JavaScript's role in Web Development

What is JavaScript (JS)?

- JavaScript (JS) is the #1 programming language for a web developer to learn.
- JS is used by millions of websites and is required for nearly all developer jobs.
- On the Frontend, JS interprets web pages as a hierarchical collection of objects, known as the **Document Object Model (DOM)**.
- On the Backend, **Node.js** and **React.js** work with a **server** object and databases.

What can JavaScript do for Websites?

Add interactivity

JavaScript enables interactivity, which empowers the user. When you click a button and something happens, that's probably JavaScript in action.

Manipulate the DOM

JS can get any html element and do stuff, like:

- change an element's content or modify an element's CSS properties:
- retrieve values from form objects
- perform calculations and output the result
- When fresh data appears without the whole page reloading, that's JS.

JavaScript is written...

- inside **script** tags, in either the head or body of an html file.
- in a **.js** file, which is imported into either the head or body of an html page.

> **Per Wikipedia:**
>
> JavaScript (JS), is a programming language that is one of the core technologies of the World Wide Web, alongside HTML and CSS. Over 97% of websites use JavaScript on the client side for web page behavior, often incorporating third-party libraries. All major web browsers have a dedicated JavaScript engine to execute the code on users' devices...
>
> JavaScript engines were originally used only in web browsers, but are now core components of some servers and a variety of applications. The most popular runtime system for this usage is Node.js ...
>
> Although Java and JavaScript are similar in name, syntax, and respective standard libraries, the two languages are distinct and differ greatly in design.

1. Launch your code editor (Visual Studio Code, Sublime Text, etc.). If you are in a Noble Desktop class, launch Visual Studio Code.

2. Navigate to the JavaScript-Fundamentals folder.
3. Go into the **00-Introduction** folder and open the file **00.01-Introduction-to-JavaScript.html**.

---

script tag

When written directly in an html page, JS is wrapped in a script tag, which goes right before the closing body tag. Putting the JS code at the end lets the web page fully load before any JS "goes looking" for any particular element to manipulate.

1. Add a pair of script tags right before the closing body tag. Separate the script tags and skip a line:

```
    <script>
    </script>
</body>
```

---

output

Programs produce output – be it a single-line greeting or a complete, interactive application. Output is also necessary at each step in the coding process, as it provides feedback that tells if the code written so far is working—or if it's buggy (has errors).

---

alert()

alert() is a JS method. Think of methods as the verbs of programming languages. They perform actions. In this case, the action is to pop-up a dialog box that displays the string (text) which was passed to the parentheses.

1. Inside the script tags, call the alert() method:

```
<script>
    alert();
</script>
```

---

string

In JS and other programming languages, plain text is known as a string. Strings go in quotes, either single or double.

1. Inside the script tag, add an alert containing the string 'Hello World':

```
<script>
    alert('Hello World');
</script>
```

2. Save the file and open it in Chrome. You should get an alert (browser pop-up) that says Hello World.
3. Close the alert, but leave the html page open in the browser. We'll be reloading the page often as we proceed.

---

console.log()

Running Javascript in the Browser Console:

- You do not need a DOM, that is to say, a web page, in order to write and run Javascript.
- You can write JS directly in the **Console** for testing purporse, as many developers do—-no html page required.
- To output content to the Console, use **console.log()**.

1. Open the Console. In the Chrome browser, right click the page and choose **Inspect**. Then click the **Console** tab. The keyboard shortcut for opening the Console is **command-option-i**.

Now, we'll switch to the console.log() method, which is what we'll be using from now on. console.log() also outputs whatever is in its parentheses, but its output goes to the Console.

1. Use console.log() to output another message. Strings (text) can go in either double or single quotes, so switch to double quotes:

   ```
   alert('Hello World');
   console.log("Hello from the Console");
   ```

2. Save the file and reload the browser. You get the alert again, but there's more:
3. Right-click the page and choose Inspect; then click the Console tab. It should say: Hello from the Console

---

Global Window Object

The **global window object** is the top-level object in JavaScript, corresponding to the browser window itself.

- **alert()** is a method of the window
- **console** is an child object of the *window* object.
- **log()** is a method of the *console* object. Because the window object is top-level, it is assumed, and therefore can be omitted from the syntax. But we can also add them:

1. Put **window** in front of the alert and console keywords and run it again. It still works:

   ```
   window.alert('Hello World');
   window.console.log("Hello from the Console");
   ```

---

semi-colon

Notice the semi-colons (;) which mark the end of each line of code. Semi-colons are optional, but recommended. We'll be using them.

