JS

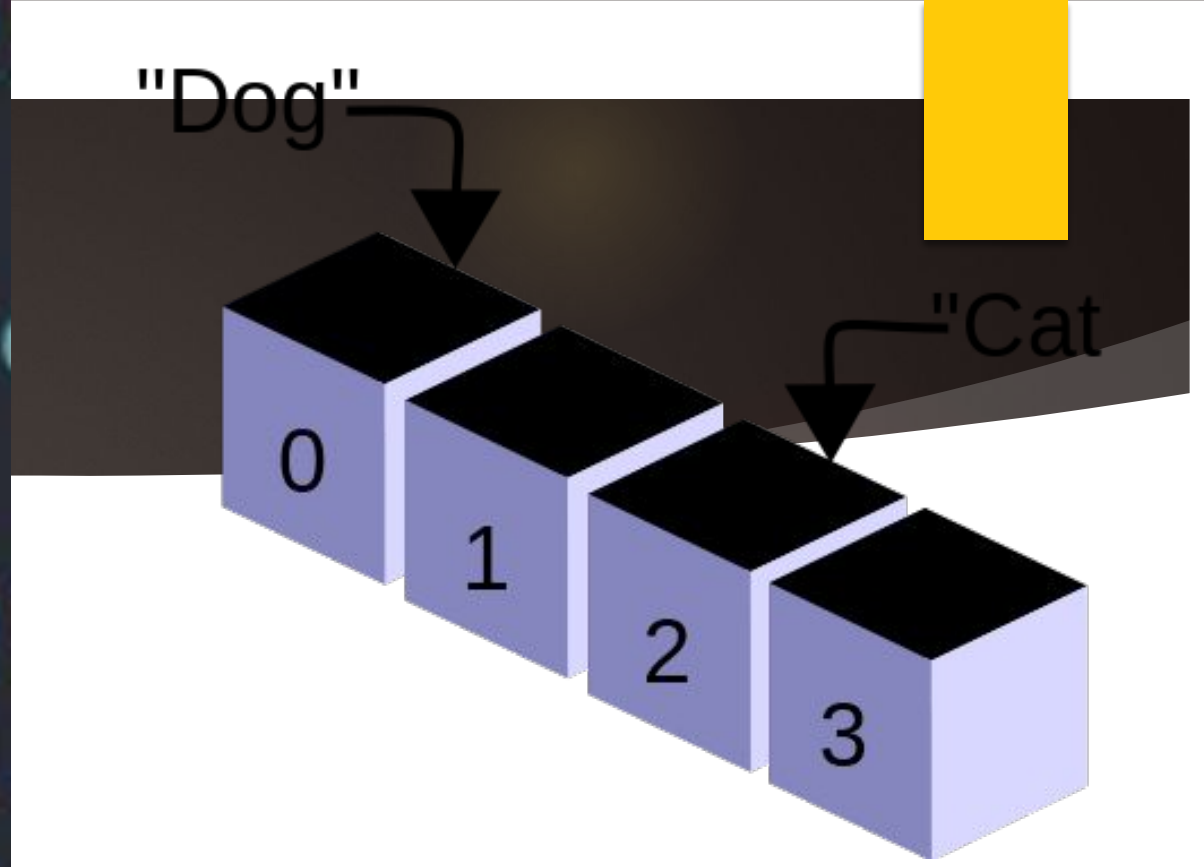
Focus points:

- Arrays
- Loops
- Document Object Model
- Events

```
ar array1 = Array(5

onsole.log(array1.l

onsole.log(array1[0
onsole.log(array1[1
onsole.log(array1[2
onsole.log(array1[3
onsole.log(array1[4
```



Array

JAVASCRIPT

Array

The Array object, as with arrays in other programming languages, enables storing a collection of multiple items under a single variable name, and has members for performing common array operations.

In JavaScript, arrays aren't **primitives** but are instead **Array objects** with the following core characteristics:

- JavaScript arrays are resizable and can contain a mix of different data types. (When those characteristics are undesirable, use typed arrays instead.)
- JavaScript arrays are not associative arrays and so, array elements cannot be accessed using strings as indexes, but must be accessed using integers as indexes.
- JavaScript arrays are zero-indexed - meaning the first element of an array is at **index 0**, the second is at **index 1**, and so on — and the last element is at the value of the array's length property minus 1.
- JavaScript array-copy operations create shallow copies. (All standard built-in copy operations with any JavaScript objects create shallow copies, rather than deep copies).

