Week 03 Notes:

Object Methods: “this”

Objects created to represent entities such as users, orders, etc.

JS object:

```
let user = {
	name: “John”,
	age: 30
};
```

Actions: select something from cart, login, logout etc.

Actions are functions

JS Function:

```
user.sayHi = function() {

	alert(“Hello!”);

};

user.sayHi(); // Hello!
```

This is a FUNCTION EXPRESSION to create a function and assign it to the property user.sayHi of the object.

METHOD: a function that is a property of an object – object.doSomething()

Above we have a METHOD: sayHi of the OBJECT: user

This is Object Oriented Programming (OOP)!

Method shorthand: almost always preferred

```
user = {
	sayHi() {
		alert(“Hello”);
	}
};
```

“this” in methods:

“this” keyword used by method to access the object - it is the object “before dot”, the one to call the method. Example:

```
let user = {
        name: “John”,
        age: 30,
        sayHi(){
                // “this” is the “current object”
                alert(this.name);
        }
};
user.sayHi(); // John
```

Arrow Functions have no “this”

Sample: Make a calculator with methods that read from prompts and returns a sum and returns a product

```
let calculator = {
  sum() {
    return this.a + this.b;
  },

  mul() {
    return this.a * this.b;
  },

  read() {
    this.a = +prompt('a?', 0);
    this.b = +prompt('b?', 0);
  }
};

calculator.read();
alert( calculator.sum() );
alert( calculator.mul() );
```

Chaining sample: Use a ladder object to allow you to go up and down.

The solution is to return the object itself from every call.

```
let ladder = {
  step: 0,
  up() {
    this.step++;
    return this;
  },
  down() {
    this.step--;
    return this;
  },
  showStep() {
    alert( this.step );
    return this;
  }
};

ladder.up().up().down().up().down().showStep(); // 1
```

We also can write a single call per line. For long chains it’s more readable:

```
ladder
  .up()
  .up()
  .down()
  .up()
  .down()
  .showStep(); // 1
```

Objects: self-contained set of related values and functions. Collection of named properties that mat to any JavaScript value (strings, numbers, Booleans, arrays, functions) If property’s value is a function- It is called a METHOD. Name and value pairs. Often used to keep related info and functionality together in the same place. Example: Functions that found the perimeter and area of a square could be grouped together as methods of the same object.

Object literal example:

```
        const superman = {
    name: 'Superman',
    'real name': 'Clark Kent',
    height: 75,
    weight: 235,
    hero: true,
    villain: false,
    allies: ['Batman','Supergirl','Superboy'],
    fly() {
        return 'Up, up and away!';
    }
};
```

8*If a property's name doesn't follow naming rules (like 'real name' above), it needs to be quoted. 'real name' includes a space so it is in quotes. A better way to name might be real_name or realName.

Creating Objects:

*Objects are mutable (properties and methods can be changed even with const declaration) at any time when program is running.

Create an empty object:

const spiderman = {}; -This way is preferred- more concise
or using constructor:

const spiderman = new Object();

Shorthand method for creating objects if property key is the same as a variable name the property is assigned to:

```
const name = 'Iron Man';
const realName = 'Tony Stark';
// long way
const ironMan = { name: name, realName: realName };
// short ES6 way
const ironMan = { name, realName };
```

Accessing Properties:

Use dot notation

superman.name // 'Superman'

Another way:

superman['name'] // 'Superman' – Less common but only way to access nonstandard property and method names (like 'real name' would use concatenation and would be superman["real" + " " + "name"])

Undefined will be returned when you try to access a property that doesn't exsist.

Computed Properties: Javascript code can be placed inside square brackets and the property key will be the return value of that code.

```
const hulk = { name: 'Hulk', ['catch' + 'Phrase']: 'Hulk Smash!' };
```

```
<< { name: 'Hulk', catchPhrase: 'Hulk Smash!' }
```

Below, a ternary operator is used to return a true or false value for the hero property depending on the value of the bewitched variable:

```
const bewitched = true;
const captainBritain = { name: 'Captain Britain', hero: bewitched ? false : true };
captainBritain
<< { name: 'Captain Britain', hero: false }
```

The new Symbol date type can also be used as a computed property key:

```
const name = Symbol('name');
const supergirl = { [name]: 'Supergirl' };
```

You can access the property using the square bracket notation:

```
supergirl[name];
<< 'Supergirl'
```

A new property can be added to an object using a symbol as a key if the square bracket notation is used:

```
const realName = Symbol('real name');
supergirl[realName] = 'Kara Danvers';
<< 'Kara Danvers'
```

The symbols used for property keys are not limited to being used by only one object - they can be reused by any other object:

```
const daredevil = { [name]: 'Daredevil', [realName]: 'Matt Murdoch' };
```

Call a Method:

superman.fly() // ‘Up, up and away!’

superman.[‘fly’]() // ‘Up, up and away!’