---

comments

Comments explain what the code is doing. Comments are for us humans and, as such, are ignored by JS when the program is run. While comments have no effect on how a program runs, they do make the code easier to read and understand. We recommend you comment your code. Even if no one else will be reading the code, it will help you remember what you wrote and why later.

---

single-line comment

A // at the start of a line turns the whole line into a comment.

14. Add a couple of one-line comments to start the script:

```
// alert() and console.log() output what's in parentheses
// window. before alert and console are optional
window.alert('Hello World');
window.console.log("Hello from the Console");
```

---

in-line comment

If the code and its comment are short enough, the comment can go on the same line as an in-line comment.

15. Add an in-line comment to describe an alert():

```
window.alert('Hello World'); // browser pop up
window.console.log("Hello from the Console");
```

---

commenting-out code

Comments are used to deactivate code without having to delete it. A double slash at the start of a line comments it out. By commenting out code instead of deleting it, we keep it around for study and review purposes or in case we want to reactivate it later. The keyboard shortcut for commenting out code is **command-slash**.

16. Comment-out the alert so that we don't keep getting that pop-up.

```
// window.alert('Hello World'); // browser pop-up
window.console.log("Hello from the Console");
```

---

multi-line comments

If a comment runs multiple lines, you can wrap all in /* ... */.

17. Replace the single-line comments with a multi-line comment; keep the commented-out alert for study and reference:

```
/*
Ways JS outputs content (besides to a Web page)
alert("howdy") -- gives a pop-up that says howdy
console.log("hola") -- outputs hola to the Console
Global Window Object: the top level object in JS
alert() is a method of window object
console is a child object of window object
Therefore, you may optionally begin w window-dot:
window.alert("howdy")
window.console.log("hola")
*/

// window.alert('Hello World'); // browser pop-up
window.console.log("Hello from the Console");
```

anonymous function

A function without a name is referred to as an **anonymous function**. These occur when the function code is written in-line or set equal to an event, as opposed to as a separate location that must be referenced by name. Example of anonymous function:

```
myButton.onclick = function() {
   alert('You clicked the button!');
};

// or:
myButton.addEventListener('click', function() {
   alert('You clicked the button!');
});
```

Also see **function**

---

argument / parameter

- Data that goes between the parentheses of a method or a function is called an **argument**.
- When the function is defined, the data in the parentheses, known as **parameter(s)** is defined as a variable.
- When the function is called, an argument is passed into the parentheses. In this way, the argument sets the value of the parameter:

```
function greetUser(name) {
   console.log(`Hello, ${name}!`);
}
greetUser("Brian"); // Hello, Brian!
```

---

array

An array is a type of **object** that store data as numbered list items. The syntax is to put the values, which can be of any datatype, inside of square brackets. Items are referenced by their numeric position, known as the **index**, which start numbering at 0.

```
let fruits = ['kiwi', 'pear', 'plum'];
console.log(fruits[0]); // kiwi
```

Also see **index**, **object**.

---

arrow function

An arrow function uses **=>** instead of the keyword **function**. An arrow function implicitly returns a value, so for functions with only one line of logic, the **return** keyword and curly braces **{}** may be omitted. This concise syntax makes arrow

functions the preferred way of writing callbacks, which are anonymous functions passed to other functions. Here we declare an array of numbers followed by calling the sort() method on the array to put the items in ascending order. The sort() method takes a callback, for which we are using arrow syntax:

```
const nums = [13, 6, 45, 2, 89, 11, 4, 23];
nums.sort((a,b) => a - b);
console.log(nums); // [2, 4, 6, 11, 13, 23, 45, 89]
```

---

assignment operator

The equals sign **=** is known as the **assignment operator**, because it assigns or sets a value: **x = 5** assigns **5** as the value of **x**.

```
let x = 8;
x = 9;
```

---

asynchronous

The term **asynchronous** (async, for short) refers to code being executed at different times, particularly due to delays in data being made available as in a server request. One function makes a request for data from a server, and another function must asynchronously wait for that data to load and be made available. Callback functions are used to handle asynchronous requests.