How to create an array

This example shows three ways to create new array: first using **array literal** notation, then using the **Array() constructor**, and finally using **String.prototype.split()** to build the array from a string.

```
// 'cars' array created using array literal notation.
const cars = ['Mustang', 'Golf'];
console.log(cars.length);


// 'cars' array created using the Array() constructor.
const cars = new Array('Mustang', 'Golf');
console.log(cars.length);


// 'cars' array created using String.prototype.split().
const cars = 'Mustang, Golf'.split(', ');
console.log(cars.length);
```

Create a string from an array

This example uses the **join()** method to create a string from the cars array. The **join()** method creates and returns a new string by concatenating all of the elements in an array (or an array-like object), separated by commas or a specified separator string. If the array has only one item, then that item will be returned without using the separator.

```
const cars = ['Mustang', 'Golf'];
const carsString = cars.join(', ');
console.log(carsString);
// "Mustang, Golf"
```

Access an array item

This example shows how to access items in the cars array by specifying the index number of their position in the array.

```
const cars = ['Mustang', 'Golf'];

// The index of an array's first element is always 0.
cars[0]; // Mustang

// The index of an array's second element is always 1.
cars[1]; // Golf

// The index of an array's last element is always one
// less than the length of the array.
cars[cars.length - 1]; // Golf

// Using a index number larger than the array's length
// returns 'undefined'.
cars[99]; // undefined
```

Find the index of an item in an array

This example uses the **indexOf()** method to find the position (index) of the string “Golf" in the cars array. The **indexOf()** method returns the **first index** at which a given element can be found in the array, or **-1** if it is not present.

```
const cars = ['Mustang', 'Golf', 'Mustang'];
console.log(cars.indexOf('Golf'));
// 1

console.log(cars.indexOf('Mustang'));
// 0

console.log(cars.indexOf('Picasso'));
// -1
```

Check if an array contains a certain item

This example shows two ways to check if the cars array contains “Golf" and “Picasso": first with the **includes()** method, and then with the **indexOf()** method to test for an index value that's not **-1**. The **includes()** method determines whether an array includes a certain value among its entries, returning true or false as appropriate.

```
const cars = ['Mustang', 'Golf'];

cars.includes('Golf'); // true
cars.includes('Picasso'); // false

// If indexOf() doesn't return -1, the array contains the given item.
cars.indexOf('Golf') !== -1; // true
cars.indexOf('Picasso') !== -1; // false
```

Append an item to an array

This example uses the **push()** method to append a new string to the cars array. The **push()** method adds one or more elements to the end of an array and returns the new length of the array.

```
const cars = ['Mustang', 'Golf'];
const newLength = cars.push('Picasso');
console.log(cars);
// ["Mustang", "Golf", "Picasso"]
console.log(newLength);
// 3
```

Remove the last item from an array

This example uses the **pop()** method to remove the last item from the cars array. The **pop()** method removes the **last** element from an array and returns that element. This method changes the length of the array.

```
const cars = ['Mustang', 'Golf', 'Picasso'];
const removedItem = cars.pop();
console.log(cars);
// ["Mustang", "Golf"]
console.log(removedItem);
// Picasso
```

Truncate an array down to just its first N items

This example uses the **splice()** method to truncate the cars array down to just its first 2 items. The **splice()** method changes the contents of an array by removing or replacing existing elements and/or adding new elements in place.

```
const cars = ['Mustang', 'Golf', 'Picasso', 'Multipla', 'Veyron'];
const start = 2;
const removedItems = cars.splice(start);
console.log(cars);
// ["Mustang", "Golf"]
console.log(removedItems);
// ["Picasso", "Multipla", "Veyron"]
```

Remove multiple items from the end of an array

This example uses the **splice()** method to remove the last 3 items from the cars array. If parameter is negative, it will remove that many elements from the end of the array.

```
const cars = ['Mustang', 'Golf', 'Picasso', 'Multipla', 'Veyron'];
const start = -3;
const removedItems = cars.splice(start);
console.log(cars);
// ["Mustang", "Golf"]
console.log(removedItems);
// ["Picasso", "Multipla", "Veyron"]
```

