {
    quote: "Success is not final, failure is not fatal: it is
        the courage to continue that counts.",
    author: "— Winston Churchill"
  },
  {
    quote: "The only true wisdom is in knowing you know
        nothing.",
    author: "— Socrates"
  }
]
```

Next, let’s attach an event listener to the button element which listens for a click event.
>
What we will need to do here is to first pick a quote at random from the array by combining the **floor()** and **random()** Math methods and multiplying that by the length of the array so that the random number which gets generated will be a valid index for our quotes array.
>
Then we’ll use an **if/else** statement to determine whether the **innerText** of our quote output element already contains the text of our randomly selected quote. If it doesn’t, then we’ll go ahead and set the **innerText** of both the quote output and the author output; if it does, then we will get a new random quote before updating the **innerText** of our DOM elements. This way, the same quote won’t come up twice in a row.
>

```
button.addEventListener("click", function() {
  let randomQuote = quotes[Math.floor(Math.random() *
        quotes.length)];



  if (quoteDiv.innerText !== randomQuote.quote) {
    quoteDiv.innerText = randomQuote.quote;
    authorDiv.innerText = randomQuote.author;
  } else {
    randomQuote = quotes[Math.floor(Math.random() *
        quotes.length)]
    quoteDiv.innerText = randomQuote.quote;
    authorDiv.innerText = randomQuote.author;
  }
});
```

> And there you have it: our random inspirational quote generator!

Conclusion & Takeaway

> In this section, we practiced a few of the concepts we learned so far and built a random quote generator, which selects a quote at random from a predefined array and displays the quote and its author on the page. We utilized an event listener, an **if/else** statement, the **innerText** property, and a couple new concepts we learned in this module – **Math.floor()** and **Math.random()**.

> <!–JS

~~~
```
const button = document.getElementById(“new-quote-btn”);
const quoteDiv = document.getElementById(“quote-output”);
const authorDiv = document.getElementById(“author-output”);

const quotes = [
{
quote: “The only way to do great work is to love what you do.”,
author: “— Steve Jobs”
},
{
quote: “Believe you can and you’re halfway there.”,
author: “— Theodore Roosevelt”
},
{
quote: “I have not failed. I’ve just found 10,000 ways that won’t work.”,
author: “— Thomas Edison”
},
{
quote: “It does not matter how slowly you go as long as you do not stop.”,
author: “— Confucius”
},
{
quote: “Success is not final, failure is not fatal: it is the courage to continue
that counts.”,
author: “— Winston Churchill”
},
{
quote: “The only true wisdom is in knowing you know nothing.”,
author: “— Socrates”
}]

button.addEventListener(“click”, function() {
let randomQuote = quotes[Math.floor(Math.random() * quotes.length)];
~~~

~~~
if (quoteDiv.innerText !== randomQuote.quote) {
quoteDiv.innerText = randomQuote.quote;
authorDiv.innerText = randomQuote.author;
} else {
randomQuote = quotes[Math.floor(Math.random() * quotes.length)]
quoteDiv.innerText = randomQuote.quote;
authorDiv.innerText = randomQuote.author;
}
});
```–>
~~~

Working with Strings

> ### Goals
>
> By the end of this lesson, you should understand:
>
> - The string data type and several of its properties and methods
> - ES6 Template Literals and interpolation
> - The basics of Regex
>
> ### Introduction
>
> So far in this course, we've covered many different data types in JavaScript. Now, we're going to cover another key data type in depth (including some new features that were added to it in ES6): strings. Strings, basically, are sequences of characters enclosed in either single or double quotation marks, or backticks (``). In this lesson, we will explore string properties and methods, ES6 template literals and interpolation, and the basics of Regex – which is a way of testing if strings match certain patterns. By the end of this lesson, you should have a good understanding of these concepts and be able to use them effectively in your JavaScript projects.

Properties and Methods

> Strings, like arrays and objects, have a number of properties and methods that can be used to manipulate them. In this section, we will cover some of the most commonly used string properties and methods.
>
> ### Properties
>
> - **length**: Returns the length of a string.
> - **constructor**: Returns a reference to the string's constructor function.
>   ### Methods
> - **charAt()**: Returns the character at a specified index in a string.
> - **concat()**: Joins two or more strings and returns a new string.
> - **indexOf()**: Returns the index of the first occurrence of a specified value in a string.
> - **lastIndexOf()**: Returns the index of the last occurrence of a specified value in a string.
> - **replace()**: Searches a string for a specified value, or a regular expression, and returns a new string where the specified value or regular expression has been replaced with another string.
> - **slice()**: Extracts a section of a string and returns a new string.
> - **split()**: Splits a string into an array of substrings.
> - **toLowerCase()**: Converts a string to lowercase.
> - **toUpperCase()**: Converts a string to uppercase.
> - **trim()**: Removes whitespace from both ends of a string.
>
> Let's look at some examples of how to use some of these properties and methods.

```
let myString = "Hello, world!";
console.log(myString.length); // Output: 13
console.log(myString.charAt(4)); // Output: o
console.log(myString.concat(" How are you?")); // Output: Hello,
        world! How are you?
console.log(myString.indexOf("world")); // Output: 7
console.log(myString.replace("world", "John")); // Output:
        Hello, John!
console.log(myString.slice(7, 12)); // Output: world
console.log(myString.split(",")); // Output: ["Hello", "
        world!"]
console.log(myString.toLowerCase()); // Output: hello, world!
console.log(myString.toUpperCase()); // Output: HELLO, WORLD!


let myString2 = " Hello again, world! ";
console.log(myString2.trim()); // Output: Hello again, world!
```

Template Literals and Interpolation

In ES6 (ECMAScript 2015), a new feature called template literals was introduced. Template literals allow you to create strings that contain placeholders, which can be replaced with values at runtime. Template literals are enclosed in backticks (`) instead of single or double quotes. Placeholders are indicated by wrapping them in curly braces (${}`).
>
Template literals offer several advantages over traditional string concatenation. They are more readable and allow for easier formatting. They also allow you to use expressions inside the placeholders, which can be very useful in certain situations.
>
Here's an example of how to use template literals:

```
let firstName = "John";
let lastName = "Doe";
let fullName = `${firstName} ${lastName}`;


console.log(fullName); // Output: John Doe


let num1 = 10;
let num2 = 20;


console.log(`The sum of ${num1} and ${num2} is ${num1 + num2}`);
// Output: The sum of 10 and 20 is 30
```

As you can see in the example above, template literals make it easy to include variables and expressions in strings. This can be very helpful when you need to dynamically generate strings based on user input or other factors.

A Primer on Regular Expressions (Regex)

Another important topic in JavaScript, and one which can prove very useful in a number of different situations (especially validating user input, which we will cover a little later in the course) is regular expressions – most often referred to as "regex".

A regular expression is a pattern that can be used to match text. Regular expressions are used in many programming languages, including JavaScript, to search for, replace, or extract text from a string. Regular expressions are powerful tools for text manipulation and can be used to perform complex search and replace operations with just a few lines of code. In JavaScript, regular expressions are represented by the `RegExp` object. Regular expressions are defined using a pattern, which is enclosed in forward slashes (`/`).

Here are some basic regular expression patterns:
- /**hello**/: Matches the string "hello".
- /**[a-z]**/: Matches any lowercase letter from "a" to "z".
- /**[A-Z]**/: Matches any uppercase letter from "A" to "Z".
- /**[0-9]**/: Matches any digit from 0 to 9.
- /**[a-zA-Z0-9]**/: Matches any alphanumeric character.
>
Regular expressions also have special characters that represent certain classes of characters, such as:
. : Matches any character except a newline character.
: Matches any digit.
*: Matches any word character (i.e., any alphanumeric character or underscore).
***: Matches any whitespace character.
>
As you can see in the example above, regular expressions can be used to search for specific patterns in a string. In this example, we searched for the string "fox", any vowel, any digit, any whitespace character, and any alphanumeric character.
>
Let's look at some examples using some of the patterns listed above. For our string in this example, we're going to use what's called a pangram, which is a sentence that contains all the letters of the alphabet:

```
let myString = "The quick brown fox jumps over the lazy dog.";
let regex1 = /fox/;


console.log(regex1.test(myString)); // Output: true


let regex2 = /[aeiou]/;
console.log(regex2.test(myString)); // Output: true


let regex3 = /\d/;
console.log(regex3.test(myString)); // Output: false


let regex4 = /\s/;
console.log(regex4.test(myString)); // Output: true


let regex5 = /\w/;
console.log(regex5.test(myString)); // Output: true
```

Conclusion & Takeaway

> In this lesson, we’ve covered string properties and methods, ES6 template literals and interpolation, and the basics of regex. These are fundamental concepts in JavaScript and are essential to understanding how to work with strings in the language. By understanding these concepts and practicing with the exercises provided, you’ll be well on your way to becoming proficient in working with strings in JavaScript.

Practice Time

Work alone or with a partner to complete the following practice exercises. Try to come up with solutions first, but if you get stuck you can always refer back to the material in this lesson or ask an instructor or TA for help. Happy coding!

1. Write a JavaScript function that takes a string as input and returns the reverse of the string. For example, if the input is “hello”, the output should be “olleh”.
2. Write a JavaScript function that takes two strings as input and returns a new string that concatenates the two input strings with a space in between. For example, if the inputs are “John” and “Doe”, the output should be “John Doe”.
3. Write a JavaScript function that takes a string as input and returns the number of vowels in the string. For this exercise, consider the following letters as vowels: a, e, i, o, and u.
4. Write a JavaScript function that takes a string as input and returns the same string with all vowels replaced with the letter “o”. For example, if the input is “hello”, the output should be “hollo”.
5. Write a JavaScript function that takes a string as input and returns the same string with all whitespace characters removed.
6. Write a JavaScript function that takes a string as input and returns true if the string is a palindrome (i.e., the string is the same when read forwards and backwards), and false otherwise. For example, if the input is “racecar”, the output should be true.
7. Write a JavaScript function that takes a string as input and returns true if the string contains only alphanumeric characters, and false otherwise. For this exercise, consider alphanumeric characters to be any letter or digit.
8. Write a JavaScript function that takes a string as input and returns an array of all the words in the string. For this exercise, consider a word to be any sequence of characters separated by whitespace.
9. Write a JavaScript function that takes a string as input and returns a new string where the first letter of each word is capitalized. For example, if the input is “the quick brown fox”, the output should be “The Quick Brown Fox”.
10. Write a JavaScript function that takes a string as input and returns true if the string contains at least one uppercase letter, one lowercase letter, and one digit, and false otherwise.

Submit your code for all of these exercises.

Working with Dates

> ### Goals
>
> By the end of this lesson, you should understand:
>
> - The ECMAScript/Unix epoch, UTC and why they're both important
> - The ISO 8601 standard date/time format
> - How to create and work with Date objects in JavaScript
>
> ### Introduction
>
> JavaScript's **Date** object provides a powerful set of features to work with dates and times. In this lesson, we will cover important concepts such as the ECMAScript/UNIX epoch, UTC, local time zone and the UTC offset, as well as the ISO 8601 format. Understanding these concepts is crucial for effectively working with dates in JavaScript. We will also explore the basics of creating Date objects and how to utilize formatting methods to customize the output. By the end of this lesson, you will have the knowledge and tools to manipulate dates and times in your JavaScript applications.

A Few Important Considerations

Before diving into this lesson, there are a few key concepts which we will need to cover first, as JavaScript dates rely heavily on them. They are: the ECMAScript/Unix epoch, UTC, local time zone and UTC offset, and the ISO 8601 format.

ECMAScript/UNIX epoch

ECMAScript and UNIX time both use a common reference point to measure time (known as the epoch). The epoch is defined as midnight on January 1, 1970, in what is called Coordinated Universal Time (UTC). In JavaScript, as you will soon see, creating a date object using the Date constructor will return the number of milliseconds since the UNIX epoch. This reference point of the ECMAScript/UNIX epoch can be useful for measuring time intervals and determining the difference between dates, but it can also take some time to get used to thinking about dates in this way.

UTC

Coordinated Universal Time (UTC) is the international standard for timekeeping and the basis for the time zones used around the world. You might have heard this referred to in other contexts as Greenwich Mean Time or GMT; however, while there is no time difference between UTC and GMT, GMT is a time zone and UTC is the time standard. Dates objects in JavaScript (as well as their methods) make heavy use of UTC precisely because it provides a means of working with dates and times in a standardized way.

Local time zone and UTC offset

The local time zone is the time zone in which the user is located. The UTC offset is the difference between the local time and UTC.

ISO 8601 format

The ISO 8601 format is an international standard for representing dates and times. The format is designed to be unambiguous and easy to read, and includes the date and time in a specific order with separators between each component. The format can be used with both UTC and local time. There is a way to format dates according to ISO 8601, but we won’t cover that in this lesson. If you want to learn how to format dates in this way, see the Asynchronous Elective Learning (AEL) assignment in this module titled ‘More on Dates and Date Methods’. For now, let’s just see an example of what an ISO date looks like:

```
"2023-03-08T03:21:30.689Z"
```

Creating Date Objects

> With those initial important concepts covered, let's move on to see how to create date objects in JavaScript.

Here we will cover only the most basic way of creating date objects, but there are actually several. If you wish to learn more about the additional ways of creating date objects, see the Asynchronous Elective Learning (AEL) assignment in this module titled 'More on Dates and Date Methods'.
>
Creating a date object with no arguments
The simplest way to create a new Date object is to use the constructor with no arguments. This will create a new Date object representing the current date and time. The output will be a UNIX timestamp, which is the number of seconds that have elapsed since January 1, 1970, at 00:00:00 UTC.

Try putting the code into the editor to view the output.

```
let currentDate = new Date();
console.log(currentDate); // Output: current date and time as a
        UNIX timestamp
```

Obviously, the UNIX timestamp output is not very convenient to work with. Fortunately, we can format this output to be more readable and useful for the majority of use cases. We'll take a look at some of the different formatting options next.

Formatting Dates

> The Date object provides several methods for formatting dates and times in different ways. Here are some commonly used formatting methods:
> - **toLocaleString()** - returns a string representing the date and time in the local time zone, using a format appropriate for the user's locale.
> - **toLocaleDateString()** - returns a string representing the date portion of the date in the local time zone, using a format appropriate for the user's locale.
> - **toLocaleTimeString()** - returns a string representing the time portion of the date in the local time zone, using a format appropriate for the user's locale.
>
> Let's take a closer look at each of these methods, including their optional arguments.

toLocaleString()

The **toLocaleString()** method returns a string representing the date and time in the local time zone, using a format appropriate for the user's locale. This method accepts two optional arguments:

```
let myDate = new Date();


console.log(myDate.toLocaleString()); // Output: current date
        and time in local format
console.log(myDate.toLocaleString('en-US')); // Output: current
        date and time in local format, with American English
        conventions
console.log(myDate.toLocaleString('en-US', {timeZone:
        'America/Los_Angeles'})); // Output: current date and
        time in local format, with American English conventions,
        in the America/Los_Angeles time zone
```

The first argument is a locale string that specifies the language and country/region for the output. For example, 'en-US' represents American English, while 'fr-FR' would represent French in France. You get the idea.

For a comprehensive list of locale codes, check out http://www.lingoes.net/en/translator/langcode.htm. Note that some of these may not be supported.

The second argument of **toLocaleString()** is an options object, which lets you make various customizations. Click here to see a list of the available options.
>

toLocaleDateString():

The **toLocaleDateString()** method returns a string representing the date portion of the date in the local time zone, using a format appropriate for the user's locale. This method also accepts the same two optional arguments:

```
let myDate = new Date();



console.log(myDate.toLocaleDateString()); // Output: current
        date in local format
console.log(myDate.toLocaleDateString('en-US')); // Output:
        current date in local format, with American English
        conventions
console.log(myDate.toLocaleDateString('en-US', { timeZone:
        'America/New_York' })); /// Output: current date and
        time in local format, with American English conventions,
        in the America/New_York time zone
```

toLocaleTimeString():

The **toLocaleTimeString()** method returns a string representing the time portion of the date in the local time zone, using a format appropriate for the user's locale. This method also accepts the same two optional arguments:

```
let myDate = new Date();

console.log(myDate.toLocaleTimeString()); // Output: current
        time in local format
console.log(myDate.toLocaleTimeString('en-US')); // Output:
        current time in local format, with American English
        conventions
```

Here's an example showing more options for customizing the output with the options object argument:

```
let myDate = new Date();
let options = { weekday: 'long', year: 'numeric', month: 'long',
        day: 'numeric' };



console.log(myDate.toLocaleDateString('en-US', options)); //
        Output: Monday, March 8, 2023
```

In this example, we’ve created an options object that specifies the desired format for the date string. The ‘weekday’, ‘year’, ‘month’, and ‘day’ properties specify which parts of the date to include in the output, and ‘long’ specifies the format for each part. The resulting output is a string that includes the day of the week, the full month name, the day of the month, and the year, separated by commas.

Conclusion and Takeaway

> In this lesson, we first covered some important preliminary concepts necessary for working with dates, namely the ECMAScript/Unix epoch, UTC, local time zone and UTC offset, and the ISO 8601 format. Then we dove into working with the Date object – namely, how to create date objects and utilize different formatting methods like **toLocaleString()**, **toLocaleDateString()**, and **toLocaleTimeString()**. By using these methods with their optional arguments, you can customize the output to meet your needs. By understanding the concepts we went over in this lesson, you can effectively work with and manipulate dates and times in your JavaScript applications.

Additional Resources

Additional Resources for Day 2

This page provides a list of external resources that you can use to enhance your understanding of concepts learned in Day 2.

Strings

JavaScript String Reference

JavaScript String Methods

JavaScript RegExp Reference

JavaScript Template Literals

JS Date

JavaScript Date Objects

List of locale codes

Date formatting options

More on Dates and Date Methods (AEL)

> ### Goals
>
> By the end of this lesson, you should understand:
>
> - How to create Date objects using UNIX timestamps, ISO 8601 strings, and individual numeric segments
> - Common getters and setters for Date objects and how to use them to manipulate the year, month, and day
>
> ### Introduction
>
> In the main lesson on JavaScript **Date** objects, we covered the basics of how to create them as well as some commonly used methods to format them. This asynchronous elective learning lesson will take things a step further and cover three additional ways to create Date objects: using a UNIX timestamp, an ISO 8601 string, or individual numeric segments. We'll also explore commonly used getters and setters that allow you to retrieve and modify specific components of a Date object, and wrap things up by going over a few more methods to format dates which JavaScript makes available to us.

Creating Date Objects

> ### Creating date objects with a UNIX timestamp argument
>
> UNIX timestamps are the number of seconds that have elapsed since January 1, 1970, at 00:00:00 UTC. To create a new Date object from a UNIX timestamp, pass the timestamp as an argument to the Date constructor:

```
let unixTimestamp = 1636279462;
let dateFromUnixTimestamp = new Date(unixTimestamp * 1000); //
Multiplying by 1000 to convert seconds to milliseconds


console.log(dateFromUnixTimestamp); // Output: Date object
representing the date and time corresponding to the UNIX
timestamp
```

Creating date objects with an ISO 8601 string argument

To create a new Date object from an ISO 8601 formatted string, pass the string as an argument to the Date constructor:

```
let isoString = '2022-11-07T14:30:00Z'; // Z indicates UTC time
let dateFromISOString = new Date(isoString);


console.log(dateFromISOString); // Output: Date object
representing the date and time corresponding to the ISO 8601
formatted string
```

Creating date objects with the numeric segments

You can also create a new Date object by specifying the different segments of the date and time as separate numeric arguments to the Date constructor. The arguments are in the following order: **year**, **month** (0-11), **day**, **hour**, **minute**, **second**, and **millisecond**.

```
let year = 2023;
let month = 2; // March (0-based indexing)
let day = 8;
let hour = 12;
let minute = 30;
let second = 0;
let millisecond = 0;


let dateFromParts = new Date(year, month, day, hour, minute,
second, millisecond);


console.log(dateFromParts); // Output: Date object representing
March 8, 2023, at 12:30:00
```

Getters and Setters

> Getters and setters are methods that allow you to get and set the values of various properties of a Date object. Here are some examples of commonly used getters and setters:
> - **getFullYear()** - gets the year of a date as a four-digit number.
> - **setFullYear()** - sets the year of a date to a specified value.
> - **getMonth()** - gets the month of a date as a number (0-11).
> - **setMonth()** - sets the month of a date to a specified value (0-11).
> - **getDate()** - gets the day of the month of a date as a number (1-31).
> - **setDate()** - sets the day of the month of a date to a specified value (1-31).
> - **getHours()** - gets the hour of a date as a number (0-23).
> - **setHours()** - sets the hour of a date to a specified value (0-23).
> - **getMinutes()** - gets the minutes of a date as a number (0-59).
> - **setMinutes()** - sets the minutes of a date to a specified value (0-59).
> - **getSeconds()** - gets the seconds of a date as a number (0-59).
> - **setSeconds()** - sets the seconds of a date to a specified value (0-59).
> - **getMilliseconds()** - gets the milliseconds of a date as a number (0-999).
> - **setMilliseconds()** - sets the milliseconds of a date to a specified value (0-999).
>
> Here's some examples of how to use these to get and set the year, month, and day of a Date object:

```
let myDate = new Date();

console.log(myDate.getFullYear()); // Output: current year

myDate.setFullYear(2024);

console.log(myDate.getFullYear()); // Output: 2024
console.log(myDate.getMonth()); // Output: current month (0-
based indexing)

myDate.setMonth(6); // Set month to July

console.log(myDate.getMonth()); // Output: 6 (July)
console.log(myDate.getDate()); // Output: current day of the
month

myDate.setDate(15);

console.log(myDate.getDate()); // Output: 15
```

Formatting Dates

> In addition to the formatting methods covered in the main lesson, there are a few others which JavaScript makes available to us:
> - **toISOString()** - returns a string representing the date and time in ISO format (e.g. “2023-03-08T12:30:00.000Z”)
> - **toUTCString()** - returns a string representing the date and time in UTC (GMT)
> - **toString()** - returns a string representing the date and time in a default format that varies depending on the particular JavaScript engine or implementation
> - **toDateString()** - returns a string representing the date portion of the date object, in a default format that varies depending on the particular JavaScript engine or implementation
> - **toTimeString()** - returns a string representing the time portion of the date object, in a default format that varies depending on the particular JavaScript engine or implementation
>
> Let’s look at some examples.

toISOString()

```
let myNewDate = new Date()
console.log(myNewDate.toISOString()) // Output: current date and
time in ISO format
```

toUTCString()

```
console.log(myNewDate.toUTCString()) // Output: current date and
time in UTC
```

toString()

```
console.log(myNewDate.toString()) // Output: current date and
time in implementation-dependent format
```

toDateString()

```
console.log(myNewDate.toDateString()) // Output: current date in
implementation-dependent format
```

toTimeString()

```
console.log(myNewDate.toTimeString()) // Output: current time in
implementation-dependent format
```

Additional Resources

Additional Resources for Day 2

This page provides a list of external resources that you can use to enhance your understanding of concepts learned in Day 2.

Strings

JavaScript String Reference

JavaScript String Methods

JavaScript RegExp Reference

JavaScript Template Literals

JS Date

JavaScript Date Objects

Conclusion and Takeaway

> In this lesson, we learned how to create Date objects using UNIX timestamps, ISO 8601 strings, and individual numeric segments. We also explored various getters and setters that enable us to retrieve and modify specific properties of a Date object, such as the year, month, and day. In addition, we looked at a few more date formatting methods. Combined with the concepts covered by the main lesson, you should now have a solid arsenal of tools to work with dates and times in JavaScript.

Practice Time

> Work alone or with a partner to complete the following practice exercises. Try to come up with solutions first, but If you get stuck you can always refer back to the material in this lesson or ask an instructor or TA for help. Happy coding!

1. Create a Date object with the current date and time, and then log it to the console using toISOString().
   >
2. Create a Date object with the current date and time. Use the getFullYear() method to get the current year, and then use the setFullYear() method to set the year to 2024. Finally, log the new date to the console.
   >
3. Create a Date object with the current date and time, and then log the date and time in the following formats.
   - toLocaleString(): returns a string representing the date and time in the local time zone
   - toLocaleDateString(): returns a string representing the date portion of the date in the local time zone
   - toLocaleTimeString(): returns a string representing the time portion of the date in the local time zone

4. Create a Date object with the current date and time. Use the getMonth() method to get the current month, and then use the setMonth() method to set the month to 11 (December). Finally, log the new date to the console.
5. Create a Date object with the current date and time, and then log the date and time in the following formats, with American English conventions:
    - Local date and time format
    - Local date format
    - Local time format
6. Create a Date object for the date “January 15, 2022 3:45:30 PM” using the numeric arguments that make up the different parts of the date and time, and then log it to the console.
7. Create a Date object for the date “March 8, 2023 9:30:00 AM”. Use the getHours() method to get the current hour, and then use the setHours() method to set the hour to 14. Finally, log the new date to the console.
8. Create a Date object with the current date and time. Use the toLocaleString() method with the “en-US” argument and the options argument to format the date and time with the following properties:
    - weekday: long
    - year: numeric
    - month: long
    - day: numeric
    - hour: numeric
    - minute: numeric
    - second: numeric
9. Create a Date object for the date “2022-06-30T12:45:30Z” using an ISO 8601 formatted string argument, and then log it to the console.
10. Create a Date object for the date “January 1, 2022 12:00:00 AM”. Use the getDate() method to get the current day of the month, and then use the setDate() method to set the day of the month to 15. Finally, log the new date to the console.

Submit all of your code for these exercises.

Mini Project: JavaScript Age Calculator

> ### Goals
>
> By the end of this lesson, you will:
>
> - Have practiced many of the concepts learned so far this week
> - Have built a simple JavaScript Age Calculator
>
> ### Introduction
>
> In this lesson we're going to create a small project implementing concepts that we've learned throughout this module. We will build a simple JavaScript age calculator, where a user can enter their birthday and the program which check to make sure the entered value matches a regular expression, then show their calculated age on the screen.

Getting Started

> You will see three files open in Codio: **index.html**, **style.css**, and **script.js**.
>
> The HTML and CSS files are already completed. All of our code will be written in the **script.js** file.

Create the functionality

Like always, we should begin by getting the necessary elements from the DOM. For this mini-project, we'll need the form element, the birthdate input element, and the age output element:

```
// get DOM elements
const form = document.querySelector('form');
const birthdateInput = document.querySelector('#birthdate');
const ageOutput = document.querySelector('#age');
```

Next we need to define two event listeners for our form, one for a "submit" event and one for a "reset" event. We'll be discussing these more when we cover forms in depth next week. For now, let's add the following code to **script.js** which will be our submit listener:

```
// listen for form submit event
form.addEventListener('submit', function(e) {
  e.preventDefault();


  // get input value
  const birthdateString = birthdateInput.value;


  // validate input with regex
  const dateRegex = /^\d{2}\/\d{2}\/\d{4}$/;
  if (!dateRegex.test(birthdateString)) {
    alert('Please enter a valid birthdate in MM/DD/YYYY
        format.');
    return;
  }


  // parse input as date object
  const birthdate = new Date(birthdateString);


  // calculate age
  const ageInMilliseconds = Date.now() - birthdate.getTime();
  const ageDate = new Date(ageInMilliseconds);
  const age = Math.abs(ageDate.getUTCFullYear() - 1970);


  // display age
  ageOutput.textContent = `You are ${age} years old.`;
});
```

There’s a few different things going on here. First, we’re grabbing the **event** as a parameter to the listener function which we’re calling **e** for short. Normally when a form is submitted the browser window reloads, but we don’t want that to happen before we run our logic. That’s why we’re calling the **preventDefault()** method on **e**.
>
Next we’re defining a variable to store the value of the birthdate input using **birthdateInput.value**, and a regex expression to make sure the birthdate input value is entered in the format we want. If the birthdate is not formatted correctly, we’re using an alert to let the user know.
>
Then, we’re creating some variables related to the birthdate and age using the Date object, and performing a calculation using **Math.abs()** to get the absolute value which represents the user’s age so we can display it on our web page accordingly.

Here’s a breakdown of this calculation:

1. `const ageInMilliseconds = Date.now() - birthdate.getTime();` - Calculates the difference in milliseconds between the current time and the birthdate
2. `const ageDate = new Date(ageInMilliseconds);` - Creates a new Date object using the age in milliseconds. Since the Date object counts milliseconds from January 1, 1970, this Date object will represent how much time has elapsed since then, essentially giving us the “age” but offset by the 1970 epoch start.
3. `const age = Math.abs(ageDate.getUTCFullYear() - 1970);` - Calculates the actual age in years. **ageDate.getUTCFullYear():** will essentially return **1970** + **x** (where **x** is the number of years since 1970 for the **ageDate**). To get the actual age, then, we need to subtract 1970. Visualizing this as a mathematical equation, it would read `(1970 + x) - 1970 = x`. Hope you paid attention in algebra! 🤓
   >
   Lastly, we now need to create a reset event listener to clear the age output when the reset button is clicked:
   >

```
form.addEventListener('reset', function() {
  ageOutput.textContent = null;
})
```

> And that’s a wrap!

Conclusion & Takeaway

> We just finished creating a simple JavaScript age calculator which, in addition to reinforcing several of the concepts we've learned so far, also gave a sneak peak at other JavaScript features we will learn more about later – namely, how to work with forms, form events, and validation. We will cover those topics in more depth next week.

Events - Part 1

Goals

By the end of this lesson, you should understand:
>
- The purpose and usage of some of the window and document events, such as the **load**, **DOMContentLoaded**, and **readystatechange** events
- How to handle mouse and keyboard events
>
Introduction
One of the most powerful features of JavaScript is its ability to respond to a variety of different events. In this lesson, we will explore three significant events related to window and document interactions: **load**, **DOMContentLoaded**, and **readystatechange** events. These events allow developers to execute code at the right time during page loading, ensure smooth user experiences, and detect loading errors or issues. Additionally, we will also cover various mouse and keyboard events commonly used to respond to corresponding user interactions.
>

Window and Document Events

> Window and document events are events that are triggered when a user interacts with the browser window or the HTML document. Some useful window/document events include the **load** event, the **DOMContentLoaded** event and the **readystatechange** event (all of which we will look at more closely in a moment). Aside from these, other common events which we won't cover here include the **resize** event, which fires when the window is resized, and the **unload** event, which fires when the user leaves the page, and the **scroll** event, which fires when the user scrolls the page.

'load' Event

The **load** event is an event that fires on the **window** object rather than the **document** object. It occurs after all the HTML elements have been parsed and all the associated resources such as images, stylesheets, and scripts have been loaded as well. This event allows you to write code that will be executed only once the page has completely finished loading, ensuring a smooth user experience. Incidentally, the DOM load event can also be used to monitor page loading performance and optimize web pages for better performance.

Let's see a quick example of how to listen for the 'load' event. Add the following to **script.js**:

```
window.addEventListener("load", function () {
  console.log("Page loaded successfully!")
})
```

Now, go ahead and refresh the preview window and you should see the message "Page loaded successfully!".
>

'DOMContentLoaded' Event

The **DOMContentLoaded** event fires when the initial HTML document has been completely parsed, but before any stylesheets, images, etc. This event is most commonly used to trigger the execution of scripts that need to run once the DOM is ready, but not necessarily its additional resources. It can be used in combination with other events, such as the 'load' event. By using 'DOMContentLoaded', developers can ensure that their code will execute at the right time and avoid any potential conflicts between different scripts running on the page.

Let’s see a quick example of how to listen for the DOMContentLoaded event. Add the following to **script.js**:

```
document.addEventListener("DOMContentLoaded", function () {
  console.log("DOM content loaded successfully!")
})
```

Refresh the preview window and you should see the message “DOM content loaded successfully!”.
>

‘readystatechange’ Event

The **readystatechange** event occurs whenever the **readystate** attribute of the document changes. This event is also typically used to determine when a document has been completely loaded and can be interacted with, similar to ‘load’ and ‘DOMContentLoaded’. In addition, it can be used to detect errors or issues with loading resources, such as scripts, stylesheets, images, etc. and can help developers improve the user experience by providing feedback on loading progress and displaying helpful error messages when needed.

Let’s see a quick example of how to listen for the ‘readystatechange’ event. Add the following to **script.js**:

```
document.addEventListener("readystatechange", function () {
  console.log("Ready state changed: " + document.readyState)
})
```

Refresh the preview window, and you should actually see *two* messages logged to the console this time: “Ready state changed: interactive” and “Ready state changed: complete”.

This is because there are three states associated with the ‘readystatechange’ event: **loading** (which didn’t get logged), **interactive**, and **complete**. To help understand each of the above events and when they’re triggered, here’s the order (taken from the MDN web docs):
1. **readystate interactive**
2. **DOMContentLoaded**
3. **readystate complete**
4. **load**

Mouse and Keyboard Events

> Mouse events are events that are triggered when a user interacts with the mouse. Some common mouse events include the **click** event, which we've already seen and which fires when the user clicks the mouse, the **mouseover** event, which fires when the mouse pointer moves over an element, and the **mouseout** event, which fires when the mouse pointer moves away from an element.

Keyboard events are events that are triggered when a user interacts with the keyboard. Some common keyboard events include the **keydown** event, which fires when a key is pressed down, the **keyup** event, which fires when a key is released, and the **keypress** event, which fires when a key is pressed and released.

Let's now see some examples. Since we've already been using the **click** event, we'll focus on **mouseover** and **mouseout**. And as for keyboard events, while you may indeed find certain use cases for **keydown** and **keyup**, you'll likely find yourself using **keypress** more often than not. Thus, we'll only look at an example of **keypress**, but just know that the other two exist if you need to use them!

'mouseover' Event

We'll start by looking a very basic example of how we can handle the **mouseover** event. The code looks much the same as for **click** events, except we specify **mouseover** in the event listener:

```
const myDiv = document.getElementById("my-div")


myDiv.addEventListener("mouseover", function () {
  myDiv.style.backgroundColor = "blue"
})
```

Now, when we move the mouse over this div element, it will change from red to blue.

'mouseout' Event

Next, let's see an example of how to handle the **mouseout** event. Again, the code looks very similar, and we'll be using the same div element from the last example:

```
myDiv.addEventListener("mouseout", function () {
  myDiv.style.backgroundColor = "green"
})
```

With this additional event listener in place, when we first move the mouse onto the div element it turns blue, but then when we move the mouse *off* of the element it will turn green.

‘keypress’ Event

Next, let’s look at an example of how to handle the **keypress** event:

```
const myInput = document.getElementById("my-input")
const output = document.getElementById("output")


myInput.addEventListener("keypress", function (event) {
  output.textContent = `You pressed and released the
        ${event.key} key!`
})
```

In the code above, when any key on the keyboard is both pressed *and* released we access the name of that key with the **keypress** event and once again output a message in the ‘output’ element.

Conclusion & Takeaway

> In this lesson, we covered several important topics related to JavaScript events, including window and document events, mouse events, and keyboard events. We also went through code examples to help better understand these concepts. By understanding how to use events effectively in JavaScript, we can create dynamic and interactive web applications that respond to user interaction and provide a more engaging user experience. In the next lesson, we'll explore some more events we can make use of in our frontend code, as well as the important concepts of bubbling, capturing, and delegation.

Events - Part 2

Goals

By the end of this lesson, you should understand:
>
- The usage of different input and animation events
- The concepts of bubbling, capturing, and delegation
>
Introduction
In this lesson, we will explore some additional types of events in JavaScript: specifically, input and animation events. Input events are triggered when a user interacts with an input field such as a text input or checkbox, while animation events are triggered during the start, stop, or repetition of animations. We'll also go over the different event handling techniques of bubbling, capturing, and delegation, which are important concepts to know so that we can more efficiently *manage* events.

Input Events

> Input events are events that are triggered when a user interacts with an input field, such as a text input, checkbox, or radio button. Some common input events include the **change** event, which fires when the value of an input field changes, the **focus** event, which fires when the input field receives focus, and the **blur** event, which fires when the input field loses focus.

Let’s see some examples for each of these events.

Change Event

Here is an example of using the **change** event:

```
const mySelect = document.getElementById("my-select")
const changeOutput = document.getElementById("change-output")


mySelect.addEventListener("change", function (event) {
  changeOutput.textContent = `You selected
        ${event.target.value}!`
})
```

Here we are listen for the **change** event on a select element with the id “my-select”, and then update the text content of the output element to display a message saying which option was selected.

Focus Event

Now let’s look at an example of using the **focus** event:

```
const myInput = document.getElementById("my-input")
const focusBlurOutput = document.getElementById("focus-blur-
        output")

myInput.addEventListener("focus", function (event) {
  focusBlurOutput.textContent = "The input is in focus!"
})
```

We now listen for the **focus** event on an input field with the id “my-input”, and when that input receives focus (i.e. - when we click inside of it) we update the text content of our **focusBlurOutput** element to display a message saying that the input is in focus.

Blur Event

As the opposite to the **focus** event, the **blur** event will be triggered when the input no longer has focus:

```
myInput.addEventListener("blur", function (event) {
  focusBlurOutput.textContent = "The input no longer has focus!"
})
```

Here we add another event listener for the **blur** event on our same input field, and update the text content of our output element to display a message saying that the input no longer has focus.

Animation Events

> Animation events are events that are triggered when an animation starts, stops, or repeats. Some common animation events include the **animationstart** event, which fires when an animation starts, the **animationend** event, which fires when an animation ends, and the **animationiteration** event, which fires when an animation repeats.

Let’s see some examples for these events. We will only look at **animationstart** and **animationend** in this lesson, as you probably won’t find yourself using **animationiteration** that often. But, once again, it’s important to know that it exists if you need it!
>

Animationstart Event

Here is an example of how to handle the **animationstart** event:

```
const box = document.getElementById("box")
const output = document.getElementById("output")


box.addEventListener("animationstart", function (event) {
  output.textContent = "The animation has started!"
})
```

In this example, we listen for the **animationstart** event on an div with an id of “box”. When the element begins an animation, we update the text content of our output element to state that the animation has started.

Animationend Event

Here is an example of how to handle the **animationend** event:

```
box.addEventListener("animationend", function (event) {
  output.textContent = "The animation has ended!"
})
```

In this example, we listen for the **animationend** event on an div with an id of “box”. When the element completes an animation, we update the text content of our output element to state that the animation has ended.

Bubbling, Capturing, and Delegation

> Bubbling, capturing, and delegation are three techniques, or *approaches*, used to handle events in JavaScript.
>
> **Bubbling** is a technique where an event is first handled by the innermost element and then propagated up to the outer elements. This is the default behavior in JavaScript. As an example, if we have a button inside a div, and both have click event listeners, when we click the button, the button's click event listener will be called first, then the div's click event listener will be called.

Capturing is the opposite of bubbling, where the event is first handled by the outermost element and then propagated down to the inner elements. To use event capturing, you can pass a third argument to the **addEventListener()** method with the value of **true**.
>
```
element.addEventListener(eventType, eventHandler, true);
```
>
However, capturing is rarely used in practice.

Last but not least, **delegation** is a technique where we attach a single event listener to a parent element that will handle events triggered by its child elements. This is useful when we have many child elements that need to trigger the same event listener, as we can avoid attaching a listener to each child element.

Let's look at an example of how to use delegation to handle the **click** event on a list of items:

```
const myList = document.getElementById("myList")

myList.addEventListener("click", function (event) {
  if (event.target.tagName === "LI") {
    event.target.classList.toggle("selected")
  }
})
```

Here we use event delegation to handle the **click** event of the **ul** element and toggle the **selected** class on whichever **li** element triggered the event.

Conclusion & Takeaway

> Understanding and being able to effectively handle a great variety of different events, of which we’ve covered many (but not all!) in these last two lessons, adds yet more tools to our frontend development toolbox. The different ways of approaching event handling that we learned about – bubbling, capturing, and delegation – offer us greater control in managing these events, as well as improved performance and user experience in our applications. In the next lesson, we’ll start tackling some advanced DOM manipulation!

Practice Time

> Work alone or with a partner to complete the following practice exercises, attempting to complete as many as you can. Try to come up with solutions first, but if you get stuck you can always refer back to the material in this lesson or ask an instructor or TA for help. Happy coding!
> ### Exercise 1:

Add a **keydown** event listener to a text input field that changes the background color of the field to red when the user presses the “Shift” key and blue when the user presses the “Ctrl” key.

Exercise 2:

Use event delegation to handle the **click** event of multiple links, and change the color of the clicked link to red.
Exercise 3:

Create a web page that displays an unordered list of items and highlights each item when the user hovers over it, using the **mouseover** event.
>
Submit your code for all exercises.
>

Advanced DOM

> ### Goals
>
> By the end of this lesson, you should understand:
>
> - Many different ways to create and delete elements
> - How to traverse the DOM
>
> ### Introduction
>
> As you know by now, one of the essential aspects of JavaScript is manipulating the DOM (Document Object Model), and you already began learning DOM manipulation last week. In this lesson, in addition to reviewing a bit of what we covered previously, we will expand on that to discuss in depth some more advanced DOM manipulation techniques. In addition to looking at how we can create different elements and nodes, and how we can insert them into the DOM at different places, we'll also be covering DOM traversal – a fundamentally important skill to master in frontend development.

Creating Elements

> To begin with let's have a look at the different methods to **create** DOM elements:
>
> 1. **createElement**:
>   The createElement method allows you to create an HTML element dynamically in JavaScript. To create a new element, you need to specify its tag name using this method.

Add the following to **script.js**:

```
const newHeadingElement = document.createElement('h2')
console.log('Newly Created Element: ', newHeadingElement)
```

> 2. **createTextNode**:
>   The createTextNode method allows you to create a *text node* that can be added to an HTML element. To create a new text node, you need to specify the text content using this method.

Add the following to **script.js**:

```
const newText = document.createTextNode('Hello World!')
console.log('New Text Node: ', newText)
```

3. **appendChild**:
   The appendChild method allows you to add a child element to an existing DOM element. The child element itself can either be a new HTML element or an existing one. The appendChild method adds the child element as the *last* child of the parent element.

Add the following to **script.js**:

```
const parentElement = document.querySelector('#parent')
newHeadingElement.appendChild(newText)
parentElement.appendChild(newHeadingElement)

console.log('New innerHTML of Parent Element (after
        "appendChild"): ', parentElement.innerHTML)
```

> 4. **insertBefore**:
>   The insertBefore method allows you to insert a new child element before an existing child element of a parent element. The new child element can be created dynamically as part of the same code block, or an existing one can be used.

Add the following to **script.js**:

```
const existingElement = document.querySelector('#existing')
parentElement.insertBefore(newHeadingElement, existingElement)

console.log('New innerHTML of Parent Element (after
        "insertBefore"): ', parentElement.innerHTML)
```

> 5. **cloneNode**:
>   The cloneNode method allows you to create a copy of an existing DOM element. The new element will have the same attributes and child elements as the original element.

Add the following to **script.js**:

```
const clonedElement = existingElement.cloneNode(true)
clonedElement.innerText = '**CLONED ELEMENT** ' +
        clonedElement.innerText
clonedElement.setAttribute('id', 'cloned')
parentElement.appendChild(clonedElement)

console.log('New innerHTML of Parent Element (after
        "cloneNode"): ', parentElement.innerHTML)
```

Removing Elements

> Now that we've covered creating elements, let's see how we can remove them. There are two methods to cover here: **removeChild()** and **replaceChild()**. But first, as always, let's start by getting our DOM elements:

```
const parentElement = document.getElementById('parent')
console.log('Parent element: ', parentElement)

const firstChildElement = document.getElementById('first-child-
        element')
console.log('First child element: ', firstChildElement)
```

> ### removeChild Method
>
> The removeChild method is used to remove a child node from the DOM. It takes one argument, which is the child node to be removed. Add the following to **script.js**:

```
parentElement.removeChild(firstChildElement)
console.log('Parent element (after first child removed): ',
        parentElement)
```

In this example, we call the removeChild method on the parent element, passing the first child element as an argument. This will remove the child element from the DOM. Click the refresh button in the preview window to see the updated console output.
>

replaceChild method

The replaceChild method is used to replace an existing child node with a new child node. It takes two arguments, which are the new child node and the old child node, respectively. Add the following to **script.js**:

```
const secondChildElement = document.getElementById('second-
        child-element')
const newParagraph = document.createElement('p')
newParagraph.textContent = 'This is the new paragraph element'

console.log('Parent element (before second child replaced): ',
        parentElement)

parentElement.replaceChild(newParagraph, secondChildElement)
console.log('Parent element (after second child replaced): ',
        parentElement)
```

In this example, we first retrieve the second child element, and then create a new child element using the **createElement** method. We set the text content of the new child element using **textContent** property. Then we call the **replaceChild** method on the parent element, passing the new child element and old child element as arguments. This will replace the old child element with the new child element.

Click the refresh button in the preview window to see the updated console output.
>

DOM Traversal - Part One

> The term “DOM traversal” refers to the process of navtigating through the hierarchical tree structure of an HTML document. JavaScript gives us a variety of built-in methods and properties to traverse the DOM, and over the course of this page and the next we will look at several of them.

parentElement and *parentNode*

The **parentElement** and **parentNode** properties are used to get the parent node of an element. The only difference between these two properties is that **parentElement** returns only an element node, while **parentNode** can return any node type.

Go ahead and add the following to **script.js**:

```
const childDiv = document.getElementById('first-child-element')
const parentElementNode = childDiv.parentElement
const parentNode = childDiv.parentNode

console.log('Parent element node: ',parentElementNode)
console.log('Parent node: ', parentNode)
```

In this example, we first retrieve the child element using **getElementById**. Then we use the **parentElement** property to get the parent element of the child element, and the **parentNode** property to get the parent *node* of the child element.

children and *childNodes*

The **children** and **childNodes** properties are used to get the child nodes of an element. The **children** property returns only element nodes, while the **childNodes** property can return any node type.

Add the following to **script.js**:

```
const parentElement = document.getElementById('parent')
const children = parentElement.children
const childNodes = parentElement.childNodes

console.log('Children: ', children)
console.log('Child nodes: ', childNodes)
```

In this example, we first retrieve the parent element then we use the **children** and **childNodes** properties to get the child elements and child nodes of the parent element, respectively.

hasChildNodes

The **hasChildNodes** method is used to check if an element has any child nodes. It returns a Boolean value indicating whether the element has any child nodes.

Add the following to **script.js**:

```
const hasChildren = parentElement.hasChildNodes();
console.log('Has children: ', hasChildren);
```

DOM Traversal - Part Two

firstChild and *firstElementChild*

The **firstChild** and **firstElementChild** properties are used to get the first child node of an element. The **firstChild** property can return any node type, while the **firstElementChild** property returns only an element node.

Add the following to **script.js**:

```
const firstChild = parentElement.firstChild
const firstElementChild = parentElement.firstElementChild

console.log('First child: ', firstChild)
console.log('First element child: ', firstElementChild)
```

> ## *lastChild* and *lastElementChild*

The **lastChild** and **lastElementChild** properties are used to get the last child node of an element. The **lastChild** property can return any node type, while the **lastElementChild** property returns only an element node.

Add the following to **script.js**:

```
const lastChild = parentElement.lastChild
const lastElementChild = parentElement.lastElementChild

console.log('Last child: ', lastChild)
console.log('Last element child: ', lastElementChild)
```

> ## *nextSibling* and *nextElementSibling*

The **nextSibling** and **nextElementSibling** properties are used to get the next sibling node of an element. The **nextSibling** property can return any node type, while the **nextElementSibling** property returns only an element node.

Add the following to **script.js**:

```
const firstChildElement = document.getElementById('first-child-
        element')
const nextSibling = firstChildElement.nextSibling
const nextElementSibling = firstChildElement.nextElementSibling

console.log('Next sibling: ', nextSibling)
console.log('Next element sibling: ', nextElementSibling)
```

previousSibling and *previousElementSibling*

The **previousSibling** and **previousElementSibling** properties are used to get the previous sibling node of an element. The **previousSibling** property can return any node type, while the **previousElementSibling** property returns only an element node.

Add the following to **script.js**:

```
const previousSibling = firstChildElement.previousSibling
const previousElementSibling =
        firstChildElement.previousElementSibling

console.log('Previous sibling: ', previousSibling)
console.log('Previous element sibling: ',
        previousElementSibling)
```

Additional Resources

Additional Resources for Day 3

This page provides a list of external resources that you can use to enhance your understanding of concepts learned in Day 3.

Events

JavaScript Events

Introduction to Events
You will also find information about Event bubbling and Event delegation here

JavaScript Event Listener

Animation Events

Events Examples

Advanced DOM

DOM Navigation

createElement()

remove()

Conclusion & Takeaway

> In this lesson, we got a deeper understanding of JavaScript’s capabilities for manipulating and traversing the DOM. We delved into how elements can be dynamically created, manipulated, and removed, granting a developer the flexibility to modify webpage content on the fly. Additionally, we explored how to navigate through the structure of the DOM using various properties and methods.

Advanced DOM - Part 2

> ### Goals
>
> By the end of this lesson, you should understand:
>
> - The basics of how to use the Intersection Observer API
>
> ### Introduction
>
> In this lesson, we will explore the Intersection Observer API, a tool used to asynchronously observe changes in the intersection between a target element and an ancestor element or the document's viewport. We will go over how to create an IntersectionObserver instance, define its callback function, configure it to detect changes, and set thresholds to trigger the callback function.

Intersection Observer API

> The Intersection Observer API provides a way to asynchronously observe changes in the intersection of a target element with an ancestor element or with a top-level document's viewport. An observer can be configured to watch for changes in the visibility of one or more target elements, and to trigger a callback function whenever the intersection ratio of a target element changes.
>
> The intersection ratio is a measure of how much of the target element is currently visible within the viewport. The value of the ratio ranges from 0.0 to 1.0, where 0.0 indicates that none of the target element is visible, and 1.0 indicates that the entire target element is visible.
>
> ## Using the Intersection Observer API:
>
> To use the Intersection Observer API, you first need to create an instance of the IntersectionObserver class. The constructor for this class takes a callback function as its first argument, and an options object as its second argument. The options object can be used to configure various aspects of the observer, such as the threshold at which the callback function should be triggered, and whether the observer should only detect changes in the vertical or horizontal axis.
>
> Let's walk through an example of creating an IntersectionObserver instance:

```
const observer = new IntersectionObserver(logIntersection)
```

Here, we're passing in a callback function called "logIntersection" (which we will define in a moment) to handle what should happen when our observer is activated.

Next, now that we've created an IntersectionObserver instance, we can use its **observe** method to start watching for changes in the intersection of a target element. The observe method takes an element as an argument, and will add the element to the list of targets being watched by the observer.
>
Add the following to **script.js**:

```
const element = document.querySelector('#first-observer-target')
observer.observe(element)
```

The callback function that we passed to the IntersectionObserver constructor will be triggered whenever the intersection ratio of a target element changes. The callback function takes two arguments: an array of IntersectionObserverEntry objects, and the observer instance that triggered the callback.
>
The IntersectionObserverEntry objects contain information about the target elements, such as their bounding rectangles and the current intersection ratio. You can use this information to update the UI or trigger other actions based on changes in the visibility of the target elements.
>
Let's now define the **logIntersection** callback function:

```
function logIntersection(entries, observer) {
  entries.forEach(entry => {
    // Do something with the entry data
    const entryObject = {
      isIntersecting: entry.isIntersecting,
      intersectionRatio: entry.intersectionRatio,
      target: {
        element: entry.target.tagName,
        id: entry.target.id
      }
    }

    if (entry.isIntersecting) {
      console.log(`${entry.target.id} is intersecting`)
    } else {
      console.log(`${entry.target.id} is NOT intersecting`)
    }

    console.log('Entry: ', entryObject)
    console.log('\n')

    observer.unobserve(entry.target)
  })
}
```

In this example, the callback function uses a *forEach* loop to iterate over each entry in the array, and then performs some action based on the entry data. For the purposes of this very basic example, if the entry is intersecting we just log some information to the console; and if it is not intersecting we log that to the console as well.
>

Thresholds

> The Intersection Observer API provides a way to set one or more thresholds at which the callback function should be triggered. A threshold is a percentage value between 0.0 and 1.0 that represents the minimum intersection ratio.
>
> To set a threshold, you can include it as a property of the options object when creating an IntersectionObserver instance. You can also set multiple thresholds by including an array of values.
>
> Here is an example of how to set a threshold of 0.5:

```
const secondObserverOptions = { threshold: 0.5 }
const secondObserver = new IntersectionObserver(logIntersection,
        secondObserverOptions)
const element2 = document.querySelector('#second-observer-
        target')


secondObserver.observe(element2)
```

In this example, our options object includes a threshold property with a value of 0.5. This means that the callback function will be triggered whenever the intersection ratio of a target element is at least 50%.

Multiple thresholds can also be specified by including an array of values:

```
{ threshold: [0.25, 0.5, 0.75] }
>
```

Viewport Margins

> It is also possible to define viewport margins for an observer, which act kind of like an offset and can be useful for detecting when an element is about to enter or exit the viewport. This is accomplished by adding margin values to the **rootMargin** property of the options object.
>
> The **rootMargin** property is a string that contains four margin values separated by spaces. The margin values can be specified in pixels, percentages, or a combination of both.
>
> Here is an example of how to add a margin of 100 pixels to the bottom of the viewport:

```
const thirdObserverOptions = { rootMargin: '0px 0px 100px 0px'
        };
const thirdObserver = new IntersectionObserver(logIntersection,
        thirdObserverOptions)
const element3 = document.querySelector('#third-observer-
        target')


thirdObserver.observe(element3)
```

Here the rootMargin property defines 0 pixels for the top, right, and left margins, and 100 pixels for the bottom margin. This effectively means that our observer's callback function will be triggered when the target element is 100px from the bottom of our viewport.
>

Conclusion & Takeaway

> The Intersection Observer API is a versatile tool allowing us to observe changes in the visibility of target elements within the viewport. In this lesson, we covered the basics of how to use and configure an IntersectionObserver instance using various options that the API provides. In the next two lessons we will work on a couple of mini-projects which incorporate the concepts we've learned this week: the first will give us some practice with creating and removing DOM elements, and the second will utilize an intersection observer to add a slick-looking animation to elements as a user scrolls down the page!

Additional Resources

> This page provides a list of external resources that you can use to enhance your understanding of concepts learned in this lesson.

Intersection Observer API
>

Drag and Drop API (AEL)

> ## Goals
>
> By the end of this lesson, you should understand:
>
> - The basics of how to use the Drag and Drop API
>
> ## Introduction
>
> In this asynchronous elective learning lesson, we will explore the Drag and Drop API – another powerful browser API which can create some really cool functionality in web applications! We'll start from the basics and progress to more advanced usage, working through several examples.

Basic Drag and Drop

The basic drag and drop functionality involves three events:
1. The **dragstart** event is triggered when the user starts dragging an element.
2. The **dragover** event is triggered when the user drags an element over a drop target.
3. The **drop** event is triggered when the user drops an element onto a drop target.

Here is an example of how to implement basic drag and drop functionality using the Drag and Drop API:

```
const draggable = document.getElementById('draggable');
const dropzone = document.querySelector('.dropzone');
const dropzonePlaceholder = document.querySelector('.dropzone-placeholder');


// Set draggable data for drop zone (draggable element ID) and adding "dragging-active" class
draggable.addEventListener('dragstart', (event) => {
  event.dataTransfer.setData('text/plain', event.target.id);
});

// Allow drop (drop not allowed by default)
dropzone.addEventListener('dragover', (event) => {
  event.preventDefault();
});


// Append draggable element to dropzone
dropzone.addEventListener('drop', (event) => {
  event.preventDefault();
  const draggableId = event.dataTransfer.getData('text/plain');
  console.log(draggableId);
  const element = document.getElementById(draggableId);
  dropzone.replaceChild(element, dropzone.children[0]);
});
```

In this example, we have two elements: a draggable element and a drop zone element. We add event listeners to the draggable element for the **dragstart** event, and to the drop zone element for the **dragover** and **drop** events.

>

In the **dragstart** event listener, we use the **setData** method of the event's **dataTransfer** object to set the data that will be transferred when the element is dropped.

>

In the **dragover** event listener, we call the **preventDefault** method of the event object to allow the element to be dropped onto the drop zone element.

>

In the **drop** event listener, we call the **preventDefault** method of the event object to prevent the default behavior of the browser. We then use the **getData** method of the event's **dataTransfer** object to retrieve the data that was set in the **dragstart** event listener. Finally, we append the dragged element to the drop zone element using the **appendChild** method.

>

Customize the Drag Image

> The Drag and Drop API also provides additional functionality that can be used to create more complex drag and drop interactions. For example, the API provides the ability to customize the appearance of the drag image, to restrict the types of data that can be dropped onto a drop target, and to handle multiple draggable elements at once.
>
> Here is an example of how to customize the appearance of the drag image:

```
const draggable = document.getElementById('draggable');
const dropzone = document.querySelector('.dropzone');


// Create a custom drag image element and set its source to a
random image.
// NOTE: We're doing this outside the event listener so the
random image is available the first time the draggable element
is clicked.
const customDragImg = document.createElement('img');
customDragImg.src = "https://picsum.photos/150/100";

// Set draggable data for drop zone (draggable element ID), set
drag image and add "dragging-active" class to draggable element
draggable.addEventListener('dragstart', (event) => {
  event.dataTransfer.setData('text/plain', event.target.id);
  event.dataTransfer.setDragImage(customDragImg, 0, 0);
  draggable.classList.add('dragging-active');
});

// Remove "dragging-active" class when dragging ends
draggable.addEventListener('dragend', (event) => {
  draggable.classList.remove('dragging-active');
})

// Allow drop (drop not allowed by default)
dropzone.addEventListener('dragover', (event) => {
  event.preventDefault();
});


// Append draggable element to dropzone
dropzone.addEventListener('drop', (event) => {
  event.preventDefault();
  const draggableId = event.dataTransfer.getData('text/plain');
  const element = document.getElementById(draggableId);
  dropzone.replaceChild(element, dropzone.children[0]);
});
```

In this example, we use the **setDragImage** method to set a custom drag image and add a second argument in order to specify the offset of the image. This allows us to position the drag image relative to the mouse cursor.
>

Restrict Types of Data

Here is an example of how to restrict the types of data that can be dropped onto a drop target:

```
const draggable = document.getElementById('draggable');
const dropzone = document.querySelector('.dropzone');


// Set draggable data for drop zone with restricted data type,
and add "dragging-active" class
draggable.addEventListener('dragstart', (event) => {
  event.dataTransfer.setData('text/uri-list', event.target.id);
});


// Allow drop ONLY if type of data is 'text/plain'
dropzone.addEventListener('dragover', (event) => {
  const types = event.dataTransfer.types;
  if (types.includes('text/plain')) {
    event.preventDefault();
    // NOTE: As it is, our draggable element won't be dropped
here because it's type is 'text/uri-list'
  }
});


// Append draggable element to dropzone (won't work in this
case)
dropzone.addEventListener('drop', (event) => {
  event.preventDefault();
  const draggableId = event.dataTransfer.getData('text/plain');
  const element = document.getElementById(draggableId);
  dropzone.replaceChild(element, dropzone.children[0]);
});
```

In this example, we use the types property of the **dataTransfer** object to check if the dropped data is of the type ‘text/plain’. If it is, we allow the drop event to occur by calling the **preventDefault** method of the event object.

Multiple Draggables

Here is an example of how to handle multiple draggable elements at once:

```
const draggables = document.querySelectorAll('.draggable');
const dropzone = document.querySelector('.dropzone');


// For each draggable item, set data for drop zone (draggable
element ID) and add "dragging-active" class
for (const draggable of draggables) {
  draggable.addEventListener('dragstart', (event) => {
    event.dataTransfer.setData('text/plain', event.target.id);
    draggable.classList.add('dragging-active');
  });

  // Remove "dragging-active" class when dragging ends
  draggable.addEventListener('dragend', (event) => {
    draggable.classList.remove('dragging-active');
  })
}

// Allow drop (drop not allowed by default)
dropzone.addEventListener('dragover', (event) => {
  event.preventDefault();
});


// Append current draggable element to dropzone
dropzone.addEventListener('drop', (event) => {
  event.preventDefault();
  const draggableId = event.dataTransfer.getData('text/plain');
  const element = document.getElementById(draggableId);