---

block scope

Block refers to variables that are available only within the particular code block in which they are declared, as defined by a pair of curly braces { }. Also see **scope**, **function scope**, and **global scope**.
The rules for **variable scope**:

- variables not declared inside any { } are **global**, which means they are available everywhere in the script.
- **let** and **const** declared inside { } are **block scoped**, which means they are available only inside that code block.
- **var** declared inside { } are **global** (unless the { } is a function)
- Any variable declared inside a function is available only to that function. These are known as **local variables** or **function-scoped variables**.

---

boolean

A variable or value that can be only **true** or **false**. Conditional logic resolves to a boolean:

```
let isOnline = true;
isOnline = false;
```

block (of code)

A code block is all the code that occurs between a set of curly braces. When declared with let or const, any variable declared inside a block is available only to that block and is therefore called **block scoped**. The main types of blocks are functions, if-statements and loops:

```
// code block / block scope examples
// 1. function
function greetUser(username) {
   return `Welcome ${username}`;
}

// 2. for loop
// i is scoped to the block (does not exist outside the block)
function greetUser(username) {
for(let i = 0; i < 5; i++) {
   console.log(i); // 0, 1, 2, 3, 4
}
console.log(i); // undefined (i only exists inside its block)

// 3. if statement
let x = 5;
// if x is greater than 2, square it and save the value to y
if(x > 3) {
   let y = x ** 2;
   console.log(y); // 25
}
console.log(y); // undefined (y only exists inside its block)
```

callback

An anonymous function that is passed to another function as its argument is known as a **callback**. Callbacks frequently have arguments of their own. Here we declare an array of numbers followed by calling the sort() method on the array to put the items in ascending order. The sort() method takes a callback:

```
const nums = [13, 6, 45, 2, 89, 11, 4, 23];
nums.sort(function(a,b) {
   return a - b;
});
console.log(nums); // [2, 4, 6, 11, 13, 23, 45, 89]
```

comment

Comments are notes or commented out code which are ignored by the browser.

```
// declare a variable
let fruit = "apple";
// alert(fruit);
```

---

condition

A condition is something that is evaluated and resolves to *true* or *false*. Conditions are found in loops and in if-else statements (conditional logic). The condition of a loop must be true for the loop's code block to execute. In this example, the condition is **x < 10**. If the condition is true, the code inside the {} will run:

```
let x = 9;
if(x < 10) {
   x++;
}
```

---

console

The **console** is a browser debugging environment where you can output code via the command. The following would produce output of **25** in the console, which may be accessed via Dev Tools or **Inspect > Console**: `let x = 5; console.log(x * x); // 25`.

---

const

The **const** keyword for declaring primitive variables (string, number, boolean) prevents them from being changed. Objects declared with **const** can have their properties and items changed by cannot have their data type changed.

---

counter variable

A variable that exists to keep count of something. Loops rely on counter variables to be incremented or decremented with each iteration of the loop so that the condition driving the loop can ultimately become false and the loop can therefore stop.

---

document

The webpage, a child object of the Window. JS "sees" the document and all of its elements as a hierarchy of so many nested objects, which JS can manipulate in a myriad of ways.

DOM (Document Object Model)

JavaScript "sees" the entire web page as a hierarchical tree of objects, known as the DOM, with the Window being the top-level object. The structure is analogous to how in HTML, we have a nested hierarchy of tags.

dot syntax

The use of dots to access or “drill down into” objects is known as dot syntax. In this example, **triStates.ny.capitol** returns the value of the **capitol** property, "Albany", which is a child of the **ny** property, itself an object, which is a child of the **triStates** object:

```
const triStates = {
  ct: {
    nickname: "Constitutino State",
    capitol: "Hartford"
  },
  nj: {
    nickname: "Garden State",
    capitol: "Trenton"
  },
  ny: {
    motnicknameto: "Empire State",
    capitol: "Albany"
  }
}

console.log(triStates.ny.capitol); //  Albany
console.log(triStates.ny); // {nickname: "Empire State", capitol: "Albany"}
console.log(triStates.nj.motto); //  Garden State
console.log(triStates.nj); // {nickname: "Garden State", capitol: "Trenton"}
```

else

Conditional logic is often expressed by if-else statements: if a condition is true, do something; **else** do something else:

```
let x = 2;
if (x > 3) {
  console.log('The condition is true!');
} else {
  console.log('The condition is false!');
}
```

Also see **if-else logic**.

