Dev Tools/JavaScript notes

Open with right click (RC) or pressing F12

* window.innerHeight and window.innerWidth - The height and width of the browser not including any menus/scrollbars.
* window.location.href - The current URL in the address bar.
* window.history - An object representing the history of this tab, providing methods like history.back(), the equivalent of pressing the back button.
* window.alert() - The alert method you just used.
* window.prompt() and window.confirm() - Similar to window.alert(), but for prompting for user input and getting user confirmation, respectively.

Diagram, venn diagram

Description automatically generated

The first and most common way you will declare variables is using the let keyword. This keyword creates a variable which has **block scope**. This means that the variable will **only** be available for use within the code block in which it is declared.

if (false) {

let myVar = "hello!";

console.log(myVar);

} else {

console.log("Inside the else block:", myVar);

}

Now, because the condition is not true the if statement will attempt to execute the else block and log "Inside the else block: hello!" to the console. However, **because myVar is declared with let and thus has block scope, it is restricted to the if block, that is, it cannot exist outside the closing curly brace (}) just before the else**. Running this code will produce a ReferenceError, because in the context of the else block, myVar is not defined.

Diagram, venn diagram

Description automatically generated

Variables declared with const are similar to those declared with let in some ways. They have block scope like let variables and will behave identically in that respect

Unlike let variables, constants **cannot** be reassigned *or* redeclared

Unlike let variables, constants cannot be declared without being assigned a value

Arrays and objects stored in constants can be modified, but you can't reassign a new object or array to the same constant

Diagram, venn diagram

Description automatically generated

Variables declared with var behave significantly different than their const and let counterparts. The most significant difference is that variables declared with var are **accessible outside the scope in which they are declared**. This means that they can be inadvertently overwritten and can create bugs in your code that are difficult to detect.

Diagram

Description automatically generated

Graphical user interface, text, application

Description automatically generated

| **Character** | **Escape Sequence** |
| --- | --- |
| Horizontal Tab | \t |
| Newline | \n |
| Carriage Return | \r |
| Single Quote | \' |
| Double Quote | \" |
| Backslash | \\ |

Graphical user interface, text, application

Description automatically generated

A picture containing text

Description automatically generated

| **Method/Property** | **Description** |
| --- | --- |
| length | Returns the length of a string |
| charAt() | Returns the character at the specified index (position) |
| charCodeAt() | Returns the Unicode of the character at the specified index |
| concat() | Joins two or more strings, and returns a new joined strings |
| endsWith() | Checks whether a string ends with specified string/characters |
| fromCharCode() | Converts Unicode values to characters |
| includes() | Checks whether a string contains the specified string/characters |
| indexOf() | Returns the position of the first found occurrence of a specified value in a string |
| lastIndexOf() | Returns the position of the last found occurrence of a specified value in a string |
| localeCompare() | Compares two strings in the current locale |
| match() | Searches a string for a match against a regular expression, and returns the matches |
| repeat() | Returns a new string with a specified number of copies of an existing string |
| replace() | Searches a string for a specified value, or a regular expression, and returns a new string where the specified values are replaced |
| search() | Searches a string for a specified value, or regular expression, and returns the position of the match |
| slice() | Extracts a part of a string and returns a new string |
| split() | Splits a string into an array of substrings |
| startsWith() | Checks whether a string begins with specified characters |
| substr() | Extracts the characters from a string, beginning at a specified start position, and through the specified number of character |
| substring() | Extracts the characters from a string, between two specified indices |
| toLocaleLowerCase() | Converts a string to lowercase letters, according to the host's locale |
| toLocaleUpperCase() | Converts a string to uppercase letters, according to the host's locale |
| toLowerCase() | Converts a string to lowercase letters |
| toString() | Returns the value of a String object |
| toUpperCase() | Converts a string to uppercase letters |
| trim() | Removes whitespace from both ends of a string |
| valueOf() | Returns the primitive value of a String object |

A picture containing table

Description automatically generated

Chart

Description automatically generated

| **Method/Property** | **Purpose** | **Usage** | **Result** |
| --- | --- | --- | --- |
| Number.isNaN() | Returns whether the passed value is Not a Number | isNaN("Hello!"); | true |
| Number.isFinite() | Returns whether the passed value is finite | isFinite(Infinity); | false |
| Number.isInteger() | Returns whether the passed value is an integer | isInteger(123); | true |
| Number.parseFloat() | Attempts to convert the passed value to a float | parseFloat("123.45"); | 123.45 |
| Number.parseInt() | Attempts to convert the passed value to an integer | parseInt("123.45"); | 123 |
| **The following are instance methods which operate on a Number instance** | | | |
| toFixed() | Returns a string representing the number with the passed number of decimal places | 123.45.toFixed(4); | "123.4500" |
| toPrecision() | Returns a string representing the number with the passed precision | 123.45.toPrecision(4); | "123.5" |
| toString() | Returns a string representing the number in the specified base (10 by default) | 123.45.toString(); | "123.45" |

Graphical user interface

Description automatically generated

A picture containing diagram

Description automatically generated

A picture containing logo

Description automatically generated

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Same As** |
| = | Assignment | let x = 5; | x = 5 |
| += | Addition Assignment | x += 3 | x = x + 3 |
| -= | Subtraction Assignment | x -= 3 | x = x - 3 |
| \*= | Multiplication Assignment | x \*= 3 | x = x \* 3 |
| /= | Division Assignment | x /= 3 | x = x / 3 |
| %= | Remainder Assignment | x %= 3 | x = x % 3 |
| \*\*= | Exponentiation Assignment | x \*\*= 3 | x = x \*\* 3 |

A picture containing logo

Description automatically generated

**Postfixing:**

let x = 3;

y = x++;

// y = 3

// x = 4

**Prefixing:**

let x = 3;

y = ++x;

// y = 4

// x = 4

Logo

Description automatically generated with low confidence

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **JavaScript Example** | **Returns** |
| == | Equal to (loose equality) | console.log(1 == 1) | true |
| === | Equal to (strict equality) | console.log(1 === "1") | false |
| != | Not equal to (loose equality) | console.log(1 != 1) | false |
| !== | Not equal to (strict equality) | console.log(1 !== "1") | true |
| > | Greater than | console.log(2 > 1) | true |
| < | Less than | console.log(2 < 1) | false |
| >= | Greater than or equal to | console.log(2 >= 1) | true |
| <= | Less than or equal to | console.log(2 <= 1) | false |

A picture containing text

Description automatically generated

**The && Operator**:

| **a** | **b** | **a && b** |
| --- | --- | --- |
| true | true | **true** |
| true | false | **false** |
| false | true | **false** |
| false | false | **false** |

**The || Operator**:

| **a** | **b** | **a || b** |
| --- | --- | --- |
| true | true | **true** |
| true | false | **true** |
| false | true | **true** |
| false | false | **false** |

**The ! Operator**:

| **a** | **!a** |
| --- | --- |
| true | false |
| false | true |

Graphical user interface

Description automatically generated

1. == (**Loose Equality**): Compares values after coercing them to compatible types
2. === (**Strict Equality**): Compares values without type coercion, comparing the data type as well

The simplest way to understand the difference is to look at an example comparing two numbers where one number is a string:

let a = 1;

let b = "1";

a == b // true

a === b // false

Graphical user interface, application

Description automatically generated

A picture containing diagram

Description automatically generated

**Standard conditional statement:**

if (condition) {

// code if true

} else {

// code if false

}

**Equivalent ternary expression:**

condition ? code if true : code if false

Ternary expressions are best used for simple conditionals. Because the expression is all on one line, they are not really good for situations where you want to execute multiple lines of code based on one of the conditions. While it's certainly possible to do this with the strategic use of parentheses and semicolons, it is not recommended because the whole point of the ternary is to make your code more concise and easier to read. The most common use for a ternary expression is in assigning a value to a variable where the value you want to assign is dependent on a condition:

let myConditionalVariable = someCondition ? trueValue : falseValue

As a more practical example, consider that a developer wants to assign a price to a user's plan based on their membership level, either "basic" or "pro". To create the price variable they could use a ternary conditional expression:

let memberType = 'basic';

let price = memberType === 'basic' ? 5 : 10;

In the example, the condition that is evaluated is whether memberType === 'basic'. If this condition is true, then price will be 5, and otherwise it will be 10. The equivalent long-hand conditional expression would be:

let memberType = 'basic';

let price;

if (memberType === 'basic') {

price = 5;

} else {

price = 10;

}

These expressions can be chained together to test multiple conditions as well, demonstrated here in a ternary conditional that adds a couple more plans to the above logic. This is the ternary version of a conditional statement that tests multiple conditions, which you'll learn about in the next unit:

let memberType = 'elite';

let price = memberType === 'basic' ? 5

: memberType === 'pro' ? 10

: membertype === 'elite' ? 20

: 0;

This is much less verbose than a whole series of if statements. Ternary conditional expressions are great when used correctly but can actully make your code less readable if overused, so while it might be tempting to try to get your entire application written on one line, you should always keep readability in mind.

Diagram

Description automatically generated

let memberType = 'elite';

let price;

if (memberType === 'basic') {

price = 5;

} else if (memberType === 'pro') {

price = 10;

} else if (memberType === 'elite') {

price = 20;

} else {

price = 0;

}

console.log(price); // 20

Diagram

Description automatically generated

There's one more pattern you can use to determine the execution of your code based on various conditions: the switch case statement. These statements are similar to if/else statements except that they give you the ability to handle many conditions at once without excessive use of else if clauses that can make your code messy. Switch statements are slightly different from standard conditionals in that they're usually used with an expression rather than a condition. This means that while an if statement is used to test whether a condition is true or false, a switch case is usually used to check the value of a variable and respond accordingly. The syntax for a switch statement is as follows:

switch (expression) {

case 'value1':

// code block 1

break;

case 'value2':

// code block 2

break;

case 'valuen':

// code block n

break;

default:

// default result

}

Like conditional statements, you can string as many cases together as you want. A switch case evaluates the expression one time and then compares its result to all the cases. The code executed will be the first case that matches. A good example might be using a switch case to determine which day of the week it is based on a number, 0 through 6. Assuming "Sunday" is day 0, your switch statement might look like this:

let day;

let dayNumber = 2;

switch (dayNumber) {

case 0:

day = 'Sunday';

break;

case 1:

day = 'Monday';

break;

case 2:

day = 'Tuesday';

break;

case 3:

day = 'Wednesday';

break;

case 4:

day = 'Thursday';

break;

case 5:

day = 'Friday';

break;

case 6:

day = 'Saturday';

break;

default:

day = 'Invalid day number';

}

console.log(day); // Tuesday

First day is declared but assigned no value, then dayNumber is set to 2. Case 2 in the switch case sets day to 'Tuesday' which is then logged to the console. In this situation the variable dayNumber is explicitly set, but most of the time you will see a switch case used to return the result (e.g. the day of the week in this example) from a function based on the value passed into it. For this example, that would look like this:

function findDayOfWeek(dayNumber) {

switch (dayNumber) {

case 0:

return 'Sunday';

case 1:

return 'Monday';

case 2:

return 'Tuesday';

case 3:

return 'Wednesday';

case 4:

return 'Thursday';

case 5:

return 'Friday';

case 6:

return 'Saturday';

default:

return 'Invalid day number';

}

}

let day = findDayOfWeek(2);

console.log(day); // Tuesday

It's important to remember the break statement in each case if the case doesn't return something, because without it the switch case will keep going through the rest of the cases. It's also important to remember that switch cases use strict equality when checking the cases, so the expression result must be both the same value and the same type for the case to be triggered. If you don't define a default case and no other case matches, the code will continue along outside the switch statement. as if it wasn't even there.

Switch case statements are nice when you need to test a single expression for many different possible results, but they are not as commonly used as the other conditional tools you've learned in this module. Nevertheless, they are a great tool to have in your arsenal.