Check if property or method exists:

‘city’ in superman;

superman.city !== undefined;

// both of these return false

superman.hasOwnProperty(‘city’); //false

superman.hasOwnProperty(‘name’); //true

Find all the properties:

Use for in loop:

```
                 for(const key in superman) {
    console.log(key + ": " + superman[key]);
}
<< "name: Superman"
<< "real name: Clark Kent"
<< "height: 75"
<< "weight: 235"
<< "hero: true"
<< "villain: false"
<< "allies: Batman,Supergirl,Superboy"
<< "fly: function (){
    console.log(\"Up, up and away!\");
}"
```

Object.keys() method will return an array of all the keys of an object that is provided as an argument. Allows us to iterate over this array to access all of the keys.

```
for(const key of Object.keys(superman)) {
    console.log(key);
}
<<  name
    real name
    height
    weight
    hero
    villain
    allies
    fly
```

Object.values() for an array of values:

```
for(const value of Object.values(superman)) {
    console.log(value);
}
<<  Superman
    Clark Kent
    75
    235
    true
    false
    [ 'Batman','Supergirl','Superboy' ]
    [Function: fly]
```

Object.entries() for key-value pairs:

```
for(const [key,value] of Object.entries(superman)) {
    console.log(`${key}: ${value}`);
}
```

```
<<  name: Superman
    real name: Clark Kent
    height: 75
    weight: 235
    hero: true
    villain: false
    allies: [ 'Batman','Supergirl','Superboy' ]
    fly: [Function: fly]
```

Adding Properties:

Can add to objects at any time in a program. Simply assign a value to the new property.

```
superman.city = 'Metropolis';
<< 'Metropolis'
```

*Properties not in a certain order like an array, set, or map.

Changing Properties:

Can change at any time using assignment:

```
        superman['real name'] = 'Kal-El';
<< 'Kal-El'
```

Removing Properties:

Can remove from object using the delete operator:

```
        delete superman.fly
<< true
```

Nested Objects:

Possible for an object to contain other objects:

```
const jla = {
    superman: { realName: 'Clark Kent' },
    batman: { realName: 'Bruce Wayne' },
    wonderWoman: { realName: 'Diana Prince" },
    flash: { realName: 'Barry Allen' },
    aquaman: { realName: 'Arthur Curry' },
}
```

Access values using dot or bracket notation – or mix the two

```
        jla.wonderWoman.realName
<< "Diana Prince"
jla['flash']['realName']
<< "Barry Allen"
jla.aquaman['realName']
```

```
<< "Arthur Curry"
```

Objects assigned by reference. Variable assigned to an object that already exists will point to the exact same object or space in memory. Any changes made to either will effect the same object

```
const thor = { name: 'Thor'
// more properties here
};
const cloneThor = thor;// This points to the exact same object thor and changes can
be made with thor or cloneThor
```

Objects as Function Parameters:

```
                function greet({greeting,name,age}) {
    return `${greeting}! My name is ${name} and I am ${age} years old.`;
}
```

```
        greet({ greeting: `What's up dude`, age: 10, name: `Bart` });
<< 'What\'s up dude! My name is Bart and I am 10 years old.'
```

This Keyword:

```
const dice = {
    sides: 6,
    roll() {
        return Math.floor(this.sides * Math.random()) + 1;
    }
}
```

Namespacing:

Possible name collisions using libraries or working in teams. Solution: use the OBJECT LITERAL PATTERN to create a namespace for groups of related functions. Create an object literal that serve as the namespace, then add values as properties of that object and any functions as methods.

Place functions inside an object- creating a namespace for them. Below the namespace is myMaths:

```
        const myMaths = {
    square(x) {
        return x * x;
    },
    mean(array,callback) {
        if (callback) {
```

```
        array.map( callback );
        }
        const total = array.reduce((a, b) => a + b);
        return total/array.length;
    }
};
```

You then use the namespace before the function when invoking:

```
myMaths.square(3)
<< 9
myStats.mean([1,2,3])
<< 2
```

Built-in objects:

JSON: JavaScript Object Notation

Lightweight data storage- both human and machine readable

String representation of the object literal notation. Differences:

Property names double quoted, permitted values are double-quoted strings, numbers, true, false, null, arrays and objects. Functions are not permitted values.