event

An **event** is something that happens during the running of an application. Events are often used as triggers to call functions. Common events include **click** and **change**:

```
myButton.addEventListener('click', doSomething);
mySelectMenu.addEventListener('change', doSomethingElse);
```

---

for loop

A for loop executes a block of code again and again, as long as a condition is true. The condition must become false eventually, or else the loop never ends (infinite loop). A for loop considers three pieces of information between its parentheses in order to decide if it will run the code inside its curly braces: a **counter**, a **condition** and an **incrementer**:

```
for(let i = 0; i < 10; i++) {
  console.log(i*i); // 1, 4, 9, 16, 25, 36, 49, 64, 81, 100
}
```

---

function

A function is a block of code that runs only when it is invoked (called). The function can be called in the code or by an event on the web page, such as a button click. A function usually (but not always) has a name to call it by:

```
function greetUser(name) {
  console.log(`Hello, ${name}!`);
}

greetUser("Brian"); // Hello, Brian!
myButton.addEventListener('click', greetUser);
```

---

function scope / local scope

Function scope / local scope refers to variables that are available only inside the function in which they are defined. Also see **scope**, **global scope** and **local scope**.

---

global/ window object

The highest level object in JS is the **global window object**, the direct children of which include the *document*, **console**, **navigation** and **history** objects. The window object is assumed and need not be referenced:

```
alert("Hello!");
// or:
window.alert("Hello again!");
```

---

global scope, global memory

Global scope / memory refers to variables that are declared at the level of the **global object** and as such are available throughout a script, as opposed to being confined to the scope of a particular code block. Also see **scope**, **function scope**

and **local scope**. In this example **x** is global because it is declared outside of any code block, but **i** is scoped to the for loop, and so is *not* available outside the code block. Attempts to reference it result in **ERROR not defined**.

```
let x = 5; // global variable

for(let i = 0; i < 10; i++) {
    console.log(i);
}

console.log(x); // 5
console.log(i); // ERROR i is not defined
```

---

history object

A direct child of the window, the **history object** has methods pertaining to the browser history. Although the *window* part is optional, it is customary to include window when referencing the history object:

```
    // reload the last visited browser page
    window.history.back();
```

---

hoisting

A function can be called before it is defined in the code, because functions are automatically hoisted (lifted) to the top of the code block. By contrast, variables are not hoisted and therefore can only be referenced after they have been declared.

```
// calling the function above where it is defined works, because functions are hoisted (lifted) to the top of the code
sayHey("Joe"); // Hey, Joe!

function sayHey(name) {
    console.log(`Hey, ${name}!`);
}

// referenceing a variable above where it is declared throws an erro, because variables are NOT hoisted (lifted) to the top of the code
console.log('Bunny likes ' + veg); // Error: veg is not defined
let veg = 'kale';
```

---

if-else logic, if-statement

The logic of decision making in programming, conditional logic is often expressed by if-else statements: if a condition is true, do something; else do something else:

```
let x = 10;
let y = 3;

if(x > y) {
   console.log('x is greater than y!');
} else {
   console.log('x is equal to or less than y!');
}
```

Also see **conditional logic**.

---

increment

To increment means to increase or go up. To increment a variable means to have its numeric value go up. In loops, a counter variable is incremented or decremented each time through the loop.

```
// square the counter variable i each time the loop runs
// each time the loop runs, increment i by i with i++
for(let i = 0; i <= 5; i++) {
   console.log(i ** 2); // 0, 1, 4, 9, 16, 25
}

// square the counter variable i each time the loop runs,
// but increment by 2 each time with i+=2
for(let i = 0; i < 10; i+=2) {
   console.log(i ** 2); // 0, 4, 16, 36, 64, 100
}


### index

The position number of an item in an array, with the first item at index 0:

```js
let fruits = ['kiwi', 'pear', 'plum'];
console.log(fruits[0]); // kiwi
```

---

key

The name of a property is called the *key*. This object has keys of **name**, **age** and **retired**.
```

```
let guy = {
   name: "Bob",
   age: 40,
   retired: false
}
```

length

Called on an array, the *length* property returns the number of items in the array:

```
let fruits = ['kiwi', 'pear', 'plum'];
console.log(fruits.length); // 3
```