Diagram

Description automatically generated

let fruits = ['apples', 'oranges', 'bananas', 'cherries'];

let numberOfFruits = fruits.length;

for (let i = 0; i < numberOfFruits; i++) {

console.log(fruits[i] + ' are delicious!');

}

console.log('I love fruit!');

Diagram

Description automatically generated

let fruits = ['apples', 'oranges', 'bananas', 'cherries'];

let numberOfFruits = fruits.length;

let i = 0;

while (i < numberOfFruits) {

console.log(fruits[i] + ' are delicious!');

i++;

}

console.log('I love fruit!');

let i = 10;

do {

console.log('checking i...');

console.log('i is', i);

} while (i < 10);

console.log('Loop complete');

Diagram

Description automatically generated

Iteration and looping aren't much use if you can't control them. You learned in the comparative programming module that you can use the break and continue statements within for loops and while loops to control them by skipping iterations or breaking out of the loop completely. Let's review that here and look at how to do it specifically in JavaScript. Each statement has a distinct purpose:

* **break**: breaks out of the loop entirely.
* **continue**: continues with the next iteration of the loop, effectively "skipping" an iteration.

These statements are almost always coupled with a conditional check which gives the developer the ability to break the loop or continue with the next iteration only when a specific condition is met. Here's an example of a while loop that will log every number from 0 to 1 million ... *or will it...?*

let i = 0;

while (i <= 1000000) {

if (i === 5) {

console.log('Breaking!');

break;

}

console.log(i);

i++;

}

console.log('Loop has been broken.');

[Run this code](http://pythontutor.com/visualize.html#code=let%20i%20%3D%200%3B%0Awhile%28i%20%3C%3D%201000000%29%20%7B%0A%20%20if%28i%20%3D%3D%3D%205%29%20%7B%0A%20%20%20%20console.log%28'Breaking!'%29%3B%0A%20%20%20%20break%3B%0A%20%20%7D%0A%20%20console.log%28i%29%3B%0A%20%20i%2B%2B%3B%0A%7D%0Aconsole.log%28'Loop%20has%20been%20broken.'%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

In the above example, even though the loop is set to continue until 1,000,000, the if statement within the loop checks on each iteration to see if i is 5 and if it is, the loop will be broken by the break statement. Otherwise, i is logged to the console and then incremented and the loop repeats normally. The same thing works for a for loop:

for (let i = 0; i <= 1000000; i++) {

if (i === 5) {

console.log('Breaking!');

break;

}

console.log(i);

}

console.log('Loop has been broken.');

[Run this code](http://pythontutor.com/visualize.html#code=for%28let%20i%20%3D%200%3B%20i%20%3C%3D%201000000%3B%20i%2B%2B%29%20%7B%0A%20%20if%28i%20%3D%3D%3D%205%29%20%7B%0A%20%20%20%20console.log%28'Breaking!'%29%3B%0A%20%20%20%20break%3B%0A%20%20%7D%0A%20%20console.log%28i%29%3B%0A%7D%0Aconsole.log%28'Loop%20has%20been%20broken.'%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

Alternatively, you can use the continue statement to skip an iteration of the loop under a certain condition. For example, consider a situation where you only want to log even numbers between 0 and 10:

for (let i = 0; i <= 10; i++) {

if (i % 2 !== 0) {

continue;

}

console.log(i);

}

console.log('Loop complete.');

[Run this code](http://pythontutor.com/visualize.html#code=for%28let%20i%20%3D%200%3B%20i%20%3C%3D%2010%3B%20i%2B%2B%29%20%7B%0A%20%20if%28i%20%25%202%20!%3D%3D%200%29%20%7B%0A%20%20%20%20continue%3B%0A%20%20%7D%0A%20%20console.log%28i%29%3B%0A%7D%0Aconsole.log%28'Loop%20complete.'%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

The above code is similar to the break statement except that the continue statement doesn't break out of the loop entirely. Instead, it **continues** to the next iteration each time the condition is met. The condition in this case uses the modulo operator to check whether or not the number is **not** even (remember that any number which has a remainder of 0 when divided by 2 is even). If the number is not even, the iteration is skipped.

**Labelling:**

In the event you're working with a nested loop, you can label your loops in order to continue or break a specific label:

let i = 0;

iLoop:

while (i <= 1000000) {

let j = 0;

jLoop:

while (j <= 100) {

if (j === 3) {

console.log('Breaking the outer loop from the inner loop.');

break iLoop;

}

console.log('j is', j);

j++;

}

if (i === 5) {

console.log('Breaking!');

break;

}

console.log('i is', i);

i++;

}

console.log('Loop has been broken.');

[Run this code](http://pythontutor.com/visualize.html#code=let%20i%20%3D%200%3B%0AiLoop%3A%0A%20%20while%28i%20%3C%3D%201000000%29%20%7B%0A%20%20%20%20let%20j%20%3D%200%3B%0A%20%20%20%20jLoop%3A%0A%20%20%20%20%20%20while%28j%20%3C%3D%20100%29%20%7B%0A%20%20%20%20%20%20%20%20if%28j%20%3D%3D%3D%203%29%20%7B%0A%20%20%20%20%20%20%20%20%20%20console.log%28'Breaking%20the%20outer%20loop%20from%20the%20inner%20loop.'%29%3B%0A%20%20%20%20%20%20%20%20%20%20break%20iLoop%3B%0A%20%20%20%20%20%20%20%20%7D%0A%20%20%20%20%20%20%20%20console.log%28'j%20is',%20j%29%3B%0A%20%20%20%20%20%20%20%20j%2B%2B%3B%0A%20%20%20%20%20%20%7D%0A%0A%20%20%20%20if%28i%20%3D%3D%3D%205%29%20%7B%0A%20%20%20%20%20%20console.log%28'Breaking!'%29%3B%0A%20%20%20%20%20%20break%3B%0A%20%20%20%20%7D%0A%0A%20%20%20%20console.log%28'i%20is',%20i%29%3B%0A%20%20%20%20i%2B%2B%3B%0A%20%20%7D%0Aconsole.log%28'Loop%20has%20been%20broken.'%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

This example is a little more complex, but we can break it down: The outer loop is set to iterate from 0 to 1 million. If it encounters 5, the if statement within the outer loop will break it. However, before it hits this if statement, there is another while loop set to iterate a variable j from 0 to 100. Within that loop is an if statement that checks each iteration to see if j === 3. If that condition is true it will break the **outer** loop using the label iLoop!

When the outer loop is originally created it is given a label of iLoop. If we had simply used break; inside the inner loop rather than break iLoop;, it would have broken the inner loop (the one labelled jLoop) and continued with the outer one. The if statement checking whether i === 5 would still have broken the outer loop at that point, but using a label allowed us to break it from inside another loop. This technique is not widely used, but if you find yourself in a situation where you need to break out of a specific loop, remember that you can label your loops and attach that label to a break or continue statement later on.

Timeline

Description automatically generated

let cars = ['saab', 'volvo', 'ford'];

console.log(cars[0]); // 'saab'

console.log(cars[1]); // 'volvo'

console.log(cars[2]); // 'ford'

Table

Description automatically generated

Along with their properties like length, arrays have a number of built-in methods for performing various operations on them. Some of the most common operations you might perform on arrays using these methods include:

* Popping and pushing elements
* Shifting and unshifting elements
* Deleting and splicing elements
* Changing the values of specific elements
* Slicing, sorting and merging
* Testing whether an array includes an element
* Mapping and reducing

These terms won't all make sense to you right now and there are other methods besides the ones listed here, but once you know a few of them you'll find it easy to understand the rest.

Let's go through each of the list items above. For all of these examples we'll use the same array so you can easily see the differences and the functionality of each method:

let fruits = ['apples', 'pears', 'cherries', 'bananas', 'peaches', 'oranges'];

**Popping and pushing elements:**

You can pop an item off the end of an array, or push one or more onto the end of it using the pop() and push() methods respectively. The pop() method returns the item you popped off and the push() method returns the new length of the array, if you assign them to a variable:

let fruits = ['apples', 'pears', 'cherries', 'bananas', 'peaches', 'oranges'];

let lastItem = fruits.pop();

console.log(lastItem);

console.log(fruits);

let newLength = fruits.push('mangoes');

console.log(newLength);

console.log(fruits);

[Run this code](http://pythontutor.com/visualize.html#code=let%20fruits%20%3D%20%5B'apples',%20'pears',%20'cherries',%20'bananas',%20'peaches',%20'oranges'%5D%3B%0Alet%20lastItem%20%3D%20fruits.pop%28%29%3B%0A%0Aconsole.log%28lastItem%29%3B%0Aconsole.log%28fruits%29%3B%0A%0Alet%20newLength%20%3D%20fruits.push%28'mangoes'%29%3B%0Aconsole.log%28newLength%29%3B%0Aconsole.log%28fruits%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

**Shifting and unshifting elements:**

While the pop() method pops an item off the *end* of the array, you can pop an item off the *front* of the array using the shift() method. It returns the item you "shifted". Likewise, you can add one or more items to the beginning of the array using the unshift() method:

let fruits = ['apples', 'pears', 'cherries', 'bananas', 'peaches', 'oranges'];

let firstItem = fruits.shift();

console.log(firstItem);

console.log(fruits);

let newLength = fruits.unshift('mangoes');

console.log(newLength);

console.log(fruits);

[Run this code](http://pythontutor.com/visualize.html#code=let%20fruits%20%3D%20%5B'apples',%20'pears',%20'cherries',%20'bananas',%20'peaches',%20'oranges'%5D%3B%0Alet%20firstItem%20%3D%20fruits.shift%28%29%3B%0A%0Aconsole.log%28firstItem%29%3B%0Aconsole.log%28fruits%29%3B%0A%0Alet%20newLength%20%3D%20fruits.unshift%28'mangoes'%29%3B%0Aconsole.log%28newLength%29%3B%0Aconsole.log%28fruits%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

**Deleting and splicing elements:**

You might think that you can delete an item by passing its index to the delete keyword:

let fruits = ['apples', 'pears', 'cherries', 'bananas', 'peaches', 'oranges'];

delete fruits[2];

console.log(fruits);

[Run this code](http://pythontutor.com/visualize.html#code=let%20fruits%20%3D%20%5B'apples',%20'pears',%20'cherries',%20'bananas',%20'peaches',%20'oranges'%5D%3B%0Adelete%20fruits%5B2%5D%3B%0Aconsole.log%28fruits%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

If you run the code though, you'll see it doesn't actually delete the item, it replaces it with undefined. The correct way to truly delete a specific item in an array is to use the splice() method. To use splice, you specify the index you would like to begin at and the number of items you want to remove:

let fruits = ['apples', 'pears', 'cherries', 'bananas', 'peaches', 'oranges'];

fruits.splice(2, 1); // Remove one item starting at index 2

console.log(fruits);

[Run this code](http://pythontutor.com/visualize.html#code=let%20fruits%20%3D%20%5B'apples',%20'pears',%20'cherries',%20'bananas',%20'peaches',%20'oranges'%5D%3B%0Afruits.splice%282,%201%29%3B%20%20//%20Remove%20one%20item%20starting%20at%20index%202%0Aconsole.log%28fruits%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

**Changing the values of specific elements:**

You can also use splice() to insert elements into the array at the starting index, by specifying the elements you want to insert in a comma-separated list after the 2nd argument:

let fruits = ['apples', 'pears', 'cherries', 'bananas', 'peaches', 'oranges'];

// Remove 'cherries', insert 'pineapples' and 'mangoes'

// 'bananas' is shifted to the right

fruits.splice(2, 1, 'pineapples', 'mangoes');

console.log(fruits);

[Run this code](http://pythontutor.com/visualize.html#code=let%20fruits%20%3D%20%5B'apples',%20'pears',%20'cherries',%20'bananas',%20'peaches',%20'oranges'%5D%3B%0A%0A//%20Remove%20'cherries',%20insert%20'pineapples'%20and%20'mangoes'%0A//%20'bananas'%20is%20shifted%20to%20the%20right%0Afruits.splice%282,%201,%20'pineapples',%20'mangoes'%29%3B%0Aconsole.log%28fruits%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

The above removes one element, starting at index 2 and then inserts two new elements starting at that position. The splice() element returns an array of the inserted or deleted elements. If you don't want to replace any items, but just want to splice some items into the array starting at a specific point, you can specify 0 for the second argument:

let fruits = ['apples', 'pears', 'cherries', 'bananas', 'peaches', 'oranges'];

// Insert 'pineapples' and 'mangoes' starting at index 2

// 'cherries' is shifted to the right

fruits.splice(2, 0, 'pineapples', 'mangoes');

console.log(fruits);

[Run this code](http://pythontutor.com/visualize.html#code=let%20fruits%20%3D%20%5B'apples',%20'pears',%20'cherries',%20'bananas',%20'peaches',%20'oranges'%5D%3B%0A%0A//%20Insert%20'pineapples'%20and%20'mangoes'%20starting%20at%20index%202%0A//%20'cherries'%20is%20shifted%20to%20the%20right%0Afruits.splice%282,%200,%20'pineapples',%20'mangoes'%29%3B%0Aconsole.log%28fruits%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