Example:

```
const batman = '{"name": "Batman","real name": "Bruce Wayne","height": 74, "weight": 210, "hero": true, "villain": false, "allies": ["Robin","Batgirl","Superman"]}'
```

parse() method: Take a string of JSON data and returns a JavaScript object:

```
JSON.parse(batman);
<< { name: 'Batman',
'real name': 'Bruce Wayne',
height: 74,
weight: 210,
hero: true,
villain: false,
allies: [ 'Robin', 'Batgirl', 'Superman' ] }
```

stringify() method does opposite, converting JavaScript object and returning a string of JSON data:

```
        const wonderWoman = {
    name: 'Wonder Woman',
    'real name': 'Diana Prince',
    height: 72,
    weight: 165,
    hero: true,
    villain: false,
    allies: ['Wonder Girl','Donna Troy','Superman'],
    lasso: function(){
        console.log('You will tell the truth!');
    }
}
JSON.stringify(wonderWoman);
<< '{"name":"Wonder Woman","real name":"Diana Prince","height":72,
"weight":165,"hero":true,"villain":false,"allies":["Wonder Girl",
"Donna Troy","Superman"]}'
```

Math Object: built in object that has several properties representing meath constants as well as methods that carry out common math operations.\

Math Constants:

Math.PI // The ratio of the circumference and diameter of a circle

<< 3.141592653589793

Math.SQRT2 // The square root of 2

<< 1.4142135623730951

Math.SQRT1_2 // The reciprocal of the square root of 2

<< 0.7071067811865476

Math.E // Euler's constant

<< 2.718281828459045

Math.LN2 // The natural logarithm of 2

<< 0.6931471805599453

Math.LN10 // The natural logarithm of 10

<< 2.302585092994046

Math.LOG2E // Log base 2 of Euler's constant

<< 1.4426950408889634

Math.LOG10E // Log base 10 of Euler's constant

<< 0.4342944819032518

Math Methods:

Math.abs() – returns absolute value of a number

Math.abs(3);

<< 3

Math.abs(-4.6);

<< 4.6

Math.ceil() -round a number up to the next integer

Math.ceil(4.2);

<< 5

Math.floor() – Round a number down to next integer

Math.round() – Round a number to nearest integer

Math.trunc() – returns the integer-part of a number

Math.exp() – raise a number to the power of Euler's constant

Math.pow() – raise a number(first arg) to the power of another (2nd arg)

Math.sqrt() – returns the positive square root of a number

Math.cbrt() – returns cube root

Math.hypot() – returns square root of the sum of the squares of all its arguments

Math.log() – Returns the natural logarithm of a number

There are also Max and Min methods, trig methods

Math.random() – used to create random number between 0 and 1

To generate a number between 0 and a chosen number (6) – multiply that value:

6 * Math.random();

<< 4.580981240354013

// use rounding methods for decimals:

(Math.floor(6 * Math.random());

<< 4

Date object: Each object represents a single moment in time

Constructor:

```
const today = new Date();
today.toString();
<< 'Tue Feb 14 2017 16:35:18 GMT+0000 (GMT)'
```

```
const christmas = new Date('2017 12 25');

christmas.toString();

<< 'Mon Dec 25 2017 00:00:00 GMT+0000 (GMT)'
```

```
new Date(year,month,day,hour,minutes,seconds,milliseconds)

//better to provide each bit of info about the date as a
separate argument:
```

const solstice = new Date(2017, 5, 21);

// Summer Solstice

solstice.toString();

<< 'Wed Jun 21 2017 00:00:00 GMT+0100 (BST)'

*Jan is 0, Feb is 1, etc.

Getter Methods:

Since properties of date objects are unable to be viewed or changed directly, there are GETTER METHODS that return info about the date object lie the month and year. Two versions – Local time and UTC time. getTime(), getTimexoneOffset(), and getYear() methods don't have UTC equivalents.

getDate(), getMonth(), getYear(), getHours(), getMinutes(), getFullYear(), etc.

Setter Methods:

setDate(), setMonth(), setFullYear() etc. is in milliseconds since Epoch

Use toString() method to see the actual date

moment.js library – methods that make it easier to work with dates

RegExp Object: Regular Expression object – a pattern that can be used to search strings for matches to the pattern.