---

let

The **let** keyword is for declaring block-scoped variables, meaning that the variables only exist inside the curly braces in which they are declared. In this example, **power** is declared in the **global scope** and so is available everywhere in the script. But **i** and **cube** are declared inside the for loop, which makes them **block scoped** variables, so NOT available in the global scope:

```
let power = 3;

for(let i = 0; i < 10; i++) {
   let cube = i ** power;
   console.log(cube);
}

console.log(power); // 3
console.log(i); // ERROR cube is not defined
console.log(cube); // ERROR cube is not defined
```

---

loop

A loop is a programming routine that executes a block of code again and again, as long as a condition is true. There are a few kinds of loops in JS, including "for loops" and "while loops":

```
   for(let i = 0; i < 10; i++) {
      console.log(i ** 4); // 1, 16. 81, 625 ... 10000
   }
```

---

method

Methods are the verbs of programming languages-—they do stuff; they perform actions. Just as verbs go together with nouns, methods are attached to objects.

- Methods perform a myriad of tasks: output content to html elements, load data from the server, play sounds, create new html elements, and so on.
- Dot-syntax is how we access the methods of a JS object. First comes the object, followed by a dot. Then comes the method. So it goes object-dot-method:

```
let dateTime = new Date();
// the Date object's getHours() method:
let hour = dateTime.getHours();
// the console's log method:
console.log(hour); // integer in 0-23 range
```

---

modulo

The **modulo operator**, symbol % returns the remainder when one number is divided by another:

```
console.log(15 % 4); // 3
```

---

nesting

Nesting refers to something inside of another similar thing, such as a loop inside of a loop, a block of conditional logic (if-statement inside) conditional logic, or an array inside of an array:

```
// nested for loop: a loop inside a loop
for(let i = 0; i < 5; i++) {
    for(let j = 0; j < 5; j++) {
        console.log(i**j); // i to the j power
    }
}

// nested if-else:
let weather = "cloudy";
let windy = true;

if(weather == "cloudy") {
    console.log("Go to the park");
    if(windy) {
        console.log("Fly a kite");
    } else {
        console.log("Have a picnic");
    }
}
```

---

objects

Objects are variables that can store more than one value at a time. They are divided broadly into Objects, per se, and a type of object called an Array. Objects proper store data inside curly braces as **properties**, consisting of name-value (key-value) pairs: name:

```
let cat = {
  name: "Fluffy",
  age: 4,
  cute: true
};
```

---

property

A property is a name-value (key-value pair) that belongs to—is scoped to—an object. As such, a property is essentially a local variable. The properties of an object can be nested. Properties are accessed by means od **dot syntax**:

```
let cat = {
  name: "Fluffy",
  age: 4,
  cute: true,
  owner: {
    name: "Bob",
    age: 40,
    job: "Programmer"
  }
};
console.log(cat.name); // Fluffy
console.log(cat.owner.name); // Bob
console.log(cat.age); // 4
console.log(cat.owner.age); // 40
console.log(cat.cute); // true
console.log(cat.owner.cute); // undefined
console.log(cat.job); // undefined
console.log(cat.owner.job); // Programmer
```

---

return value

Think of a function as an input-output machine, with the arguments being the input and the **return** value being the output. A **return** value “exports” the function result for use elsewhere. In order to "capture" the return value, set the function call equal to a variable. Many JS methods return a value.

```
function addNums(a, b) {
   return a + b;
}

let sum1 = addNums(5, 6);
console.log(sum1); // 11

let sum2 = addNums(8, 9);
console.log(sum2); // 17

// JS methods that return a value:
// instantiate the Date object
let dateTime = new Date();
// get the current hour as an integer from 0-23
let hour = dateTime.getHours(); // the hour from 0-23 is returned
console.log(hour); // 15 (if current time is 3:00-3:59pm)
// declare an array
let cars = ['Audi', 'BMW', 'Chevy', 'Dodge'];
// pop (remove) the last item; the pop() method returns the item
let lastCar = cars.pop();
console.log(lastCar); // Dodge
```

---

string

Pure text in quotes is a string. Setting a variable equal to a string gives the variable a data type of **string**:

```
let firstName = "Brian";
console.log(firstName, typeof(firstName)); // Brian string
```

---

ternary

A ternary expression is a concise, one-line alternative to an else-if statement. It uses **?** and **:** instead of **if()** and **else**, while also dispensing with the curly braces:

```
let x = 5;

// if x is less than 10, add 1; else set x equal to 0:
if(x < 10) {
   x++;
} else {
   x = 0;
}

// ternary version of the above:
x < 10 ? x++ : x = 0;
```

---

this

The this keyword refers to various objects, depending on where it is encountered:

- in the global scope, *this* is the Global Window Object
- inside a function, *this* is the object (button, menu, etc) whose event ('click', 'change', etc) called the function.
- in a function defined inside an object (a method), *this* refers to the object itself.