Remove multiple items from the beginning of an array

This example uses the **splice()** method to remove the first 3 items from the cars array.

```
const cars = ['Mustang', 'Golf', 'Picasso', 'Multipla', 'Veyron'];
const start = 0;
const deleteCount = 3;
const removedItems = cars.splice(start, deleteCount);
console.log(cars);
// ["Multipla", "Veyron"]
console.log(removedItems);
// ["Mustang", "Golf", "Picasso"]
```

Remove the first item from an array

This example uses the **shift()** method to remove the first item from the cars array. The **shift()** method removes the first element from an array and returns that removed element. This method changes the length of the array.

```
const cars = ['Mustang', 'Golf'];
const removedItem = cars.shift();
console.log(cars);
// ["Golf"]
console.log(removedItem);
// Mustang
```

Add a new first item to an array

This example uses the **unshift()** method to add, at index 0, a new item to the cars array — making it the new first item in the array. The **unshift()** method adds one or more elements to the beginning of an array and returns the new length of the array.

```
const cars = ['Golf', 'Picasso'];
const newLength = cars.unshift('Mustang');
console.log(cars);
// ["Mustang", "Golf", "Picasso"]
console.log(newLength);
// 3
```

Remove a single item by index

This example uses the **splice()** method to remove the string “Golf" from the cars array — by specifying the index position of “Golf".

```
const cars = ['Mustang', 'Golf', 'Picasso'];
const start = cars.indexOf('Golf');
const deleteCount = 1;
const removedItems = cars.splice(start, deleteCount);
console.log(cars);
// ["Mustang", "Picasso"]
console.log(removedItems);
// ["Golf"]
```

Remove multiple items by index

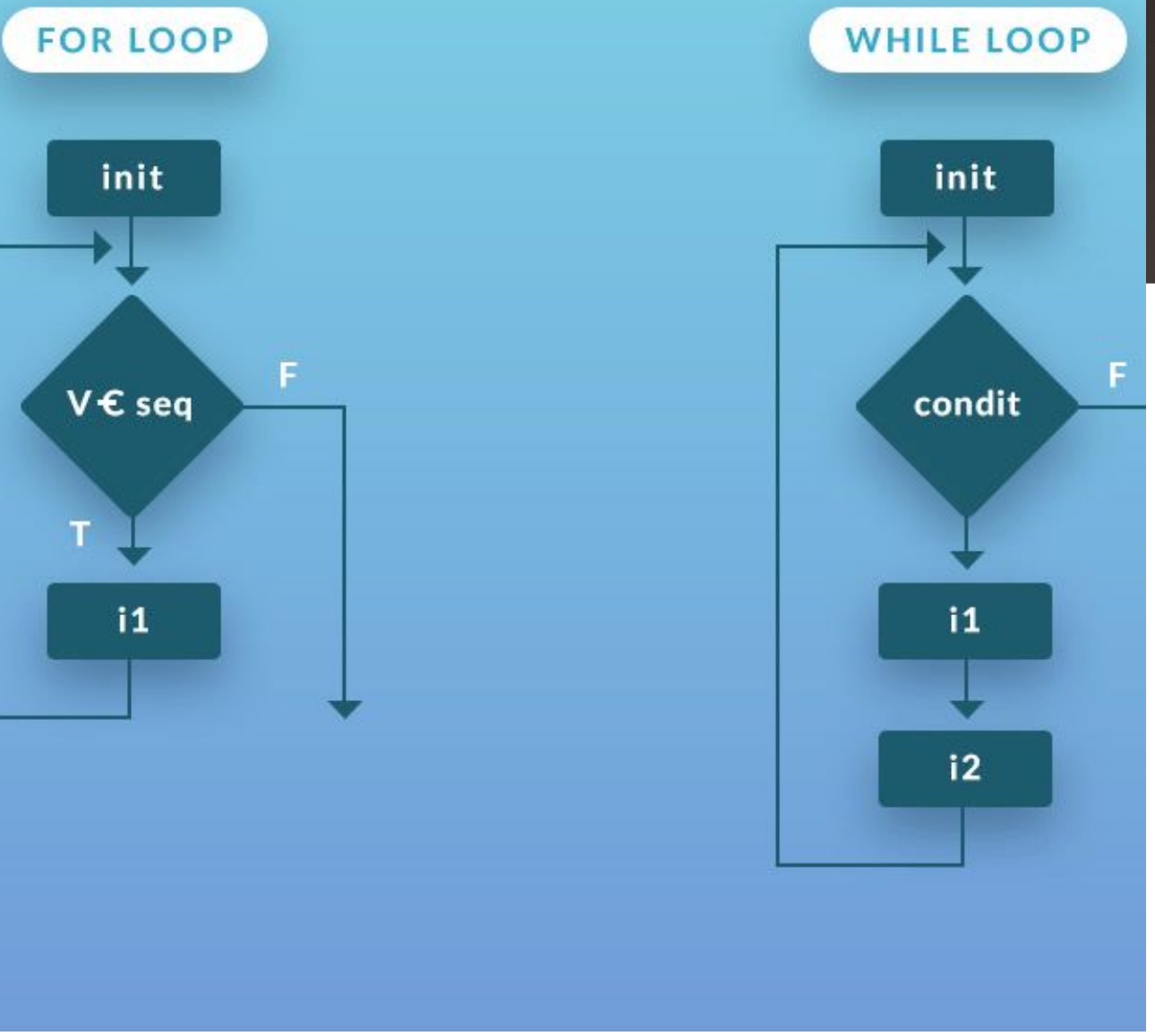
This example uses the **splice()** method to remove the strings “Golf" and “Picasso" from the cars array — by specifying the index position of “Golf", along with a count of the number of total items to remove.

```
const cars = ['Mustang', 'Golf', 'Picasso', 'Multipla'];
const start = 1;
const deleteCount = 2;
const removedItems = cars.splice(start, deleteCount);
console.log(cars);
// ["Mustang", "Multipla"]
console.log(removedItems);
// ["Golf", "Picasso"]
```