Example: any word ending in 'ing' would use the expression: `/[a-zA-Z]+ing$/`

```
const pattern = /[a-zA-Z]+ing$/;
```

or

```
const pattern = new RegExp('[a-zA-Z]+ing');
```

Methods: test() – see if a string passed as parameter matches the regular expression pattern.

```
        pattern.test('joke');
<< false
pattern.test('joking');
<< true
pattern.test('jokingly');
<< false
```

exec() -returns an array contqining the first match found, or null if no matches

```
        pattern.exec('joke');
<< null
pattern.exec('joking');
<< [ 'joking', index: 0, input: 'joking' ]
```

Groups can be combined with letters to make a more complex pattern. For example, the following regular expression represents the letter J (lowercase or capital) followed by a vowel, followed by a lowercase v, followed by a vowel:

```
pattern = /[Jj][aeiou]v[aeiou]/;
<< /[Jj][aeiou]v[aeiou]/
pattern.test('JavaScript');
<< true
pattern.test('jive');
<< true
pattern.test('hello');
<< false
```

Document Object Model (DOM)

Getting Elements

The DOM privides several methods that allow us to access any element. Can return a node object or node list (array-like object). The objects can be assigned to a variable.

const body = document.body;

We can check the type:

typeof body;

//"object" (special Node object

Use nodeType property to find what type of node:

Body.nodeType //1

Nodes have numerical codes.

Code	Type
1	element
2	attribute
3	text
8	comment
9	body

Legacy DOM Shortcut Methods

- `Document.body` returns the body element of a web page, as we saw in the previous example.
- `Document.images` returns a node list of all the images contained in the document.
- `Document.links` returns a node list of all the `<a>` elements and `<area>` elements that have an `href` attribute.
- `Document.anchors` returns a node list of all the `<a>` elements that have a `name` attribute.
- `Document.forms` returns a node list of all the forms in the document. This will be used when we cover forms in Chapter 8

Get Elements by ID

document.getElementById('title');

Get Elements by their Tag Name

document.getElemetnsByTagName('li');

This makes a node list that we can use the index notation to find each list item:

```
listItems[0];
<< <li class='hero'>Superman</li>
listItems[1];
<< <li class='vigilante hero' id='bats'>Batman</li>
listItems[2];
<< <li class='hero'>Wonder Woman</li>
```

Get Elements by Class name

document.getElementByClassName('hero');

You can use .length to check how many elements have that class name.

Query Selectors

document.querySelector() – Use CSS notation to find the first element in the document that matches a CSS selector provided as an argument.

document.querySelectorAll() – Uses CSS notation but returns a node list of all the elements in the document that match the CSS query selector.

For example, the following could be used instead of `document.getElementById()` :

```
document.querySelector('#bats');
<< <li class="vigilante hero" id="bats">Batman</li>
```

And this could be used instead of `document.getElementsByClassName` :

```
document.querySelectorAll('.hero');
<< NodeList [<li class="hero">, <li id="bats">, <li class="hero">]
```

CSS pseudo-selectors can be used to pinpoint an element. Example:

```
const wonderWoman = document.querySelector('li:last-child');
```

The `querySelector()` method can be called on*any*element, rather than just `document` . For example, we can get a reference to the `<ul>` element, using the following code:

```
const ul = document.querySelector('ul#roster');
```

Now we can use the `querySelector()` method on this element, to find a `<li>` element with an id of 'bats':

```
const batman = ul.querySelector('li#bats')
```

Setting An Element's attribute

setAttribute can change the value of an elemnet's attributes. Takes 2 arguments.

Example: Change the class of t the element in the wonderWoman variable to 'villain'.

```
wonderWoman.setAttribute('class', 'villain');
<< undefined
wonderWoman.getAttribute('class');
<< "villain"
```

The classList Property

**The add method can be used to add a class to an element without overwriting any classes that already exist – Changing the classname property will overwrite all other classes that have already been set on the element.

```
wonderWoman.classList.add('warrior');
```

Use remove method to remove a specific class from an element:

```
wonderWoman.classList.remove('warrior');
```

Use toggle method to add a class if an element doesn't have it already and remove the class if it does have it.'

```
wonderWoman.classList.toggle('hero'); // will remove the 'hero' class
<< false
wonderWoman.classList.toggle('sport'); // will add the 'hero' class back
<< true
```

Use contains method to check to see if an element has a particular class:

```
wonderWoman.classList.contains('hero');
<< true
wonderWoman.classList.contains('villain');
<< false
```

Create an Element:

createElement() method that takes a tag name as a parameter

```
const flash = document.createElement('li');
```