If you simply want to change the value of an existing element, you can do it directly by replacing its index with the new value:

let fruits = ['apples', 'pears', 'cherries', 'bananas', 'peaches', 'oranges'];

fruits[2] = 'pineapples';

console.log(fruits);

[Run this code](http://pythontutor.com/visualize.html#code=let%20fruits%20%3D%20%5B'apples',%20'pears',%20'cherries',%20'bananas',%20'peaches',%20'oranges'%5D%3B%0Afruits%5B2%5D%20%3D%20'pineapples'%3B%0Aconsole.log%28fruits%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

**Slicing, sorting and merging:**

You can also slice an array to obtain a subset of it by using the slice() method. The following slices the array and returns only indices 2 through 4 (not including 4). This does not modify the original array:

let fruits = ['apples', 'pears', 'cherries', 'bananas', 'peaches', 'oranges'];

let subset = fruits.slice(2, 4);

console.log(fruits);

console.log(subset);

[Run this code](http://pythontutor.com/visualize.html#code=let%20fruits%20%3D%20%5B'apples',%20'pears',%20'cherries',%20'bananas',%20'peaches',%20'oranges'%5D%3B%0Alet%20subset%20%3D%20fruits.slice%282,%204%29%3B%0Aconsole.log%28fruits%29%3B%0Aconsole.log%28subset%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

The sort() method will sort the array, by default, from lowest to highest or in alphabetical order. The sorting methodology will differ based on the contents of the array, but usually you will use this to sort numerically or alphabetically:

let fruits = ['apples', 'pears', 'cherries', 'bananas', 'peaches', 'oranges'];

fruits.sort();

console.log(fruits);

[Run this code](http://pythontutor.com/visualize.html#code=let%20fruits%20%3D%20%5B'apples',%20'pears',%20'cherries',%20'bananas',%20'peaches',%20'oranges'%5D%3B%0Afruits.sort%28%29%3B%0Aconsole.log%28fruits%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

You can merge two arrays together using the concat()method (short for **concatenate** which you might remember from the string lessons means "to link things together in a chain or a series"). Just pass one or more arrays into it to merge them all together:

let fruits = ['apples', 'pears', 'cherries', 'bananas', 'peaches', 'oranges'];

let vegetables = ['carrots', 'peas', 'beans', 'lettuce'];

let healthyFoods = fruits.concat(vegetables);

console.log(fruits);

console.log(vegetables);

console.log(healthyFoods);

[Run this code](http://pythontutor.com/visualize.html#code=let%20fruits%20%3D%20%5B'apples',%20'pears',%20'cherries',%20'bananas',%20'peaches',%20'oranges'%5D%3B%0Alet%20vegetables%20%3D%20%5B'carrots',%20'peas',%20'beans',%20'lettuce'%5D%3B%0Alet%20healthyFoods%20%3D%20fruits.concat%28vegetables%29%3B%0A%0Aconsole.log%28fruits%29%3B%0Aconsole.log%28vegetables%29%3B%0Aconsole.log%28healthyFoods%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

**Testing whether an array includes an element:**

If you need to know whether a specific element exists in an array, you can test it using the includes() method, which will return true if the element exists in the array, and false otherwise:

let fruits = ['apples', 'pears', 'cherries', 'bananas', 'peaches', 'oranges'];

let hasApples = fruits.includes('apples'); // true

let hasMangoes = fruits.includes('mangoes'); // false

console.log(hasApples);

console.log(hasMangoes);

[Run this code](http://pythontutor.com/javascript.html#code=let%20fruits%20%3D%20%5B'apples',%20'pears',%20'cherries',%20'bananas',%20'peaches',%20'oranges'%5D%3B%0Alet%20hasApples%20%3D%20fruits.includes%28'apples'%29%3B%20%20//%20true%0Alet%20hasMangoes%20%3D%20fruits.includes%28'mangoes'%29%3B%20%20//%20false%0Aconsole.log%28hasApples%29%3B%0Aconsole.log%28hasMangoes%29%3B&curInstr=0&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D)

**Mapping, filtering and reducing:**

There are three other methods that are commonly used with arrays, but you'll need to learn more about functions and the rest of JavaScript before you can understand how they work. In the meantime, cache these away in your memory bank; they're quite handy and a great thing to Google when you inevitably need to manipulate arrays in more complex ways!

1. .map(): executes a given function on every single element of the array. An example might be capitalizing all the elements of an array, or multipling them all by some number.
2. .filter(): filters the array down to only elements that meet specific criteria. An example might be filtering a list of names down to only names that begin with a certain letter.
3. .reduce(): reduces all the array elements down to a single result based on a given formula. An example might be reducing an array of numbers down to their sum, by addng them all together.

As you can see there are many methods and ways to work with arrays. To learn more about these methods you can explore the [MDN Website](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#Instance_methods). You don't need to memorize every single one. Instead try to focus on remembering all the possible things you might want to do with an array, like adding/removing elements, sorting elements, combining arrays together, plucking certain elements out of them and so on. By focusing on the possibilities you're more likely to know what to search for when you need to accomplish a task. Googling "How to sort a JavaScript array" will most definitely lead you to the sort() method, and that can point you to the documentation which can refresh your memory on how to actually use it, so knowing what you want to accomplish is the most important part!

Diagram

Description automatically generated

To define an object in JavaScript, you simply wrap a set of comma-separated property:value pairs in { curly braces }:

let car = {

color: 'white',

numberOfSeats: 5,

price: 20000

}

In the above object, the **properties** are color, numberOfSeats and price, and the **values** are 'white', 5, and 20000 respectively. This syntax can be standardized to:

let object = {

property1: value1,

property2: value2,

property3: value1,

...

propertyN: valueN

}

In the previous lessons on objects, you learned about object keys and values, and how you can create either an empty object and then create the keys and values.

let person = {}

person.name = "john";

console.log(person.name) → john

Or you can create the same object as follows

let person = {name: "john"};

console.log(person.name) → john

Computed property names gives us another way to do this, so let"s look at an example.

let prop = "name";

let person = {[prop]: "john"};

So we have created a variable and we want the value assigned to it to be the name of a key in the person object and to do this we have added the variable name wrapped in [] in the object where we would normally place the key.

So now if you log out person.name you would get john even though looking at the object there is no key name.

console.log(person.name) → john

But you can also access that value of that key like

console.log(person[prop]) → john

So we can see from doing: let prop = "name" using [prop] in the object is the same as doing {name: "John"}

So that was using a value as a computed property, what is using an expression as a computed property?

Let"s adjust our example.

let prop = "name";

let person = {

[prop]: "john",

lastName: "smith",

["full" + prop]: "john smith"

};

Ok so we have added ["full" + prop]: "john smith" So now we have a computed expression and the key evaluates to fullname

console.log(person.fullname) → john smith

Graphical user interface, text, application

Description automatically generated

Just like you can create, read, update and delete elements of an array, you can do the same with JavaScript objects. While arrays use indexes to identify their elements, the values of a JavaScript object are identified by their property names. You can access these property names using either square bracket notation (myObject['someProperty']) or dot notation (myObject.someProperty), though the latter is the preferred way in modern JavaScript.

JavaScript objects are like a dictionary. Each word in a real dictionary should only appear once, but the same definition might apply to multiple words (in other words, multiple dictionary entries may have the same definition but an entry only appears once). In the case of a JavaScript object, each **property** must only appear once and must be unique, however, the same value can appear more than once, as long as it has a different property name. This is an example of a JavaScript object representing a person:

let person = {

firstName: 'John',

lastName: 'Smith',

age: 30,

location: 'USA'

}

There are four things we can do with this person object:

1. Read its properties
2. Create new properties
3. Update its existing properties
4. Delete its properties

All four of these are quite simple. To read an object property, you can use either square bracket or dot notation. Both are the same, but the latter is preferred:

let person = {

firstName: 'John',

lastName: 'Smith',

age: 30,

location: 'USA'

};

console.log(person['firstName']); // 'John'

console.log(person.lastName); // 'Smith'

[Run this code](http://pythontutor.com/visualize.html#code=let%20person%20%3D%20%7B%0A%20%20firstName%3A%20'John',%0A%20%20lastName%3A%20'Smith',%0A%20%20age%3A%2030,%0A%20%20location%3A%20'USA'%0A%7D%0Aconsole.log%28person%5B'firstName'%5D%29%3B%20%20//%20'John'%0Aconsole.log%28person.lastName%29%3B%20%20//%20'Smith'&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

To give an object a new property, you can use the same notation, but set the value at the same time:

let person = {

firstName: 'John',

lastName: 'Smith',

age: 30,

location: 'USA'

};

person['bestFriend'] = 'Mike';

console.log(person['bestFriend']); // 'Mike'

person.wife = 'Mary';

console.log(person.wife); // 'Mary'

console.log(person);

[Run this code](http://pythontutor.com/visualize.html#code=let%20person%20%3D%20%7B%0A%20%20firstName%3A%20'John',%0A%20%20lastName%3A%20'Smith',%0A%20%20age%3A%2030,%0A%20%20location%3A%20'USA'%0A%7D%0Aperson%5B'bestFriend'%5D%20%3D%20'Mike'%3B%0Aconsole.log%28person%5B'bestFriend'%5D%29%3B%20%20//%20'Mike'%0A%0Aperson.wife%20%3D%20'Mary'%3B%0Aconsole.log%28person.wife%29%3B%20%20//%20'Mary'%0A%0Aconsole.log%28person%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

To update an existing property you can use the same syntax as creating a new value; just do it on an existing property to overwrite it:

let person = {

firstName: 'John',

lastName: 'Smith',

age: 30,

location: 'USA'

};

person['age'] = 31;

console.log(person['age']); // 31

person.location = 'Spain';

console.log(person.location); // 'Spain'

[Run this code](http://pythontutor.com/visualize.html#code=let%20person%20%3D%20%7B%0A%20%20firstName%3A%20'John',%0A%20%20lastName%3A%20'Smith',%0A%20%20age%3A%2030,%0A%20%20location%3A%20'USA'%0A%7D%0Aperson%5B'age'%5D%20%3D%2031%3B%0Aconsole.log%28person%5B'age'%5D%29%3B%20%20//%2031%0A%0Aperson.location%20%3D%20'Spain'%3B%0Aconsole.log%28person.location%29%3B%20%20//%20'Spain'&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

Finally, to delete an object property you can use the delete operator:

let person = {

firstName: 'John',

lastName: 'Smith',

age: 30,

location: 'USA'

};

delete person['lastName'];

delete person.age;

console.log(person);

[Run this code](http://pythontutor.com/visualize.html#code=let%20person%20%3D%20%7B%0A%20%20firstName%3A%20'John',%0A%20%20lastName%3A%20'Smith',%0A%20%20age%3A%2030,%0A%20%20location%3A%20'USA'%0A%7D%0Adelete%20person%5B'lastName'%5D%3B%0Adelete%20person.age%3B%0Aconsole.log%28person%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

Note: Trying to access a property that doesn't exist will not throw an error; it will return undefined. Technically, any property that doesn't exist on an object will be undefined, so while accessing a property that has been deleted will still return undefined, if you log the object to the console the property is gone. For all intents and purposes, deleting an object property does effectively remove the property from the object.

Objects in JavaScript have three types of methods:

1. Static methods
2. Instance methods
3. Methods you create

**Static methods** are methods that exist on the Object constructor itself. They usually take an object as an argument and return some property or characteristic of that object. Common static methods you may use include Object.keys() and Object.values(), which return the passed object's properties and values, respectively. Another common static method is Object.entries(), which returns an array of the object's property/value pairs.

**Instance methods on the other hand are methods that require a specific object instance to operate on.**Two you will probably use are Object.instance.hasOwnProperty(), which returns whether an object has a property in its own definition (rather than inheriting it from another object) and Object.instance.toString(), which returns a string representation of the object.

**Methods you create** are instance methods that you yourself have added as properties on the object. This type of method is a just a property on the object whose value is a function that you can call in order to execute some code.