---

var

The **var** keyword for declaring variables has largely been supplanted by **let** due to the greater control over variable scope provided by the latter. A **var** declared inside the curly braces of a loop or if-statement are in the global scope, whereas **let** variables declared inside the curly braces of a loop or if-statement are scoped to that code block:

```
// j exists outside the loop, because it is declared with var:
for(var j = 0; j < 5; j++) {
   console.log(j**2); // j squared
}
console.log(j); // 5

// i exists ONLY inside the loop, because it is declared with let:
for(let i = 0; i < 5; i++) {
   console.log(i**2); // i squared
}
console.log(i); // ERROR: i is not undefined
```

---

while

A **while loop** is a kind of loop. Unlike a **for loop**, which has the counter, condition and incrementer inside parentheses, a while loop has these elements scattered about, with the counter declared above the loop, the condition evaluated inside

parentheses, and the incrementer inside the curly braces. In general, use a for loop if you know exactly how many times you want the loop to run, and use a while loop if the number of iterations is not pre-determined:

```
let frutes = ['apple', 'banana', 'grape', 'kiwi', 'orange', 'pear'];

// using a for loop, because the number of iterations is known:
for(let i = 0; i < frutes.length; i++) {
   console.log(frutes[i] + " has " + frutes[i].length + " letters!");
}

/* using a while loop because the number of iterations is not known; the loop ends with the first 4-letter item, but its index may not be known. */

let n = 0;
while (frutes[n].length != 4) {
   console.log(frutes[n] + " is " + frutes[n].length + " letters long!");
   n++;
}

/* the for loop will run 6 times as it fully traverses the array, but the while loop will only run 3 times, because it will quit when the condition is false, which occurs when the current item is 'kiwi'.
apple has 5 letters!
banana has 6 letters!
grape has 5 letters!
kiwi has 4 letters!
orange has 6 letters!
pear has 4 letters!
apple is 5 letters long!
banana is 6 letters long!
grape is 5 letters long!
*/
```

---

window

The window object, representing the browser, is at the top of the DOM hierarchy. It is the top-level object in JavaScript. It is above the document (web page) itself, which is a child of the browser. The console is another direct child of the window.

Workbook Instructions

Here's how to proceed with each lesson and lab in this course:

HTML File

- Open the current lesson **.html** file in your Code Editor.
- Scroll down to confirm that the **script** tag is importing **FINAL.js**.
- Preview the **html** file in the Chrome browser.
- Right-click **Inspect** to check the Console for the final output.
- Back in the html file, switch to importing the **PROG.js** file, which is a blank copy of **START.js**.

JS Files

- Each lesson has 3 JS files: **START.js**, **PROG.js** and **FINAL.js**.
- Write your code for each lesson in **PROG.js**, where starter code and instructional comments are provided.
- Each numbered step in a lesson has code for you to type.
- Each step explains what you are to type next, and why.
- Check the Console with each step to make sure your code is working.
- It is recommended that you do not type your code in **START.js**, but rather keep it as a blank and so a Save As to get a **PROG.js**, as needed. That way, you will always have a blank starter page in case you wish to review a lesson by typing the code over again.

Labs

- Most lessons are accompanied by a **Lab** for you to do after you have completed that lesson.
- Labs are challenges, in that the code steps are not provided. Instead, you are given prompts and must figure out what code to type.
- Unlike lesson html files, lab html files do NOT have separate JS file to import. In order to streamline the files system, all lab JS code is typed in the Lab html page, in the script tag.
- All lab .html files have a companion Solution html file. This is where you check the answers—but do so only after having given the lab your best effort.

Submitting Lab work as Homework

- Labs are homework. When you are done, check the Solution file, but also submit your work through the class Slack Channel, or by whatever means your instructor provides.
- The Solution JS code is not necessarily the only way to do something. By submitting your code—which may differ from the “official solution”, your instructor can give you valuable coding tips and feedback on best practices.