Create a Text Node:

Once empty element is created use document.createTextNode() to add content.

```
const flashText = document.createTextNode('Flash');
```

Append Nodes:

appendChild() method that adds another node as a child node.

```
flash.appendChild(flashText);
```

Now we have a `<li>` element that contains the text we want. So the process to follow each time you want to create a new element with text content is this:

1. Create the element node
2. Create the text node
3. Append the text node to the element node

This can be made simpler by using the `textContent` property that every element object has. This will add a text node to an element without the need to append it, so the code above could have been written as the following:

```
const flash = document.createElement('li');
flash.textContent = 'Flash';
```

Function to create Elements

```
function createElement (tag,text) {
    const el = document.createElement(tag);
    el.textContent = text;
    return el
}
```

Methods for adding elements to the page

appendChild()

insertBefore()

insertAfter()

Method for removing elements to the page

removeChild()

Replacing Elements on a page

replaceChild()

```
const h1 = document.getElementById('title');
const oldText = h1.firstChild;
const newText = document.createTextNode('Justice League of America');
h1.replaceChild(newText,oldText);
```

innerHTML

Writable and used to place a chunk of HTML inside an element.

```
h1.innerHTML = 'Suicide Squad';
```

or enter raw HTML as a string:

```
heroes.innerHTML = '<li>Harley Quinn</li><li>Deadshot</li><li>Killer Croc</li><li>Enchantress</li><li>Captain Boomerang</li><li>Katana</li><li>Slipknot</li>';
```

Updating the CSS

```
const heroes = document.getElementById('roster');
const superman = heroes.children[0];
superman.style.border = "red 2px solid";
<< "red 2px solid"
```

```
superman.style.backgroundColor = 'blue';
<< "blue" // Anything that uses a dash is written in camel case (backgroundColor)
```

Or use bracket notation

```
superman.style['background color'] = 'blue';
<< "blue"
```

Events

Event Listeners

```
document.body.addEventListener("click", doSomething);
```

Inline Event Handlers

Attributes added to markup:

onclick

```
<p onclick="console.log('You Clicked Me!')">Click Me</p>
```

Older Event Handlers

Use the event handler properties that all node objects have. Can be assigned to a function that would be invoked when the event occurs. Good because keeps JavaScript out of the HTML markup.

```
document.onclick = function (){ console.log('You clicked on the page!'); }
```

Types of Events

Mouse events:

Click, mouseup, mousedown, dblclick, mouseover, mouseout, mousemove

```
const mouseParagraph = document.getElementById('mouse');
mouseParagraph.addEventListener('mouseover', highlight);
mouseParagraph.addEventListener('mouseout', highlight);
```

```
mouseParagraph.addEventListener('mousemove', () =>  console.log('You Moved!')
);
```

Keyboard Events:

Keydowm, keypress, keyup

1. The keydown event occurs when a key is pressed and will*continue to occur*if the key is held down.
2. The keypress event occurs after a keydown event but before a keyup event. The keypress event only occurs for keys that produce character input (plus the 'Delete' key). This means that it’s the most reliable way to find out the character that was pressed on the keyboard.
3. The keyup event occurs when a key is released.

Modifier Keys

Shift, Ctrl, Alt and meta (Cmd on Mac)

Fires keydown and keyup events – not keypress because don't produce characters.

shiftKey, ctrlKey, altKey, and metaKey are properties of the event object.

```
addEventListener('keydown', (event) => {
if (event.key === 'c' && event.ctrlKey) {
        console.log('Action canceled!');
    }
});
```

Touch Events

Smartphones and tablets, touch-screen etc.

Important to support mouse events and touch events so non touch devices are also supported.

touchstart – user initially touches the surface – * click event safer option

touchend – user stops touching the surface.

touchmove – after user has touched the screen then moves around without leaving

touchenter – user has already started touching the surface but then passes over the element to which the event listener is attached

touchleave – User is still touching the surfacem but leaves the element to which the event listener is attached

touchcancel – Touch event is interrupted

Remove Event Listeners

removeEventListener()

Stopping Default Behavior

preventDefault()

Event Propagation – The order that events fire on each element.

Bubbling – Default behavior Event fires on the element clicked on first, then bubbles up the document tree, firing event on each parent element until root node – Stop bubbling with event.stopPropagation()

Capturing – Starts by firing an event on the root element, then propagates downwards, firing event on each child element until reaches the target element clicked on.

Event Delegation

Attach an event listener to a parent element in order to capture events that are triggered by its child element.