Replace multiple items in an array

This example uses the **splice()** method to replace the last 2 items in the cars array with new items.

```
const cars = ['Mustang', 'Golf', 'Picasso'];
const start = -2;
const deleteCount = 2;
const removedItems = cars.splice(start, deleteCount, 'Multipla', 'Veyron');
console.log(cars);
// ["Mustang", "Multipla", "Veyron"]
console.log(removedItems);
// ["Golf", "Picasso"]
```



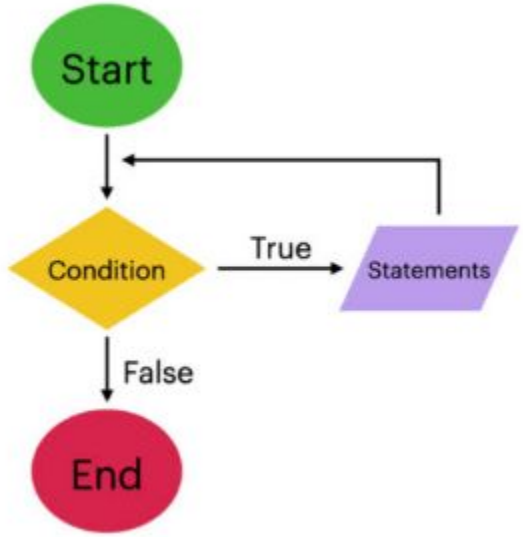

```
for (let i = 1; i < 4; i++) {
  for (let k = 1; k < 4; k+
    console.log(i + " , " +
  }
  console.log("\n");
}
```

Loops

JAVASCRIPT

Loops

In computer programming, a **loop** is a sequence of instructions that is continually repeated until a certain condition is reached. Typically, a certain process is done, such as getting an item of data and changing it, and then some condition is checked such as whether a counter has reached a prescribed number. In one sentence, loops are a way to repeat the same code multiple times.



While

The **while** loop has the following syntax:

```
while (condition) {
  // code
  // so-called "loop body"
}
```

While the condition is **truthy**, the code from the loop body is executed.
For instance, the loop below outputs **i** while **i < 3**:

```
let i = 0;
while (i < 3) {
  // shows 0, then 1, then 2
  alert(i);
  i++;
}
```

A single execution of the loop body is called an **iteration**. The loop in the example makes three iterations.

If i++ was missing from the example, the loop would repeat (in theory) forever.