Let's look at the usage of some of these methods. We'll start by creating an object, then we'll:

1. Use the static methods to get some information about it
2. Use some instance methods to find out about its properties and manipulate it a bit
3. Create our own method and call it

The first step is to create an object:

let car = {

location: 'garage',

ignition: 'off',

fueled: true,

};

First, you can use the static methods Object.keys() and Object.values() to get the object's properties (keys) and values, respectively. Object.entries() can also be used to achieve the same thing, but returns the result in an array of property/value pairs. **Note:** Object.values() is not supported in JavaScript Tutor, so that line will throw an error. Try pasting the code into the dev tools console instead if you want to see that work:

let car = {

location: 'garage',

ignition: 'off',

fueled: true,

};

console.log(Object.keys(car));

// returns ["location", "ignition", "fueled"]

// Unsupported on JavaScript Tutor

console.log(Object.values(car));

// returns ["garage", "off", true]

console.log(Object.entries(car));

// returns [Array(2), Array(2), Array(2)]

// Expanded view of the three arrays:

// 0: (2) ["location", "garage"]

// 1: (2) ["ignition", "off"]

// 2: (2) ["fueled", true]

[Run this code](http://pythontutor.com/visualize.html#code=let%20car%20%3D%20%7B%0A%20%20location%3A%20'garage',%0A%20%20ignition%3A%20'off',%0A%20%20fueled%3A%20true,%0A%7D%3B%0A%0Aconsole.log%28Object.keys%28car%29%29%3B%0A%0A//%20Unsupported%20on%20JavaScript%20Tutor%0Aconsole.log%28Object.values%28car%29%29%3B%0Aconsole.log%28Object.entries%28car%29%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

Now, let's find out if the car has an ignition property, using the instance method hasOwnProperty():

let car = {

location: 'garage',

ignition: 'off',

fueled: true,

};

console.log(car.hasOwnProperty('ignition')); // true

console.log(car.hasOwnProperty('drive')); // false

[Run this code](http://pythontutor.com/visualize.html#code=let%20car%20%3D%20%7B%0A%20%20location%3A%20'garage',%0A%20%20ignition%3A%20'off',%0A%20%20fueled%3A%20true,%0A%7D%3B%0A%0Aconsole.log%28car.hasOwnProperty%28'ignition'%29%29%3B%20%20//%20true%0Aconsole.log%28car.hasOwnProperty%28'drive'%29%29%3B%20%20//%20false&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

You can also use the toString() method to return a string representation of the object:

let car = {

location: 'garage',

ignition: 'off',

fueled: true,

};

console.log(car.toString());

[Run this code](http://pythontutor.com/visualize.html#code=let%20car%20%3D%20%7B%0A%20%20location%3A%20'garage',%0A%20%20ignition%3A%20'off',%0A%20%20fueled%3A%20true,%0A%7D%3B%0A%0Aconsole.log%28car.toString%28%29%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

Notice how the representation returned is [object Object]. This is because we haven't defined the toString() method on this specific object (the car), so it's inherited from the global object which all objects inherit from. You'll learn to override this in the next unit. In the meantime, you can add your own method to turn the ignition on. This is just like any other function which you learned to create in the Comparative Programming module, except that it exists as a property of the car object:

let car = {

location: 'garage',

ignition: 'off',

fueled: true,

start: function() {

this.ignition = 'on';

}

};

console.log(car.ignition);

car.start();

console.log(car.ignition);

[Run this code](http://pythontutor.com/visualize.html#code=let%20car%20%3D%20%7B%0A%20%20location%3A%20'garage',%0A%20%20ignition%3A%20'off',%0A%20%20fueled%3A%20true,%0A%20%20start%3A%20function%28%29%20%7B%0A%20%20%20%20this.ignition%20%3D%20'on'%3B%0A%20%20%7D%0A%7D%3B%0A%0Aconsole.log%28car.ignition%29%3B%0Acar.start%28%29%3B%0Aconsole.log%28car.ignition%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

This jumps ahead a little bit but it's pretty simple to understand. We just created a property called start just like creating any other property, except this time its value is a **function** instead of a string, boolean, integer, or something else. The this in the start method refers to the car object itself, so when we **call** the function by using car.start();, ignition is changed to on. The function doesn't return anything, it just changes the ignition property to on.

In an upcoming unit you'll learn about the this keyword and how it relates to objects like this one.

Graphical user interface, application, Word

Description automatically generated

At the end of the last unit you saw how you can create an object (a car) and give it a start() method. In that method there was a reference to the this keyword:

let car = {

location: 'garage',

ignition: 'off',

fueled: true,

start: function() {

this.ignition = 'on';

}

};

console.log(car.ignition);

car.start();

console.log(car.ignition);

[Run this code](http://pythontutor.com/visualize.html#code=let%20car%20%3D%20%7B%0A%20%20location%3A%20'garage',%0A%20%20ignition%3A%20'off',%0A%20%20fueled%3A%20true,%0A%20%20start%3A%20function%28%29%20%7B%0A%20%20%20%20this.ignition%20%3D%20'on'%3B%0A%20%20%7D%0A%7D%3B%0A%0Aconsole.log%28car.ignition%29%3B%0Acar.start%28%29%3B%0Aconsole.log%28car.ignition%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

In the above code, this refers to the object it is a part of: the car. The this keyword has different meanings depending on the context in which it is used, but you'll most likely see it used in two main ways:

1. In a method, this refers to the object that owns the method. If the method isn't owned by any object, then this refers to the global object.
2. In an event (like when a user clicks on something), this refers to the element that received the event (e.g. the thing they clicked on).

Let's look at some examples of the above scenarios. The first item in the list above is exactly what you've seen with the car object in the previous unit. The start() method is owned by the car object, so within that method, this refers to the car. That means that any time we want to access a property of the car we can use this to reference it. To demonstrate, let's add some more methods to the car:

let car = {

location: 'garage',

ignition: 'off',

fueled: true,

start: function() {

**this.ignition** = 'on';

},

drive: function() {

**this.location** = 'street';

},

park: function() {

**this.location** = 'garage';

},

refuel: function() {

**this.location** = 'gas station';

**this.fueled** = true;

},

stop: function() {

**this.ignition** = 'off';

},

};

Above, the **bolded** references to this.ignition, this.location and this.fueled refer to those properties on the car object. Because the car "owns" all those methods (start, stop, refuel and so on), this refers to the car, and allows you to access it from inside each method to modify the car's properties.

// Now we can "use" the car:

car.start();

console.log(car.ignition);

car.drive();

console.log(car.location);

// Ran out of gas!

car.fueled = false;

car.refuel();

console.log(car.location);

console.log(car.fueled);

// Let's go home

car.drive();

console.log(car.location);

car.park();

console.log(car.location);

car.stop();

console.log(car);

[Run this code](http://pythontutor.com/visualize.html#code=let%20car%20%3D%20%7B%0A%20%20location%3A%20'garage',%0A%20%20ignition%3A%20'off',%0A%20%20fueled%3A%20true,%0A%20%20start%3A%20function%28%29%20%7B%0A%20%20%20%20this.ignition%20%3D%20'on'%3B%0A%20%20%7D,%0A%20%20drive%3A%20function%28%29%20%7B%0A%20%20%20%20this.location%20%3D%20'street'%3B%0A%20%20%7D,%0A%20%20park%3A%20function%28%29%20%7B%0A%20%20%20%20this.location%20%3D%20'garage'%3B%0A%20%20%7D,%0A%20%20refuel%3A%20function%28%29%20%7B%0A%20%20%20%20this.location%20%3D%20'gas%20station'%3B%0A%20%20%20%20this.fueled%20%3D%20true%3B%0A%20%20%7D,%0A%20%20stop%3A%20function%28%29%20%7B%0A%20%20%20%20this.ignition%20%3D%20'off'%3B%0A%20%20%7D,%0A%7D%3B%0A%0A//%20Now%20we%20can%20use%20the%20car%3A%0Acar.start%28%29%3B%0Aconsole.log%28car.ignition%29%3B%0Acar.drive%28%29%3B%0Aconsole.log%28car.location%29%3B%0A%0A//%20Ran%20out%20of%20gas!%0Acar.fueled%20%3D%20false%3B%0Acar.refuel%28%29%3B%0Aconsole.log%28car.location%29%3B%0Aconsole.log%28car.fueled%29%3B%0A%0A//%20Let's%20go%20home%0Acar.drive%28%29%3B%0Aconsole.log%28car.location%29%3B%0Acar.park%28%29%3B%0Aconsole.log%28car.location%29%3B%0Acar.stop%28%29%3B%0Aconsole.log%28car%29%3B&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

Again, throughout the above code the keyword this always refers to the car, so we can use it to perform all sorts of operations on the car's other properties. If someone renamed the car object to something else, like truck, everything would still work with the new name since the reference to the **object itself** is stored in this. However, if you take one of the methods out of the car object and assign it in the global context, it no longer has any reference to the car, so this becomes undefined:

let car = {

location: 'garage',

ignition: 'off',

fueled: true,

start: function() {

this.ignition = 'on';

},

// ...

};

// car.start is a function, so really

// all we're doing here is defining it again

// under a new name to demonstrate this concept

let myStartFunction = car.start;

myStartFunction(); // TypeError

[Run this code](http://pythontutor.com/visualize.html#code=let%20car%20%3D%20%7B%0A%20%20location%3A%20'garage',%0A%20%20ignition%3A%20'off',%0A%20%20fueled%3A%20true,%0A%20%20start%3A%20function%28%29%20%7B%0A%20%20%20%20this.ignition%20%3D%20'on'%3B%0A%20%20%7D,%0A%20%20//%20...%0A%7D%3B%0A%0Alet%20myStartFunction%20%3D%20car.start%3B%0AmyStartFunction%28%29%3B%20%20//%20TypeError&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=js&rawInputLstJSON=%5B%5D&textReferences=false)

In JavaScript Tutor, this code will throw an error because once the start() function is removed from the car object, the connection between this and the car is gone, so this no longer has an ignition property.

The second common way you'll use this is to refer to an HTML element. You'll see this a lot, since you will often use JavaScript to handle events within the context of HTML elements. In this context, this refers to the element you're operating on. You'll explore this further in later units but here's an example:

<button onclick="this.style.color='blue'">Click to Change My Text Color!</button>

Click to Change My Text Color!

When you click the button, its text changes to blue since thanks to the onclick property on the button object which sets this.style.color to blue. As you work with JavaScript more and more, you'll often find yourself making changes to HTML elements and in this context you should remember that this always refers to the element itself. In the above code, it refers to the button, allowing the user to change its text color when clicked.

To summarize, remember that there are two main ways you'll use this in JavaScript: First, to refer to an object from inside it, and second, to reference an element that a user is interacting with.

Graphical user interface, text, application

Description automatically generated

A picture containing text

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Text

Description automatically generated with low confidence

**The Math object in JavaScript is a built-in utility for working with the Number data type. It contains several useful predefined functions, constants and properties which can be used for all sorts of mathematical operations. Some examples of things included in the Math object are:**

* **Math.PI: The constant pi**
* **Math.random(): A method for returning a random number between 0 and 1**
* **Math.abs(): A method for finding the absolute value of a number**
* **Math.min(): A method for returning the minimum of a series of numbers**
* **Math.max(): A method for returning the maximum of a series of numbers**
* **...and many more!**

Table

Description automatically generated with low confidence

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

Graphical user interface, application, Teams

Description automatically generated

**While all modern browsers support JavaScript, it is possible for users to disable it or for it to be disallowed for security reasons in certain circumstances. For these situations, there's a special HTML tag, <noscript>, that you can use to let the user know that their browser isn't currently allowing or supporting JavaScript, rather than showing them a blank page or an error.**

**Using the <noscript> tag is simple. Just enclose anything you want to display for JavaScript-disabled users in it like any other HTML element:**

**<html>**

**<head>**

**<title>HTML <noscript> Tag</title>**

**</head>**

**<body>**

**<script type="text/javascript">**

**console.log("Hello!");**

**</script>**

**<noscript>**

**Your browser does not support JavaScript!**

**</noscript>**

**</body>**

**</html>**

**IMPORTANT: When using <noscript> in the <head> element, you can only use <link>, <style>, and <meta> elements within it. That means it can't render any sort of text, paragraphs, or headings. As a rule of thumb your <noscript> content should be as basic as possible, using no complex HTML or styling, and you should avoid using <noscript> in the <head> unless you have a very specific reason to do so.**