  // Check if dropzone has only one child element and if it is
the placeholder element
  // If TRUE, use replaceChild() to the replace the placeholder
with a draggable element
  // If FALSE, use appendChild() to add the draggable element to
the dropzone
  if (
    dropzone.children.length === 1
    &&
    dropzone.children[0].classList.contains('dropzone-
placeholder')
  ) {
    dropzone.replaceChild(element, dropzone.children[0]);
  } else {
    dropzone.appendChild(element);
  }
});
```

In this example, we use **querySelectorAll** to select all elements with the class ‘draggable’, and add event listeners to each one for the **dragstart** event. When an element is dropped onto the drop zone element, only the dropped element is appended to the drop zone, regardless of how many elements were dragged.

Additional Resources

> This page provides a list of external resources that you can use to enhance your understanding of concepts learned in this lesson.

Drag and Drop API
>

Conclusion & Takeaways

> Through this Asynchronous Elective Learning lesson, we covered the fundamentals of the Drag and Drop API. We began with basic drag and drop functionality, exploring the crucial events such as dragstart, dragover, and drop, and then progressed to more advanced features including customizing the drag image, restricting data types, and handling multiple draggable elements. Armed with this knowledge, we have yet another tool to help us create dynamic and interactive user experiences in our web applications.

Mini Project: To Do List

> ## Goals
>
> By the end of this lesson, you will:
>
> - Get more practice with what you learned throughout this week
> - Build a simple To Do List with the ability to create and delete elements

> ## Getting Started
>
> You will see two files open in the IDE: **index.html**, and **script.js**.
>
> The HTML is already complete with Bootstrap added for styling, and we'll be focusing only on writing the JavaScript in **script.js**.

Build the functionality

> As always let's get the DOM elements we'll need, and also define some global variables:

```
// DOM Elements and Global Variables
const addTaskBtn = document.getElementById('add-task');
const input = document.querySelector('input');
const taskList = document.querySelector('#task-list');

let taskId = 0;
let randomImgId = 1;
```

Now, we need to create a function to handle adding a new task. This function will take in one parameter (the value that a user enters in the input element), will need to create a new element, add some Bootstrap classes to it, and also set the innerHTML of our newly created item in the format that we want. Then, it needs to append our new ask item to the task list:

```
// Add a new task to the list
const addTask = (task) => {
  const taskItem = document.createElement('div');
  taskItem.classList.add('form-check', 'd-flex', 'align-items-
      center', 'gap-3');
  taskItem.innerHTML = `
    <input class="task-check" type="checkbox" id="task-
      ${taskId}">
    <label class="task-check-label" for="task-${taskId}">${task}
      </label>
    <button type="button" class="btn-close" aria-label="Close"
      data-task-id="${taskId}"></button>
  `;
  taskList.appendChild(taskItem);
  taskId++;
}
```

Now let's create another function to remove a task from the list:

```
// Remove a task from the list
const removeTask = (taskId) => {
  const taskItem = document.querySelector(`#task-
        ${taskId}`).parentNode;
  taskList.removeChild(taskItem);
}
```

Now let's add a click listener to our **addTaskButton**, which will handle getting the value from our input element, modifying it, and passing it to the **addTask** function.
>
Just for fun, we'll also add a randomly selected image to include with our task description. Additionally, we should be sure to remove any leading or trailing white space from the user input using the **trim()** string method, and should check to make sure that the input element isn't empty before we add anything to our task list. Then, we need to increment the **randomImgId** so that we can keep getting a new random image from Picsum:

```
// Handle Add Task button click
addTaskBtn.addEventListener('click', () => {
  const task = input.value.trim();
  const taskWithImg = `<img class="me-3"
        src="https://picsum.photos/50/50?random=${randomImgId}">
        <span>${task}</span>`;



  if (task !== '') {
    addTask(taskWithImg);
    input.value = '';
    randomImgId++;
  }
});
```

Lastly, we need to create another click listener (we'll use event delegation here) to handle when the user clicks the button to delete the task, and when the user clicks on a task item or checks the checkbox. If the checkbox for our task item is checked, it should create a strike-through effect on our text, and if the delete button is clicked the task item should get removed from the DOM:

```
// Handle completing a task and removing it from the list
taskList.addEventListener('click', (e) => {
  const target = e.target;
  if (target.matches('.task-check')) {
    const label = target.parentNode.querySelector('label');
    if (target.checked) {
      label.classList.add('text-decoration-line-through');
    } else {
      label.classList.remove('text-decoration-line-through');
    }
  } else if (target.matches('.btn-close')) {
    const taskId = target.getAttribute('data-task-id');
    removeTask(taskId);
  }
});
```

And that’s that! Another mini project under our belts!

Conclusion & Takeaway

> In this lesson, we further reinforced our understanding of working with the DOM by building a simple To Do list, which incoporated some of the advanced DOM manipulation we learned and honed our practical skills for managing user interaction in a web application.

Mini-Project: Fade and Slide on Scroll

Goals

By the end of this lesson, you will:
- Have built out a common UI animation using Intersection Observer
- Have another tool you can use to add flair and dynamism to your personal website

Getting Started

This lesson contains a brief walkthrough of adding a fade-and-slide-in on scroll effect to DOM elements, so that as a user scrolls down the page elements will both fade and slide into view.

You will see three files open in Codio: **index.html**, **style.css**, and **script.js**.
>
As per usual the HTML and CSS files are already complete, so have a look at them briefly if you want but we'll only be writing the JavaScript.
>

Build the functionality

> For this project, we will use what we learned about the Intersection Observer API, and work through a more practical use case.

As always, let's begin by grabbing our DOM elements:

```
const items = document.querySelectorAll('.item:not(:first-
        child)');
```

Next, we'll define the options to use with our IntersectionObserver instance. In this case, we'll just set a threshold of **0.5** so that our callback function will be triggered when the observed elements are 50% within the viewport:

```
const options = {
  threshold: 0.5
}
```

Next, let's create the actual observer instance as well as the callback function, which will need to loop over the any entries it's observing and add the **slide-in** class to each one if it is intersecting. We will also pass in options object we just defined:

```
function addSlideIn(entries) {
  entries.forEach(entry => {
    if (entry.isIntersecting) {
      entry.target.classList.add('slide-in');
    }
  });
}

const observer = new IntersectionObserver(addSlideIn, options)
```

Lastly, we need to loop over all the DOM nodes we selected with **querySelectorAll** and tell our observer to observe each of them:

```
items.forEach(item => {
  observer.observe(item);
})
```

```
And we’re done!
>
```

Conclusion & Takeaway

> In this lesson, we created a basic web page which utilized the Intersection Observer API to give the elements a *fade-and-slide-in* effect as the page is scrolled. Next, to close out this module, we will go over the instructions for Project 10.

Coming up in Module 11, we'll be learning even *more* functionality we can achieve with JavaScript!
>

Project: Add to Personal Website

Instructions

For project 10, you will need to complete the following two steps:

Step One

As with project 9, implement some of what you learned in this module in your personal website.

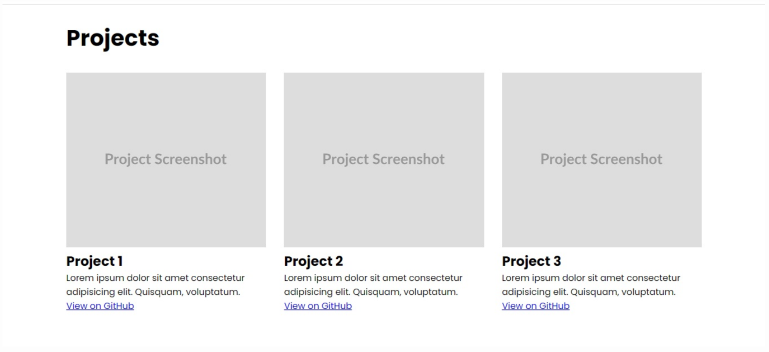
One possibility would be to apply the *fade-and-slide-in* effect to the content in your website's different sections. You can use the code you just wrote in the mini-project as a guide. Another possibility would be to implement some kind of calculator, animation event, drag and drop (if you went through the AEL lesson covering that), a list creator, etc.

You can decide which of these suggestions you want to incorporate, but **pick at least one**.

Step Two

> **NOTE**: If you didn't create the project section on your website last week, as it wasn't a hard requirement, go ahead and do that first!

Add one or more of the mini-projects you completed this week (or something you built outside the course) to GitHub and to your personal website's project section. As a reminder, each project on your personal website should include the following elements: an **image** (screenshot), a **title**, a **brief description**, and a **link to the GitHub repo**. Here's the same example image from project 9 for reference:



Projects

Project Screenshot | Project Screenshot | Project Screenshot

Project 1
Lorem ipsum dolor sit amet consectetur adipisicing elit. Quisquam, voluptatum.
View on GitHub

Project 2
Lorem ipsum dolor sit amet consectetur adipisicing elit. Quisquam, voluptatum.
View on GitHub

Project 3
Lorem ipsum dolor sit amet consectetur adipisicing elit. Quisquam, voluptatum.
View on GitHub

Projects section example

When you're finished with both steps, submit your HTML, CSS, and JavaScript below.

Happy coding!

>