While

Any expression or variable can be a loop condition, not just comparisons: the condition is evaluated and converted to a *boolean* by while. For instance, a shorter way to write **while (i != 0)** is **while (i)**:

```
let i = 3;
while (i) {
  // when i becomes 0
  // the condition becomes falsy
  // and the loop stops
  alert( i );
  i--;
}
```

```
let i = 3;
while (i) alert(i--);
```

Note: If the loop body has a single statement, we can omit the curly braces {…}

Do while

The **do while** loop has the following syntax:

```
do {
  // loop body
} while (condition);
```

The loop will first execute the body, then check the condition, and, while it's truthy, execute it again and again. This form of syntax should only be used when you want the body of the loop to execute **at least once** regardless of the condition being truthy.

```
let i = 0;
do {
  alert( i );
  i++;
} while (i < 3);
```

For loop

The **for** loop is more complex, but it's also the most commonly used loop. Here's its syntax:

```
for (begin; condition; step) {
  // ... loop body ...
}
```

Let's learn the meaning of these parts by example. The loop below runs **alert(i)** for i from **0** up to (but not including) **3**:

```
for (let i = 0; i < 3; i++) { // shows 0, then 1, then 2
  alert(i);
}
```

For loop

The general loop algorithm works like this:

1. Run begin
2. (if condition → run body and run step)
3. (if condition → run body and run step)
4. (if condition → run body and run step)
5. ...

That is, begin executes once, and then it iterates: after each condition test, body and step are executed.

begin	Executes once upon entering the loop
condition	Checked before every loop iteration. If false, the loop stops.
body	Runs again and again while the condition is truthy.
step	Executes after the body on each iteration.

Breaking the loop

Normally, a loop exits when its condition becomes **falsy**. But we can force the exit at any time using the special **break** directive. For example, the loop below asks the user for a series of numbers, “breaking” when no number is entered:

```
let sum = 0;

while (true) {

  let value = +prompt("Enter a number", '');

  if (!value) break; // (*)

  sum += value;

}
alert( 'Sum: ' + sum );
```

The break directive is activated at the commented line **(*)** if the user enters an empty line or cancels the input. It stops the loop immediately, passing control to the first line after the loop. Namely, alert.

The combination “infinite loop + break as needed” is great for situations when a loop’s condition must be checked not in the beginning or end of the loop, but in the middle or even in several places of its body.

Continue to next iteration

The **continue** directive is a “lighter version” of break. It doesn’t stop the whole loop. Instead, it stops the current iteration and forces the loop to start a new one (if the condition allows). We can use it if we’re done with the current iteration and would like to move on to the next one. The loop below uses continue to output only odd values:

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

  // if true, skip the remaining part of the body
  if (i % 2 == 0) continue;

  alert(i); // 1, then 3, 5, 7, 9
}
```

For even values of i, the continue directive stops executing the body and passes control to the next iteration of for (with the next number). So the alert is only called for odd values.

Labels for break/continue

Sometimes we need to break out from multiple nested loops at once. For example, in the code below we loop over i and j, prompting for the coordinates (i, j) from (0,0) to (2,2):

```
for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    let input = prompt(`Value at coords (${i},${j})`, '');
    // what if we want to exit from here to Done (below)?
  }
}

alert('Done!');
```

We need a way to stop the process if the user cancels the input. The ordinary break after input would only break the inner loop. That’s not sufficient – labels, come to the rescue!

Labels for break/continue

A label is an identifier with a colon before a loop:

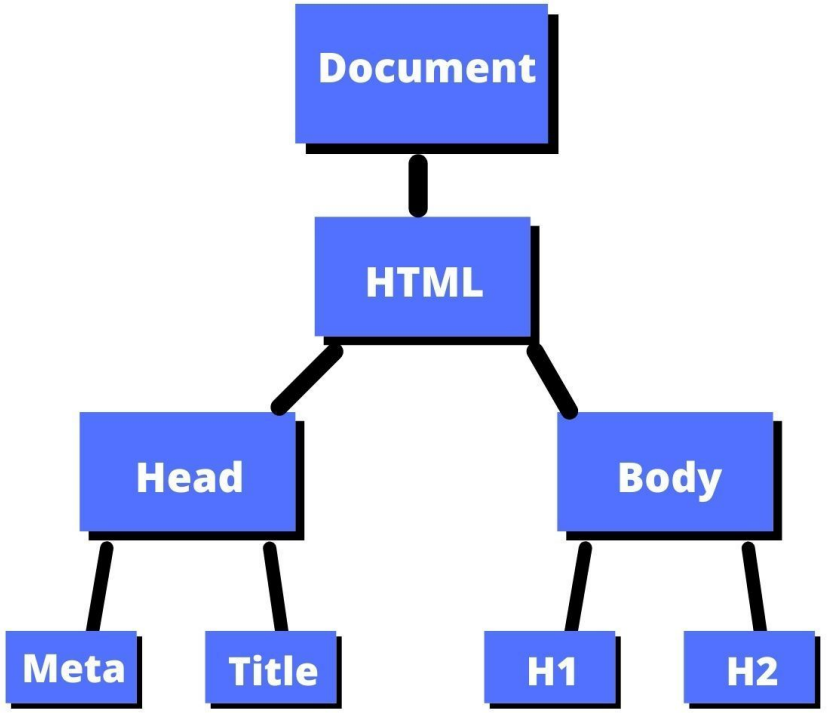
```
labelName: for (...) {
  ...
}
```

The break <labelName> statement in the loop below breaks out to the label:

```
outer: for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    let input = prompt(`Value at coords (${i},${j})`, '');

    // if an empty string or canceled, then break out of both loops
    if (!input) break outer; // (*)
  }
}
alert('Done!');
```

```
cument.querySelectorAl
utton) => {
istener('click', () =>
d);
```



DOM

JAVASCRIPT

DOM - Document Object Model

The DOM (or Document Object Model) is a tree-like representation of the contents of a webpage - a tree of “nodes” with different relationships depending on how they’re arranged in the HTML document.

```
<div id="container">
  <div class="display"></div>
  <div class="controls"></div>
</div>
```

In the above example, the ** ** is a “child” of ** ** and a sibling to ** **. Think of it like a family tree. ** ** is a parent, with its children on the next level, each on their own “branch”.

DOM - Document Object Model

When working with the DOM, you use “selectors” to target the nodes you want to work with. You can use a combination of CSS-style selectors and relationship properties to target the nodes you want. Let’s start with CSS-style selectors. In the example, you could use the following selectors to refer to ** **:

```
<div id="container">
  <div class="display"></div>
  <div class="controls"></div>
</div>
```

- div.display
- .display
- #container > .display
- div#container > div.display

DOM - Document Object Model

You can also use relational selectors **(i.e. firstElementChild or lastElementChild etc.)** with special properties owned by the nodes. So you're identifying a certain node based on its relationships to the nodes around it.

```
const container = document.querySelector('#container');
// selects the #container div (don't worry about the syntax, we'll get there)

console.dir(container.firstElementChild);
// selects the first child of #container => .display

const controls = document.querySelector('.controls');
// selects the .controls div

console.dir(controls.previousElementSibling);
// selects the prior sibling => .display
```

DOM methods

When your HTML code is parsed by a web browser, it is converted to the DOM as was mentioned above. One of the primary differences is that these nodes are objects that have many properties and methods attached to them. These properties and methods are the primary tools we are going to use to manipulate our webpage with JavaScript. We'll start with the query selectors - those that help you target nodes.

Query Selectors:

- **element.querySelector(selector)** returns a reference to the first match of selector
- **element.querySelectorAll(selectors)** returns a "nodelist" containing references to all of the matches of the selectors

**There are several other, more specific queries, that offer potential (marginal) performance benefits, but we won't be going over them now.*

Note: when using **querySelectorAll**, the return value is not an array. It looks like an array, and it somewhat acts like an array, but it's really a "nodelist". The big distinction is that several array methods are missing from nodelists. One solution, if problems arise, is to convert the nodelist into an array. You can do this with **Array.from()** or the spread operator.

Element creation

document.createElement(tagName, [options]) creates a new element of tag type **tagName**. [options] in this case means you can add some optional parameters to the function. This function does **NOT** put your new element into the DOM - it simply creates it in memory. This is so that you can manipulate the element (by adding styles, classes, ids, text, etc.) before placing it on the page.

Query Selectors:

- **element.querySelector(selector)** returns a reference to the first match of selector
- **element.querySelectorAll(selectors)** returns a "nodelist" containing references to all of the matches of the selectors

**There are several other, more specific queries, that offer potential (marginal) performance benefits, but we won't be going over them now.*

Note: when using **querySelectorAll**, the return value is not an array. It looks like an array, and it somewhat acts like an array, but it's really a "nodelist". The big distinction is that several array methods are missing from nodelists. One solution, if problems arise, is to convert the nodelist into an array. You can do this with **Array.from()** or the spread operator.

Element creation

You can place the element into the DOM with one of the following methods:

- parentNode.appendChild(childNode) appends childNode as the last child of parentNode
- parentNode.insertBefore(newNode, referenceNode) inserts newNode into parentNode before referenceNode

You can remove the element from the DOM with the following method:

- parentNode.removeChild(child) removes child from parentNode on the DOM and returns a reference to child

Altering elements

When you have a reference to an element, you can use that reference to alter the element's own properties. This allows you to do many useful alterations, like adding/removing and altering attributes, changing classes, adding inline style information and more.