Table

Description automatically generated

**In JavaScript, like all programming languages, there are certain words that are used in the syntax of the language itself. Examples of these words include if, break, Date, this and many others. These are known as reserved words because they are reserved for use within the syntax of the language itself and should never be used to name variables, functions, classes or any other type of object. The reason you should never use these words outside their intended context is that JavaScript won't know how to tell the difference between your intended use and the normal intended use.**

**Along with these reserved words, there are reserved object names such as Array, Object, Math and so on. You should never use these or any other built in objects or function names to name your variables or your own objects. Below is a list of common reserved words in JavaScript (note that this list is extensive but not exhaustive):**

* **abstract**
* **boolean**
* **break**
* **byte**
* **case**
* **catch**
* **char**
* **class**
* **const**
* **continue**
* **debugger**
* **default**
* **delete**
* **do**
* **double**
* **else**
* **enum**
* **export**
* **extends**
* **final**
* **finally**
* **float**
* **for**
* **function**
* **goto**
* **if**
* **implements**
* **import**
* **in**
* **instanceof**
* **int**
* **interface**
* **let**
* **long**
* **native**
* **new**
* **package**
* **private**
* **protected**
* **public**
* **return**
* **short**
* **static**
* **super**
* **switch**
* **synchronized**
* **this**
* **throw**
* **throws**
* **transient**
* **try**
* **typeof**
* **var**
* **void**
* **volatile**
* **while**
* **with**
* **yield**

Chart, waterfall chart

Description automatically generated

**The core of your ability to use JavaScript to interact with the browser and thus HTML documents lies in three high-level objects:**

1. **window: The global window object which represents the browser window**
2. **document: The document containing the HTML document, which lives inside the window**
3. **document.body: The body of the HTML document**

**The three items above can be thought of as the top of the DOM tree. Hierarchically, the window is the parent of everything and represents the browser window. The document lives inside the window, and the body is a property of the document, representing the <body> element of the HTML document. All three of these objects have a number of useful methods and properties with which JavaScript can interact. In fact, in the previous lesson you saw two units where the example given updated a <span> element with the number of seconds since the user loaded the page. You might have noticed that in the JavaScript code we accessed the <span> element using the document:**

**let timerElement = document.getElementById('timer');**

**Here, the document.getElementById() method was used to access the specific span element we wanted, based on its id attribute, which was timer. The window object also has a number of useful methods and properties like this, such as:**

* **window.document: The document object in the example above**
* **window.history: A reference to the window's history (i.e. browser history)**
* **window.innerWidth/window.innerHeight: The inner size of the current window including the scrollbars but not including menu bars**
* **window.outerWidth/window.outerHeight: The outer size of the current window including menu bars**
* **window.location: A getter/setter for the current URL**
* **window.localStorage: A storage medium for storing data the user might need for the page to function properly, such as cookies.**

**It's not important that you know every single method of the document, window and body. What's more important is to understand that they are your access point for everything you will ever need to do to make JavaScript interact with your websites. You'll learn many different ways to use these objects in the upcoming lessons. Try popping the runnable example below out into a new window and experimenting with changing the window size, the browser size, and adjusting the various functions and properties that are being used. For a complete reference to the objects in this unit, see the following resources:**

* [**Window**](https://developer.mozilla.org/en-US/docs/Web/API/Window)
* [**Document**](https://developer.mozilla.org/en-US/docs/Web/API/Document)
* [**Body**](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/body)

Diagram

Description automatically generated

**The DOM can actually be thought of as an upside down tree, where the trunk is at the top and all the branches and leaves hang off the bottom. From here we can make the analogy that:**

1. **The entire tree (the document) lives inside the window**
2. **The trunk of the tree is the <html> element, from which all other elements (branches) emanate**
3. **From the trunk, the tree has two major branches, <head> and <body>**
4. **The smaller branches of the tree (links, headings, paragraphs, etc) emanate from either the <head> or the <body> element**
5. **Each branch of the tree has a unique path you must take to access it, beginning from the trunk and following the branches to get there**
6. **At the end of each branch is a leaf. The leaves represent the actual content (text) of the webpage**

**Document Nodes:**

**In more technical terms, every item described in the tree above - the elements, branches, leaves and so on - are called nodes. What you will actually be doing with JavaScript is navigating around the DOM tree to access different nodes, modifying them, deleting them, adding new ones and so on. The nodes represent everything from the HTML element itself to individual HTML elements, paragraphs, links, headings, images, even classes and attributes. In the DOM tree, everything is a node, and each node is an object with properties that can be manipulated, created and deleted. Over the next several units, you'll learn many different methods for accessing and manipulating these DOM nodes with JavaScript.**

A picture containing diagram

Description automatically generated

**If you need something more specific, like the first paragraph in the list, a specific one based on its id, or the number of paragraphs found, you can access them by attaching the appropriate property or index to the end of the function call, like this:**

**let firstParagraph = document.getElementsByTagName('p')[0];**

**let numParagraphs = document.getElementsByTagName('p').length;**

**let secondParagraph = document.getElementsByTagName('p').item(1);**

**let specificParagraph = document.getElementsByTagName('p').namedItem('specific');**

**console.log(firstParagraph);**

**console.log(numParagraphs);**

**console.log(secondParagraph);**

**console.log(specificParagraph);**

**In the above, though, the document.getElementsByTagName('p') is becoming quite repetitive. This is a good chance to refactor the code a bit. If you know you'll be using the same variable over and over, like the list of paragraphs above where we might want do a number of different things with it, you can assign the HTMLCollection itself to a variable, and then use the properties/indexes on the variable instead. Here's an example that achieves the same as the above, but in a less verbose way:**

**let allParagraphs = document.getElementsByTagName('p');**

**console.log(allParagraphs); // The HTMLCollection**

**console.log(allParagraphs[0]); // The first paragraph in the document**

**console.log(allParagraphs.length); // The number of 'p' elements found**

**console.log(allParagraphs.item(2)); // The 'p' element at index 2 in the collection**

**console.log(allParagraphs.namedItem('specific')); // The 'p' element with id="specific"**

Timeline

Description automatically generated with low confidence

The getElementsByClassName() method is nearly identical in functionality to the getElementsByTagName() method. It is also a method of the document object and it also returns an HTMLCollection, so all the same functionality is available to you. The syntax is identical as well; just pass the method the class name you wish to find:

document.getElementsByClassName('className');

The only difference between the getElementsByClassName method and the getElementsByTagName() method is that the former returns elements matching the **class** you specify, which could be different types of elements, whereas the latter will return **only** elements matching that specific type. In other words, if you've got a class like blue which you're applying to some <h1> elements, some <p> elements and some <div> elements, using getElementsByClassName('blue') will get all three types of elements because they all have class="blue". getElementsByTagName() is used to get only one specific kind of element.

To use the getElementsByClassName() method, simply pass it the class name you wish to find and it will return all the matching elements:

let blueThings = document.getElementsByClassName('blue');

console.log(blueThings); // The HTMLCollection

console.log(blueThings[0]); // The first element with class="blue" in the document

console.log(blueThings.length); // The number of elements with class="blue" found

console.log(blueThings.item(2)); // The element at index 2 in the collection

console.log(blueThings.namedItem('specific')); // The element with both class="blue" and id="specific"

Graphical user interface

Description automatically generated

The getElementById() method gets a specific element: the one with the id attribute you specify. This method is slightly different from the other two you've learned in this lesson because it returns a **specific** element, not a collection of elements. While the id attribute is supposed to be unique to only one element on the page, in the event that the developer has made a mistake and more than one element is found with the same id, this method will return only the first one that matches.

One common slip-up with this method is mistyping it as getElementsById() (plural). The easiest way to avoid this is to remember that the method is supposed to return a **single element**. To use the getElementById() method, just pass it the id of the element you want to retrieve, and if an element with that value for its id attribute exists, it will be returned. If no element exists that matches, it will return null:

// Gets the element with id="third-heading", if it exists (and even if it's not a heading!)

let specificHeading = document.getElementById('third-heading');

It's important to remember that this method does not care what type of element it is. It's only looking at the id attribute, so the above code would return a paragraph, div, button, heading, or any other element as long as it had id="third-heading". The getElementById() method is one of the most commonly used methods in JavaScript when using it to manipulate the DOM. A runnable example below shows how you can get different specific elements from the DOM tree.

Diagram

Description automatically generated

In HTML, all elements are related in some way. The entire DOM tree is a hierarchical, familial structure where parents have children, children have siblings, siblings can have children, children can have other children and so on. This structure is critical to your understanding of how to navigate and manipulate the DOM with JavaScript, and there are several methods available within JavaScript that allow you to access different nodes based on these relationships. In the image here, the li elements are both parents and children ... they are children of the ul elements and parents to anything contained within them. The ul element could also have siblings, and is most likely a child itself of a higher element.

The DOM element object, which represents a specific HTML element, has a number of properties and methods you can use to access its related elements in the DOM tree. Four of the most important properties available are related to finding the element's parent, children and siblings:

* .parentNode: Gets the current element's parent node. If the current element is anything other than <html>, this property will return the element's parent element. Otherwise, it will return the document object.
* .children: Returns an HTMLCollection of the elements children (anything nested within it in the DOM tree).
* .nextElementSibling/.previousElementSibling: Finds the next sibling/previous sibling of the given element, i.e. the closest one at the same nesting level in the DOM tree.

By combining these four methods with your ability to get an element by its tag name, class name or id, you can efficiently navigate around the DOM and access any element you need. To access the associated property from above on any element, just attach it to the element at hand. For example, consider that we have a div element with an id of main-div:

let mainDiv = document.getElementById('main-div'); // First get the div itself

let parent = mainDiv.parentNode; // the div's parent element

let children = mainDiv.children; // the div's children

let firstChild = mainDiv.children[0]; // the div's first child

let nextSibling = mainDiv.nextElementSibling; // the next element at the same nesting level

let prevSibling = mainDiv.previousElementSibling; // the previous element at the same nesting level

Graphical user interface, text, application

Description automatically generated

You now know how to access elements in the DOM tree in a multitude of ways, such as:

* Using their tag name
* Using their class or id
* Through their parents, children and siblings

Once you've accessed the element you want, you can do almost anything you want to manipulate it.

The HTML DOM Element Object is like any other object in JavaScript. It has properties and methods and can be created, read, updated and deleted. Since JavaScript is used to manipulate and modify the DOM, you should know how to access common element properties in order to change the elements in the ways you want. Some commonly used element properties are listed below:

| **Name** | **Description** |
| --- | --- |
| tagName | A string representing the tag name of the element, such as P, DIV or H1 |
| id | The value of the id attribute on the element, as a string |
| className | The value of the class attribute on the element, as a string |
| classList | The list of classes in the class attribute as an array-like structure which can be iterated |
| attributes | A NamedNodeMap of all the element's attributes and their values. It can be iterated |
| clientHeight | A Number representing the inner height of the element |
| clientWidth | A Number representing the inner width of the element |
| innerHTML | The content inside the element including HTML markup |
| outerHTML | Same as innerHTML, except it includes the element itself as well |

Graphical user interface, application, Word

Description automatically generated

In the data structures lessons you learned that with JavaScript objects, you can gain access to the object itself by using the keyword this. You might remember that we created a car object that had a few properties (pictured here), and then we created a start function inside it that referred to the car itself and set its ignition property to "on" using the code this.ignition = 'on'. This is common with JavaScript objects, and because everything in JavaScript is an object - even HTML elements - you can use this to refer to HTML elements too.

Most of the time you'll use the concept of this referring to an HTML element in the context of a **function** that manipulates that object in some way. The following code draws on some things you'll learn in future units, but it's not important to understand it perfectly for now; just focus on the part about this:

function addOne() {

let myNumber = **this.innerHTML**;

let newNumber = parseInt(myNumber) + 1;

**this.innerHTML** = newNumber;

}

In the code, this actually refers to the following div element. Click it and see what happens!

**1**

Behind the scenes, what we're actually doing here is getting the div (which has an id of myDiv, so we can get it with document.getElementById('myDiv')), and attaching a "listener" to it to listen for clicks. When you click the div, it calls (executes) the function above. You'll learn how to do this yourself in a later unit, but what's important to see here is that in the function that actually runs when the div is clicked, **this is referring to the div itself!**

That means that when you click it, the first thing that happens is a variable called myNumber is created from the div's innerHTML property, then the function adds 1 to the number, and finally sets this.innerHTML (again this is referring to the div itself) to the new number. In short, this code uses this to say "get this div's inner HTML, add one to it, and then overwrite its innerHTML with the new number". To do that, it uses the this keyword to identify and refer to the div.