```
const div = document.createElement('div');
// creates a new div referenced in the variable 'div'
```

How to add inline style:

```
div.style.color = 'blue';
// adds the indicated style rule

div.style.cssText = 'color: blue; background: white;';
// adds several style rules

div.setAttribute('style', 'color: blue; background: white;');
// adds several style rules
```

Altering elements

Note that if you're accessing a *kebab-cased* CSS rule from JS, you'll either need to use camelCase or you'll need to use bracket notation instead of dot notation.

```
div.style.background-color // doesn't work - attempts to subtract color from div.style.background
div.style.backgroundColor // accesses the div's background-color style
div.style['background-color'] // also works
div.style.cssText = "background-color: white;" // ok in a string
```

How to edit attributes:

```
div.setAttribute('id', 'theDiv');
// if id exists, update it to 'theDiv', else create an id with value "theDiv"
div.getAttribute('id');
// returns value of specified attribute, in this case "theDiv"
div.removeAttribute('id');
// removes specified attribute
```

Working with classes

It is often standard (and cleaner) to toggle a CSS style rather than adding and removing inline CSS.

```
div.classList.add('new');
// adds class "new" to your new div

div.classList.remove('new');
// removes "new" class from div

div.classList.toggle('active');
// if div doesn't have class "active" then add it, or if
// it does, then remove it
```

Adding text and html content

Here's how you would add text content:

```
div.textContent = 'Hello World!'
// creates a text node containing "Hello World!"
// inserts it in div
```

Here's how you would add HTML content:

```
div.innerHTML = '<span>Hello World!</span>';
// renders the HTML inside div
```

Note: that **textContent** is preferable for adding text, and innerHTML should be used sparingly as it can create security risks if misused. Check out this video if you want to see an example of how.

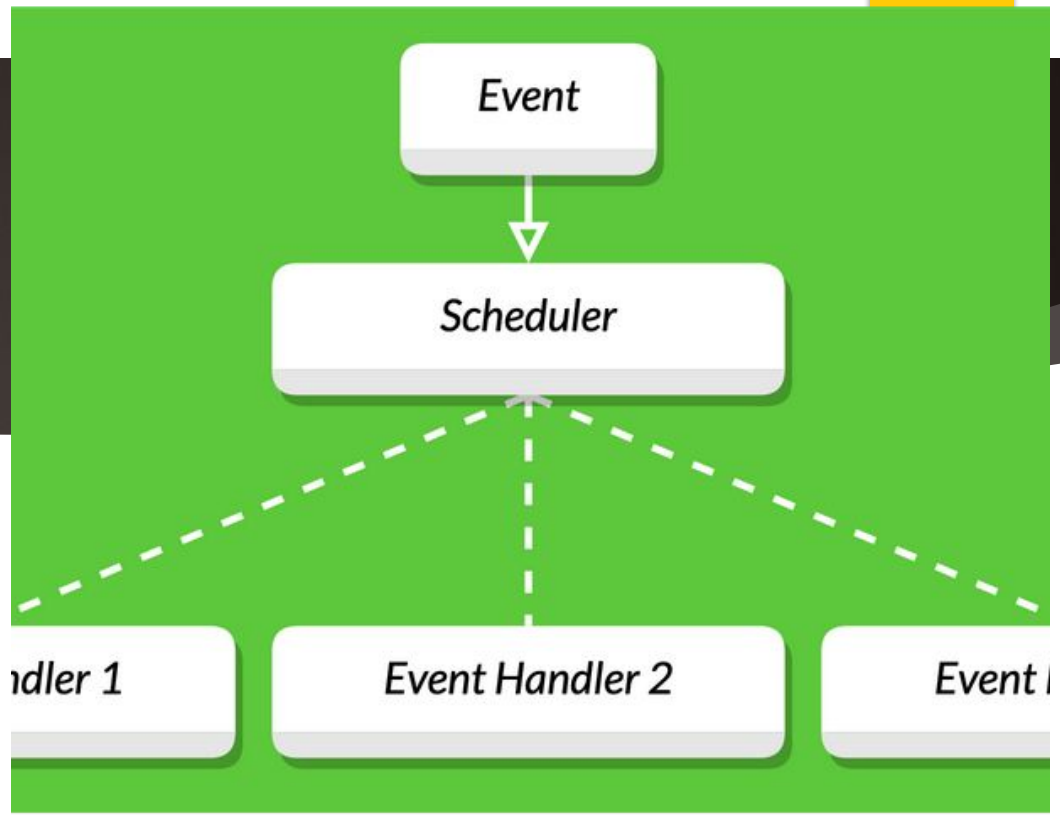
Exercise

Add the following elements to the container using **ONLY** JavaScript and the DOM methods shown previously.

1. a **<p>** with red text that says, **“Hey I’m red!”**
2. an **<h3>** with blue text that says, **“I’m a blue h3!”**
3. a ** ** with a **black** border and **pink** background color with the following elements inside of it:
   a) another **<h1>** that says **“I’m in a div”**
   b) a **<p>** that says, **“ME TOO!”**
   c) ***Hint for this one:*** after creating the with createElement, append the <h1> and <p> to it before adding it to the container.