You can use this to get or set any property from any element in the DOM. It's really useful for doing things like determining which button a user clicked on, finding the height or width of an element, changing the position of something on the screen or getting the value out of a form field. This concept is really important to understand because when there are hundreds of elements on the screen and you need to figure out which one the user is interacting with, sometimes using this is the only way to know for sure. If you're curious and want to see the real code that makes this work, you can right click on the div and click inspect. Just under the div itself you'll see the JavaScript code in a script tag. You should certainly feel free to poke around there a bit and try to understand how it works, but remember that you'll learn this in detail in a later unit so the important thing to focus on right now is simply the idea that **in the context of the DOM, this can be used to refer to specific elements when you need to determine which one a user is interacting with.**

Graphical user interface, text, application, chat or text message, email

Description automatically generated

The document.write() method writes content to the HTML document. It can write plaintext, HTML, or JavaScript code. When you call document.write() on an existing HTML document, it will replace the *entire* document including the <html> element itself, which means you lose all the CSS, any JavaScript you've loaded, and all of the content.

To use this method, you simply pass it the content you wish to write as a string:

document.write("<h1>Hello World!</h1>");

You can use the document.write() method to write JavaScript code as well, like printing the date to the document:

document.write(Date());

document.write() can take multiple arguments separated by commas and will print each one in order, but it will not add a new line if you don't use a block level element or you're writing plaintext. To add a new line, you need to add it manually with a <br> tag or write with block-level elements like paragraphs, headings and divs. The runnable example below demonstrates this functionality.

Graphical user interface, text, application

Description automatically generated

let myElement = document.getElementById('some-id-attribute');

myElement.style.width = '300px';

Graphical user interface, text, application, email

Description automatically generated

Along with updating the properties on existing elements, with JavaScript you can create entirely new elements and append them to the DOM in whatever location you wish, or you can remove, replace or relocate elements. There are several methods available for you, both as part of the document object itself as well as on each individual HTML element object. All the standard JavaScript functionality is there for you too, such as loops, conditional statements and data structures. Some of the most common methods you'll use when creating and removing elements are:

* document.createElement(): Create a new element
* element.appendChild(): Append an element as the child of another element
* element.cloneNode(): Clone an entire element. Pass true to "deep clone" which clones its children also
* element.remove(): Remove an element from the DOM

Graphical user interface, text, application

Description automatically generated

Sometimes it becomes quite tedious to create, clone, remove, replace and relocate lots of elements with JavaScript when you're trying to update the DOM. When this happens, as usual, there's a better way. The innerHTML property allows you to get or set the entire internal HTML contents of an HTML element. If that element contains other HTML, the innerHTML property will replace it if you set it, or return all the HTML if you get it. Like all the other properties on an element, it's simple to use. Here's how you can both get and set this property after using one of the many methods to retrieve an element:

let myElement = document.getElementById('some-id');

// GET the innerHTML:

let html = myElement.innerHTML;

console.log(html); // Whatever HTML was already in the element

// SET the innerHTML (this replaces it)

myElement.innerHTML = "<h1>Hello world!</h1>"

This property exists on every element in the DOM tree, right up to the <html> element itself. If you are working with a low level element such as a heading or something that just has some text in it, the innerHTML property will return the text content of the element. You can also set the innerHTML property to simple plaintext; it doesn't have to be HTML. The real power of this property, though, is in its ability to replace the entire contents of an element with complex HTML, such as replacing all the contents of a div with a whole bunch of new HTML. To create this new HTML, you can use template literals, which you learned about in the strings lesson. Just surround all your HTML with backticks!

let myDiv = document.createElement('div');

let html = `

<h1>Hello!</h1>

<p>Here is some complex HTML that will be inserted into this div.</p>

<p>It contains a heading, paragraphs, and even a list with a class.</p>

<ul class="special-list">

<li>Item 1</li>

<li>Item 2</li>

<li>Item 3</li>

</ul>

`;

myDiv.innerHTML = html; // The div will now render all the above HTML!

Diagram, timeline

Description automatically generated

here is a table containing the most common mouse events used when tracking mouse clicks:

| **Event/Property** | **Description** |
| --- | --- |
| **EVENTS** | |
| onclick | The event occurs when the user clicks on an element |
| oncontextmenu | The event occurs when the user right-clicks on an element to open a context menu |
| ondblclick | The event occurs when the user double-clicks on an element |
| onmousedown | The event occurs when the user presses a mouse button over an element |
| onmouseup | The event occurs when a user releases a mouse button over an element |
| **PROPERTIES** | |
| button | Returns which mouse button was pressed when the mouse event was triggered |

Diagram

Description automatically generated

here is a table containing the most common mouse events used when tracking mouse movements:

| **Event/Property** | **Description** |
| --- | --- |
| **EVENTS** | |
| onmouseenter | The event occurs when the pointer is moved onto an element |
| onmouseleave | The event occurs when the pointer is moved out of an element |
| onmousemove | The event occurs when the pointer is moving while it is over an element |
| onmouseout | The event occurs when a user moves the mouse pointer out of an element, or out of one of its children |
| onmouseover | The event occurs when the pointer is moved onto an element, or onto one of its children |
| onscroll | The event occurs when an element's scrollbar is being scrolled, or when scrolling with the mouse/trackpad. |
| **PROPERTIES** | |
| clientX | Returns the horizontal coordinate of the mouse pointer, relative to the current window, when the mouse event was triggered |
| clientY | Returns the vertical coordinate of the mouse pointer, relative to the current window, when the mouse event was triggered |

Diagram

Description automatically generated

There are a few common loading events that you should know in your JavaScript journey. These events are fired every time an HTML page is loaded and provide you with a way to track when certain elements in the page have completed loading. This is known as the "lifecycle" of an HTML page. The four main events that you will encounter are:

1. DOMContentLoaded: The DOM content has finished loading, but external assets like images, JS files, CSS files and so on have not loaded yet.
2. load: Everything is now loaded. This event fires on the window
3. beforeunload: Fired just before the user leaves the page. This can allow you to confirm whether the user really wants to leave the page.
4. unload: Fired when the user has confirmed they want to leave the page.

The event you are most likely to use is the window.onload event, which allows you to execute some JavaScript code immediately after you've confirmed everything on the page is loaded. This is useful for those times when you want to automate some sort of action every single time the page loads, such as initiating a popup, populating DOM elements with content, or rendering some content that depends on something else being loaded first. In the HTML lifecycle, the document will always complete loading before the window, and you can use the onload event on both objects. To use this event, just set the onload property on the window to whatever JavaScript you would like to execute:

window.onload = console.log('The window has loaded!');

Or more practically, execute a function:

function myFunction() {

console.log('The window has loaded!');

// You can do anything you want in here!

}

window.onload = myFunction();

You can also attach the event to a specific element by using it as an attribute. This can allow you to determine when certain content is loaded, such as if you wanted to determine when a specific script or image has finished loading:

function myFunction() {

console.log('This will fire when the body is loaded');

}

<body onload="myFunction();"></body>

Due to event "bubbling" in JavaScript, logging these messages to the console may not always occur in the correct order and thus it's best to just use window.onload to determine when the page is completely loaded, and window.onbeforeunload and window.onunload for determining what to do when a user exits the page. You can see this phenomenon in the following runnable example, where the body's onload event bubbles up to the document, causing the document's console.log to be printed first.

Graphical user interface, text, application, email

Description automatically generated

Knowing how to capture click events, mouse movements and loading events is great, but there are other types of events you must know in order to take advantage of everything JavaScript has to offer. Another type of event you'll need to be familiar with is the change event, which you can capture with the onchange event handler. This event is fired when certain elements change, in particular <input> and <select> elements. These events allow you to capture when a user has changed something in a form and respond accordingly, allowing you to handle powerful operations like live form validation.

The onchange event handler is simple to use. Like all the other events, you can simply attach it to one of the supported elements (either an <input> or a <select> element) as an attribute, and set the attribute equal to some JavaScript code or a function you'd like to call:

<input type="text" onchange="console.log('You changed the text box!');">

If you're working with a dropdown box, the syntax is the same but the event will fire whenever the option in the box is changed by a user changing it with the mouse or the keyboard:

<select name="my-select" id="my-select" onchange="console.log('You changed the select box!');">

<option value="1">1</option>

<option value="2">2</option>

<option value="3">3</option>

<option value="4">4</option>

</select>

One key thing to recognize about the change event is that it is **not** fired when changing elements programmatically. It's intended to capture changes that are committed to by the user, such as typing something into a text box and then moving on to a different part of the form, so it will only be fired once the typing is complete or the select box or other input has been completely changed to the new value.

Shape

Description automatically generated

| **Property/Method** | **Description** |
| --- | --- |
| altKey | Returns whether the "ALT" key was pressed when the key event was triggered |
| charCode | Returns the Unicode character code of the key that triggered the event |
| code | Returns the code of the key that triggered the event |
| ctrlKey | Returns whether the "CTRL" key was pressed when the key event was triggered |
| getModifierState() | Returns true if the specified key is activated |
| isComposing | Returns whether the state of the event is composing or not |
| key | Returns the key value of the key represented by the event |
| keyCode | Returns the Unicode character code of the key that triggered the onkeydown or onkeyup event |
| location | Returns the location of a key on the keyboard or device |
| metaKey | Returns whether the "meta" key was pressed when the key event was triggered |
| repeat | Returns whether a key is being held down repeatedly, or not |
| shiftKey | Returns whether the "SHIFT" key was pressed when the key event was triggered |
| which | Returns the Unicode character code of the key that triggered the onkeydown or onkeyup event |

Graphical user interface, text, application, email

Description automatically generated

Up to this point all the listeners you've created to listen for JavaScript events have been created via HTML attributes such as onclick, onchange and so on. These event listeners work just fine, but they can make your code messy and verbose when you've got a lot of them. Fortunately as with most things in code, when things become too verbose it's indicative that there's a better way. This time, the better way is through the use of **event listeners**.

An event listener sets up a function that will be called whenever the specified event, such as an input change, a click, a mouse movement, a keypress or other such event is delivered to the **target**. The **target** in this context is anything that supports events, but commonly will be something like an HTML element, the document or the window. Event listeners allow you to remove those verbose HTML attributes and instead put all the functionality in JavaScript. This is good for separation of concerns too, since even though JavaScript and HTML go hand in hand, their code should still be as segregated as possible to limit complexity and increase maintainability.

To add an event listener to an object you need three things:

1. The object to add the event listener to, like an element, the document or the window
2. The addEventListener() method, which takes two parameters described below
3. The **listener**, which is a function to run when the event is received and which takes the event itself as a parameter

As a simple example, if you have a button you want to use to run some JavaScript code when it's clicked, the button is the object and you can pass addEventListener() the click event and the function to run when triggered. This might sound complex, but it's actually quite simple. Here is the generic syntax for adding an event listener:

target.addEventListener(type, listener [, options]);

The options are parameters you can specify to make your listener behave differently, but they're not required to create it. You can pass options such as whether to only invoke the listener on the first event it receives and then destroy it, whether it will prevent the default action of the original object and so on. Don't worry about that stuff for now though; what you need to know right now is just how to add a simple listener for an event, and everything else you can pick up as you need to create more complex listeners. The following compares the way you've seen things done thus far with the way they should be done as a best practice:

**Without an event listener, using HTML attributes:**

<button id="my-button" onclick="console.log('You clicked the button!')">Click me!</button>

**Using an event listener:**

**HTML:**

<button id="my-button">Click me!</button>

**JavaScript:**

function myFunction (event) {

console.log('You clicked the button!');

console.log(this.id); // my-button

}

let myButton = document.getElementById('my-button');

myButton.addEventListener('click', myFunction);

There are a few important things to note here:

1. The listener itself, myFunction() in this case, takes the event as a parameter. That means that if you need to get any information about the event, such as its type, name, properties or anything else, you can get it using dot notation, like you saw in the keyboard events unit to get event.key, which represented the key that was pressed.
2. When passing the listener to the addEventListener() method, it is passed **without parentheses**. This is vitally important because if you pass the function with parentheses it will be called immediately instead of when the event is received! Remember that you are passing **the function itself** as the thing to do **when** the event happens, not calling the function right then and there.
3. Inside the listener (myFunction(), which is sometimes called a **callback function**, by the way), the value of this is the object that triggered the event. In most cases that you use this functionality, that will mean that this refers to the element, so you can get things like this.id, this.classList or any other element property.

You can have as many event listeners as you want, and every element can have multiple listeners listening to different kinds of events, but you should never have two listeners for the same event on the same element executing different callback functions. This will cause one of them to be discarded and will result in unpredictable behavior.

Text

Description automatically generated

Throughout this module you've seen functions used for a few different things. One of the most common ways developers use functions is for handling JavaScript events, because they allow you to keep your code concise, reusable and easy to maintain. In the last unit, you saw how it's possible (and preferred) to write JavaScript functions that act as **event listeners** so that you can keep your JavaScript and HTML segregated and clean. With this in mind, in this unit we'll dive a little deeper into what else is available when working with these functions.

When working with event handlers, a few of the things you might want to ask yourself are:

1. Which event was or will be fired, and what caused/will cause it to be fired?
2. What do I want to happen in response to this event?
3. Which element or elements is the response going to affect, or if it won't change any elements, what will it do for the user?
4. What information do I need to be able to handle the event correctly?
5. Where can I get the information needed (from the event itself, from the element/window/document, or somewhere else)?

To demonstrate, there are two major concepts you should understand about working with functions and event handlers in JavaScript:

**Concept 1: The event is accessible in the function:**

You saw in the keyboard events unit that it was possible not only to determine that the user pressed a key on the keyboard, but also whether they were holding it down, when they released it, which key they pressed, and a host of other things. All of this information came from the event object that was passed to the event handler (i.e. the function that was executed). The KeyboardEvent object has specific properties related to keyboard events and the same is true for the other types of events. You can use the event object in the function that handles the event to get more information about exactly what happened and respond accordingly.

**Concept 2: The element is accessible in the function:**

You also saw in the last unit that when you create an event listener using the addEventListener() method, inside of the event handler this refers to the object that you attached the listener to. This is very useful to know because it means that in the callback function that actually handles the event, you can access the event type and all its properties and methods as well as **the element that triggered it**, which gives you access to everything about that element: its type, height, width, behaviors, id, classes, location in the DOM and so on.

Understanding these two concepts is the key to understanding how to manipulate the DOM in response to JavaScript events.

Graphical user interface, text, application

Description automatically generated

Working with forms in JavaScript is a critical part of adding interactivity to websites. With JavaScript, you can not only get and set the values of different form fields, but also handle form submission and submit forms automatically. Forms don't necessarily need to be submitted to a server either: you can capture the onsubmit event and execute JavaScript in response to it, which means you can use forms to add, update, and remove DOM elements and manipulate the DOM in the same ways you can with button clicks, mouse movements and other events.

The first step to understanding how to work with forms in JavaScript is to learn how to get the values that a user has submitted. Take a simple login form for example:

<form method="POST" action="/login/">

<div class="username">

<label for="username">Username:</label>

<input type="text" id="username" name="username">

</div>

<div class="password">

<label for="password">Password:</label>

<input type="password" id="password" name="password">

</div>

<input type="submit" value="Login" />

</form>

In the above form, the username and password fields will be submitted to the /login/ URL on the server. On the server side, you would most likely access their values via the name attribute, which would be submitted as part of the POST request. With JavaScript you can access those values on the client side by getting the elements by their id and then accessing their value attribute. This allows you to manipulate, validate or do anything else you wish with the values before submitting them, or prevent submission altogether and do something else:

let user = document.getElementById('username');

let pass = document.getElementById('password');

// Now you can get their values

console.log(user.value);

console.log(pass.value);

Most form fields can be accessed in this way, but keep in mind that for some fields like checkboxes and multi-selection dropdowns, you may need to use more complex logic to access their values or whether or not they are "selected". You'll learn more about these other form operations as you write more complex code, and there are also JavaScript libraries that can serialize an entire form into a nice JavaScript object in a single line of code for these more complex situations!

Graphical user interface, text, application

Description automatically generated

Of course because an input element in an HTML form is no different than any other element, it's possible to **set** the value attribute as well as getting it, using JavaScript. This means that just like you can get values out of a form, you can put values into a form as well. This is useful for all sorts of things, from adding a little convenience for your users to creating fully interactive forms that build themselves as the user fills them out.

Consider an example such as a contact form, where a registered user might need to fill out their username, their email address and a message:

<form method="POST" action="/contact-us/">

<div class="username">

<label for="username">Username:</label>

<input type="text" id="username" name="username">

</div>

<div class="email">

<label for="email">Email:</label>

<input type="email" id="email" name="email">

</div>

<div class="message">

<label for="message">Message:</label>

<textarea id="message" name="message"></textarea>

</div>

<input type="submit" value="Submit" />

</form>

The user probably already submitted their email when registering for the site, and as a registered user they will already have a username. For the user, it's annoying and inconvenient to have to fill out three fields when their username and email could be prepopulated, especially if they send a lot of messages and thus have to fill out the form a lot. Instead, the developer could get their username and email out of a database and load the form with those fields already filled out:

let user = document.getElementById('username');

let email = document.getElementById('email');

user.value = 'johnsmith1';

email.value = 'john@johnswebsite.com';

The above is simplistic but it demonstrates the functionality well. You might be asking yourself why you wouldn't just use the value attribute right on the form fields themselves, like this:

<input type="text" id="username" name="username" value="johnsmith1">

...

<input type="email" id="email" name="email" value="john@johnswebsite.com">

You're not wrong. You could do this. However, this restricts the value of the fields to something static, and it also doesn't allow you to do anything *before* filling out those fields. By doing it in JavaScript, you could execute all kinds of JavaScript code before filling out the fields, such as checking how long the user has been a member of the site, validating that their email address is confirmed, or anything else you can imagine. Prepopulating form fields with JavaScript is also useful for things like filling out parts of a user's address based on their location, upselling by preselecting extra options in checkout forms, and a multitude of other things. None of this would be possible with simple HTML using the value attribute.

Graphical user interface, text, application, email

Description automatically generated

To effectively work with forms in JavaScript, you need to be able to populate and get the values of fields and also know how to handle form submission. To do this you'll need to understand three main things:

1. How to capture the submit event when a user submits a form, so you can handle the event in JavaScript
2. How to submit a form yourself, so you don't need to depend on a user to do it
3. How to prevent the default behavior of the form, if you're going to override it with JavaScript

Luckily all three of these concepts are quite simple. Let's look at each one individually:

**Capturing the Form's Submit Event:**

You've already learned about event handlers in JavaScript. The form element's submit event is just the same as the others you've learned. To capture it there are two things you can do:

1. Attach the onsubmit attribute to the form element itself in the HTML
2. Attach an event listener to the form and listen for the submit event

The second option, like with the other events, is the preferred option because it gives you more flexibility and keeps your code more segregated, clean and easy to maintain. For completeness though we'll look at both options:

**Using the onsubmit attribute:**

**index.html:**

<form id='login-form' action="" method="POST" onsubmit="handleSubmit(event);">

...

</form>

**script.js:**

function handleSubmit(event) {

// handle the event

}

**Using the submit event with an event listener:**

**index.html:**

<form id='login-form' action="" method="POST">

...

</form>

**script.js:**

function handleSubmit(event) {

// handle the event

}

let loginForm = document.getElementById('login-form');

loginForm.addEventListener('submit', handleSubmit);

In the form handler (the handleSubmit() function above) you can access any of the form values needed using the value attribute on each form element. What you decide to do with those values is up to you. A common workflow might be capturing all the values, creating a JavaScript object to hold them all, and POSTing them to the server from JavaScript, waiting for the response, clearing the form and sending the user a success message allowing them to submit the form without even refreshing the page! To do that you'll need to understand the next section:

**Submitting a Form With JavaScript:**

Every form element in HTML has a submit() method you can use to submit the form using JavaScript. To use it is simple; just get the form element itself, do whatever processing you need to do with the values the user has entered, and then call .submit():

**index.html:**

<form id="login-form" action="/login/" method="POST">

...

</form>

**script.js:**

function handleSubmit(event) {

...

loginForm.submit();

}

let loginForm = document.getElementById('login-form');

loginForm.addEventListener('submit', handleSubmit);

The above example is identical to the previous one except that it adds the submit() method at the end of the handleSubmit() function. This is a common workflow: the user fills out the form, the site captures the submit event with an event listener, does some processing or validation of the form values, calls form.submit() and returns some sort of response to let the user know whether or not it was successful. There is one more thing you'll need in order to do this cleanly though:

**Preventing the Default Form Behavior:**

By default, when you create an input with type="submit", its default behavior is to submit the form to the url specified in the action attribute. Without JavaScript, this usually means that the page will reload as the data is submitted to a server via an HTTP GET or POST request. With JavaScript, though, you might want to do any number of things in response to a form submission, and in many cases you may not want the page to refresh or you might want to use the form for something totally different than its default behavior. For this purpose, there is event.preventDefault();, which stops the default action of an element in its tracks and allows you to do whatever you want instead. Whenever you want to intercept the submission of a form in JavaScript, you should always prevent the default action if you plan to submit the form after processing it with your JS code. Otherwise, clicking the submit button will trigger both your JavaScript code as well as the default action and will result in unexpected behavior and potentially duplicate form submissions. Using preventDefault() is simple, just add it as the very first thing in your event handler:

**index.html:**

<form id="login-form" action="/login/" method="POST">

...

</form>

**script.js:**

function handleSubmit(event) {

event.preventDefault();

... // Everything else you want to do

loginForm.submit();

}

let loginForm = document.getElementById('login-form');

loginForm.addEventListener('submit', handleSubmit);

By combining these three concepts together - capturing the submit event, preventing the default action, and submitting the form yourself with JavaScript - you can successfully intercept any form submission and use it to execute your JavaScript code.

Graphical user interface, application

Description automatically generatedAn important distinction when working with forms is whether the data is submitted via a GET or a POST request. This isn't just relevant to JavaScript either - it's actually part of the HTTP request/response cycle itself. The key difference is that a GET request submits the form data through the URL as URL parameters while a POST request submits it in the data component of the HTTP request itself. In practicality, this means that form data submitted via a GET request will always be visible to the user in their URL while POST data will remain hidden. This is an important distinction to make, especially when it comes to working with sensitive form data.

The determination of whether to use GET or POST when submitting a form is made using the method attribute of the form element in the HTML. A form with method="POST" will keep the data hidden while a form using method="GET" will encode it as URL parameters and submit it via the URL. For forms that don't submit anything at all (for example those that have their default action prevented and are only used as a means of modifying the DOM), the method attribute can be left off completely, but it's important to know that if you don't specify it, it will default to GET which could expose sensitive data if you forget to use event.preventDefault().

To illustrate, here is a form submitted with the POST method:

<form method="POST" action="/login/">

...

</form>

And the equivalent form using the GET method:

<form method="GET" action="/login/">

...

</form>

In general you should prefer to submit form data using POST, but using GET is sometimes required (for example in order to generate sharable URLs which have captured form data, e.g. https://www.example.com/search?query=thing). In this example, the user could copy the entire URL, including their search for "thing" and send it to a friend. Using POST would not allow this because the query would be hidden in the POST request. The opposite requirement would be a login form: as you would not want a username or password to be submitted as URL parameters, because these would then show up in the browser history or be shared publicly.

The distinction between GET and POST will become more important once you need to *make* GET and POST requests with your JavaScript and access that data on a back end server.

Graphical user interface, text, application

Description automatically generated

The final step in being able to effectively work with forms in JavaScript is ensuring that the data being submitted is valid. This means verifying that all the required fields are filled out, all fields contain the correct type of data and all that data is formatted correctly, among many other things. You've undoubtedly seen form validation in action in the real world. Any time you're filling out an online form and see a message like "This field is required" or "Please enter a valid email address", you're seeing form validation at work. To do this validation yourself you can use a combination of HTML and JavaScript.

Form validation is important for three reasons:

1. It verifies that our applications get the correct data in the correct format so that they can function a expected.
2. It verifies that user data is safe and legitimate by forcing users to enter secure passwords, valid emails and so on.
3. It verifies that our own applications are safe from malicious users that might misuse forms to damage the application or steal data.