```
tFunction() {
 YOU DID IT!");

 alertFunction;
istener('click', a
```



Events

JAVASCRIPT

Events

Events are actions or occurrences that happen in the system you are programming, which the system tells you about so your code can react to them.

As mentioned above, events are actions or occurrences that happen in the system you are programming — the system produces (or "fires") a signal of some kind when an event occurs and provides a mechanism by which an action can be automatically taken (that is, some code running) when the event occurs. For example, in an airport, when the runway is clear for take off, a signal is communicated to the pilot. As a result, the plane can safely take off.

Events

In the case of the Web, events are fired inside the browser window, and tend to be attached to a specific item that resides in it. This might be a single element, a set of elements, the HTML document loaded in the current tab, or the entire browser window. There are many different types of events that can occur.

For example:

- The user selects, clicks, or hovers the cursor over a certain element.
- The user chooses a key on the keyboard.
- The user resizes or closes the browser window.
- A web page finishes loading.
- A form is submitted.
- A video is played, paused, or finishes.
- An error occurs.

There are a lot of events that can be fired.

Event Handler

To react to an event, you attach an event handler to it. This is a block of code (usually a JavaScript function that you as a programmer create) that runs when the event fires. When such a block of code is defined to run in response to an event, we say we are registering an event handler.

Note: Event handlers are sometimes called event listeners — they are pretty much interchangeable for our purposes, although strictly speaking, they work together. The listener listens out for the event happening, and the handler is the code that is run in response to it happening.

Note: Web events are not part of the core JavaScript language — they are defined as part of the APIs built into the browser.

Event Handler

Let's look at a simple example of what we mean here. In the following example, we have a single <button>, which when pressed, makes the background change to a random color:

```
<button>Change color</button>
```

```
const btn = document.querySelector('button');

function random(number) {
  return Math.floor(Math.random() * (number+1));
}

btn.addEventListener('click', () => {
  const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
  document.body.style.backgroundColor = rndCol;
});
```

Event Handler

We can access more information about the event by passing a parameter to the function that we are calling. For example:

```
const btn = document.querySelector('button');

btn.addEventListener('click', function (e) {
  console.log(e);
});
```

The ***e*** in that function is an object that references the event itself. Within that object you have access to many useful properties and functions such as which mouse button or key was pressed, or information about the event's target - the DOM node that was clicked.

```
const btn = document.querySelector('button');

btn.addEventListener('click', function (e) {
  e.target.style.background = 'blue';
});
```

Attaching listeners to groups of nodes

We can get a nodelist of all of the items matching a specific selector with **querySelectorAll('selector')**. In order to add a listener to each of them we simply need to iterate through the whole list like so:

```html
<div id="container">
    <button id="1">Click Me</button>
    <button id="2">Click Me</button>
    <button id="3">Click Me</button>
</div>
```

```js
// buttons is a node list. It looks and acts much like an array.
const buttons = document.querySelectorAll('button');

// we use the .forEach method to iterate through each button
buttons.forEach((button) => {

  // and for each one we add a 'click' listener
  button.addEventListener('click', () => {
    alert(button.id);
  });
});
```

Assignment

aleksa.fd.tutor@gmail.com

Thank you for your attention!