The simplest type of form validation is accomplished using HTML attributes such as those in the following table. These attributes are applied on the HTML input elements in the form and validate that the data is within the constraints specified by the attribute value.

| **Attribute** | **Description** |
| --- | --- |
| max | Specifies the maximum value of an input element |
| min | Specifies the minimum value of an input element |
| pattern | Specifies the value pattern of an input element |
| required | Specifies that the input field requires an input |
| type | Specifies the type of an input element |
| disabled | Specifies that the input element should be disabled |

Here is an example of an HTML form that utilizes all of these attributes except disabled:

<form id="registration-form" action="/register/" method="POST">

<div class="username">

<label for="username">Username</label>

<input name="username" id="username" type="text" required>

</div>

<div class="email">

<label for="email">Email</label>

<input name="email" id="email" type="email" required>

</div>

<div class="password">

<label for="password">Password</label>

<input name="password" id="password" type="password" required>

</div>

<div class="confirm-password">

<label for="confirm-password">Confirm Password</label>

<input name="confirm-password" id="confirm-password" type="password" required>

</div>

<div class="age">

<label for="age">Age</label>

<input name="age" id="age" type="number" min="18" max="110" required>

</div>

<div class="phone">

<label for="phone">Phone (numbers only)</label>

<input name="phone" id="phone" type="text" pattern="[0-9]+" required>

</div>

<div class="join-newsletter">

<label for="join-newsletter">Join Newsletter</label>

<input name="join-newsletter" id="join-newsletter" type="checkbox" checked>

</div>

<div id="errors"></div>

<div class="submit">

<input type="submit" value="Register">

</div>

</form>

This form is almost completely validated with HTML. The email address format is validated by using type="email", the age is verified by using a min and max value on the number input, the phone number is verified with the pattern attribute which ensures it has no characters other than the numbers 0-9 and that it has at least one number in it (the + means "match between one and unlimited times"), and all required fields are marked accordingly so any field left blank will cause validation to fail. Any field that doesn't pass the validation will raise a nice little notification for the user before submission is allowed, and this is built right into HTML5 so there's nothing you need to do to take advantage of it. There is one subtle piece that HTML cannot validate though: whether the two password fields match. This is where JavaScript comes in and where you can combine everything you've learned about getting DOM elements, working with their values and handling form submission with event listeners:

function handleSubmit(event) {

event.preventDefault();

let p1 = form.elements['password'].value;

let p2 = form.elements['confirm-password'].value;

if (p1 !== p2) {

let errorDiv = document.getElementById('errors');

errorDiv.innerHTML = "<p>Please ensure your passwords match.</p>"

} else {

console.log('Validation successful!');

form.submit();

}

}

let form = document.getElementById('registration-form');

form.addEventListener('submit', handleSubmit);

The above snippet is pretty straightforward. We've written a function that is attached to the form as an event listener and it's listening for the submit event. The submit handler first prevents the default action of the form using event.preventDefault();, then verifies that the two passwords match by getting their values from the form and comparing them, and finally submitting only if they match. Otherwise, it places an error message in the errors div just above the submit button.

Diagram

Description automatically generated

Of all the things you've learned in this module, arguably the most important is the concept of data structures. As a developer you will live and breathe data of all sorts. In the context of JavaScript, the DOM is no different than any other data structure you've worked with so far. You probably recall that the DOM is considered a tree where everything stems from the trunk (the html element), and its two major branches, head and body, give way to everything else in the DOM. The entire DOM is a series of familial, hierearchical relationships.

Like the DOM, a JavaScript object, denoted by opening and closing curly braces, is a series of familiar and hierearchical relationships. The only difference is that in the context of the DOM, the children, parents, grandchildren and so on are HTML elements or text nodes. In a JavaScript data structure, they are properties, arrays, functions other objects and so on. They're all nested within each other, just like the DOM. This concept is critical to understand.

**The DOM is literally a data structure**

Don't believe it? Go to any website, even the one you're looking at right now, right click and open Chrome dev tools, and paste in this code:

let html = document.getElementsByTagName('html');

let children = html[0].children;

for (child of children) {

console.log(child.tagName);

}

You'll see that the output is both the HEAD and BODY elements. The html element itself, when you use the document object to get it by its tag name, is a literal JavaScript object which you can iterate and access properties of just like you did in the data structures lesson. It's equivalent - albeit this is simplified - to building an object like this:

let html = [{

children: [

{ tagName: 'HEAD' },

{ tagName: 'BODY' },

]

}];

console.log(html[0].children[1].tagName);

If you paste the above code into the console it will work exactly the same as getting the real HTML element (except that the one we've created here won't actually have all the real HTML content, of course). The structure, though, is exactly the same. html is an array of objects, the first of which is the object containing a property called children. That property itself is an array of objects, each with a tagName property.

In the real DOM, these objects are the HTML elements with all their properties such as height, width, attributes, their methods and functions, and eveything else you've worked with in this unit. The point is that if you know how to navigate, iterate and manipulate these data structures, you know how to navigate, iterate and manipulate the DOM!

Graphical user interface, application

Description automatically generated

The concept of the DOM being a data structure gives way to the realization that it's possible to iterate through almost anything in the DOM, right down to text nodes. One common way this is used is in creating, manipulating or removing large numbers of repetitive DOM elements in a loop. This is really useful when you want to use JavaScript to create tables and lists in HTML.

To illustrate the concept of iterating DOM elements, consider that you want to create a table in your HTML. The table might have 1000 rows, so it would be very tedious and time consuming to create all that HTML manually! Instead, you can use JavaScript with a simple loop, and an array of the items you want to put in the table. For illlustration purposes, let's just say it's a table of 1000 random numbers between 1 and 1000, but it could really be anything you wish. Also note that there are more efficient and less verbose ways to do this, but it's shown step by step here to solidify the concept.

To understand how this is possible, just think about the things you need:

1. You need to somehow create 1000 random numbers (perhaps Math.random() may be of use here?)
2. You need to create the header row of the table; we'll say it has two columns: **Item #**, and **Value** (the actual number)
3. Inside the body of the table, you need to create 1000 rows, each with two columns (the item number and its value from the array). This can be accomplished with a simple for or while loop.

Here's a runnable example that puts this into practice and actually generates the table in question. Before reviewing this example, you may wish to revisit the [Template Literals](https://learn.codeinstitute.net/courses/course-v1:CodeInstitute+LMR101+2021_T1/jump_to_id/37e655f4ce2340cfb2760b4def0e7238) unit to review the usage of the `${}` syntax:

Graphical user interface, text, application, email

Description automatically generated

In the last unit you saw how it's possible to use JavaScript to build HTML using iteration. You can do it the other way around also! At many points in your development career you'll certainly find yourself needing to loop through all the rows in an HTML table, all the elements of a list, or all the elements of a page altogether. For these times you must know how to organize DOM elements into JavaScript data structures so you can use them in your code, pass them to back end servers and accurately manipulate them.

For this concept you will take a similar approach as the last unit, but this time you'll do it in reverse. Consider that you're writing a JavaScript application that scrapes information off a public website and uses it to do something personal. Maybe you want to aggregate all the recipes from your favorite recipe site into a cookbook as a gift, or you want to collect all the latest headlines from various news sites and determine how many times certain words appear in the headlines. To do this you need to organize the data into a predictable structure like a JavaScript object. Here's a table of people along with their ages and favorite foods:

<table>

<thead>

<tr>

<th>Name</th>

<th>Age</th>

<th>Favorite Food</th>

</tr>

</thead>

<tbody>

<tr>

<td>John</td>

<td>32</td>

<td>Pizza</td>

</tr>

<tr>

<td>Mary</td>

<td>25</td>

<td>Tacos</td>

</tr>

<tr>

<td>Mike</td>

<td>28</td>

<td>Spaghetti</td>

</tr>

<tr>

<td>April</td>

<td>30</td>

<td>Steak</td>

</tr>

</tbody>

</table>

In HTML this clearly has a hierarchical structure to it: The table is the parent of the thead and tbody. The thead and tbody are siblings, and parents of their respective row children. Each tr is also a sibling and has children which are either td or th elements. If you want to collect all the table data, row by row, you can iterate through the table body to do it:

let tbody = document.getElementsByTagName('tbody')[0];

let rows = tbody.children;

for (let row of rows) {

// Do something for each row

}

The "something" you need to do for each row is the question. The answer isn't too complex though: basically you just need to loop through each row's children (the td elements) and put their text content into a data structure. Since you're working with named values (name, age, favorite food), a JavaScript object is a good choice, but that will only hold one row of data. To hold all the rows, we need an array of objects...no problem!

let tbody = document.getElementsByTagName('tbody')[0];

let rows = tbody.children;

let people = [];

for (let row of rows) {

let person = {};

let cells = row.children;

person.name = cells[0].textContent;

person.age = cells[1].textContent;

person.favoriteFood = cells[2].textContent;

people.push(person);

}

Above, since we know the structure of the table in the HTML, it's easy to create JS properties like name, age and favoriteFood on the person object. We also know their locations in the table: the first cell is name, 2nd is age and 3rd is their favorite food. If there were many columns, it would be trivial to also iterate through the thead element's first row and create a property for each of its children as well, but since there are only three columns here its easier to just do it manually. The point is that with a little iteration you've converted an HTML table into a real JavaScript object and now you can do anything you want with it! The concept of being able to convert an HTML entity into a JS object and vice versa is useful in more ways than you can imagine, and will be critical to your understanding of using JavaScript to manipulate the DOM and add interactivity to your webpages.

Graphical user interface, text, application

Description automatically generated

You now know how to integrate many different JavaScript concepts with the DOM, from creating elements to setting their attributes, iterating through data structures to render DOM elements and vice versa, and you have a multitude of tools in your toolbox for working in JavaScript in general like conditional statements, knowledge of the different data types and how to convert between them and so on. At this point, you have most of the tools you need to render pretty much anything in HTML, but sometimes it's not prudent to do it step by step and it's best to write the structure of the HTML as a string and just use JavaScript to keep appending to the HTML string until it's complete, then set the document or the body's innerHTML property to your long HTML string.

To do this, you will need to revisit and become comfortable with the concept of template literals. Luckily this is a simple concept. The only difference between a template literal and a string is the use of backticks instead of single or double quotes. Anything that comes from JavaScript, such as variables, get injected into the string using the $ operator and a set of curly braces, like this:

let myVariable = 123;

let myLiteral = `The value of my variable is ${myVariable}.`

The above will render "The value of my variable is 123." when evaluated. You can use this concept to write complex HTML structures like tables, lists, divs filled with lots of images or anything else you can imagine, by combining the concept of a template literal with the concept of iteration, data structures and JavaScript variables. The way developers usually tackle this is to begin with an empty string and then append HTML to it step by step in a loop or using some other JavaScript logic. Here's an example:

let fruits = [

{

name: 'Apple',

image: 'images/apple.png',

},

{

name: 'Orange',

image: 'images/orange.png',

},

{

name: 'Banana',

image: 'images/banana.png',

},

];

let html = `

<table>

<thead>

<tr>

<th>Fruit</th>

<th>Image</th>

</tr>

</thead>

<tbody>

`;

for (fruit of fruits) {

let rowHtml = `

<tr class="fruit-row">

<td>${fruit.name}</td>

<td><img src="${fruit.image}" /></td>

</tr>

`;

html += rowHtml;

}

html += `

</tbody>

</table>

`;

document.body.innerHTML = html;

The process is straightforward. In the above example you begin with defining an array of objects, each object being a fruit with a name and an image url. This object is arbitrary here, but a more realistic scenario might have it coming from an API or somewhere else in your application. Either way, the intention is to create a table (or list, series of divs, paragraphs, or whatever else you want) where each row is a different fruit and the cells are the name of the fruit and an image of it. This is a pretty complex HTML structure that would take forever to create if you did it all step by step with things like createElement(), setAttribute(), appendChild() and so on.

Instead, above you create the first part of your HTML as a template literal (note this could be a regular string since it had no variables in it but we've made it a template literal so it's easier to read and can be multi-line) but stop at the part where things will need to be repeated. The top part of the table structure is static, but to generate the rows in the body, we stop there and go into a for loop. Inside the loop, we use a template literal to create a new row for each fruit and add it to the current html string. In the end, this just creates a really long string with all your HTML in it, and to finish it up you just need to close off the table body and the table itself, so that's added to the string after the loop finishes. Finally, the whole string is set to replace the document body's innerHTML. If you needed to inject this into an existing page without replacing the whole body, you could simply use createElement('div') to create a div for the table to live in, set the div's innerHTML, and use appendChild(), insertBefore or one of the other methods you've learned to insert the div into the DOM in the correct place.