Markdown Files (.md files)

- Each lesson has a markdown file with extension **.md**.
- The markdown file contains the step-by-step instructions for the lesson or lab.
- This file can be rendered in your VSCode Editor by typing **Command-Shift-V** (Mac) or **Ctrl-Shift-V** (Windows).
- You may want to use a vertical split screen view, with your coding page (PROG.js) in one pane, and the readme file for that lesson in the other pane.
- To enable vertical split screen in VSCode, choose **View > Editor Layout > Split Up**.
- Both lessons and labs have companion markdown files.

PDF files

- **Noble-Desktop-JavaScript-Bootcamp-Workbook.pdf** is a collection of all the many markdown files for each lesson and lab. It is assembled all in one place for your convenience.
- Workflow option: you may prefer to keep your whole screen for coding and to not to split your screen between the JS file and the markdown file. In that case, you may want to set up a second computer / monitor and keep the the **PDF** file open on that other monitor.
- PDF files can be viewed in Adobe Acrobat Reader, in Preview on Mac, and in the Chrome browser.

- var vs. let for declaring variables
- primitive variable types: string, number, boolean, undefined
- typeof() method for checking data type

---

variables

A variable is a container that stores a value. Some variables (vars, for short) can hold many values at a time. Other vars can only hold one value at a time. Vars that only hold one value at a time are sometimes referred to as **primitive types**. These “primitives” are the topic of this lesson.

---

variable data types

There are two broad categories of variables, **objects** and **primitives**:

object is a broad **data type** encompassing variables that can store multiple values. These include:

- **object**: a data structure with properties as name-value pairs and, optionally, with methods (functions scoped to the object)
- **array**: ordered lists of items, with each item stored by its numeric position, called the index.
- **DOM object**: JS "versions" of html elements (div, button, etc.)
- **function**: code blocks which only run when invoked (called)
- **null**: an empty object, used typically as a placeholder value

primitives are variables that are capable of storing just **one** value at a time. Primitives are tecchnically objects too, although object generally refers to the above. Primitives have the following **data types**:

- **string**: value as text in quotes (e.g. "Hello World")
- **number**: value as integer and decimal (e.g. 3, 3.5)
- **boolean**: value is either **true** or **false**
- **undefined**: with no value assigned, typically used with the understanding that a value will be assigned later

In this lesson we will focus on primitives.

---

declaring variables with let

To begin using a variable, you need to **declare** it. This is done by writing a **keyword** (**var** or **let**), followed by the name of your variable, which can be pretty much anything you like, although naming rules and conventions do apply:

---

variable naming rules and restrictions

- No spaces allowed in variable names
- No special characters allowed, except **$** and _
- Name cannot start with a number (**1day** bad; **day1** good)
- No reserved words allowed (**alert** bad; **myAlert** good)

---

variable naming conventions (best practices)

- Use **camelCase** (it's **highScore**, not **high_score**)
- Don't use all UPPERCASE, unless declaring a constant (value will never change)
- Choose concise names (**tel**, not **telephoneNumber**)
- Choose precise names (**salesTax** not **additionalCharge**)

A variable is declared with **var** or **let**, although **let** is the more modern syntax and should be used instead of **var** for reasons that we will expound when we start talking about **variable scope**.

---

string variables

- **string** is a **datatype** for text.
- **string** values go in quotes (double or single quotes both work)

1. Declare a variable with **var** and assign it a string in double quotes:

```
var pet = "cat";
console.log(pet); // cat
```

2. Change the value to another string, this time in single quotes:

```
pet = 'dog';
console.log(pet); // dog
```

3. One difference between **var** and **let** is that a **var** can be redeclared. Redeclare **pet**:

```
var pet = 'bunny';
console.log(pet); // bunny
```

---

redeclaring variables

A variable declared with **var** may be redeclared, but a variable declared with **let** cannot be redeclared. Attempting to do so throws an error.

4. Declare a variable with **let**, and then try to redeclare it:

```
let petSound = "Woof!";
console.log(petSound); // Woof!
let petSound = "Grrr!";
// Error: Identifier 'petSound' has already been declared
```

5. To change the value of an existing variable, don't redeclare it; just set it equal to something else. Comment out **let petSound = "Grrr!"**, and then set **petSound** to "Grrr!" *without redeclaring* the variable:

```
// let petSound = "Grrr!";
petSound = "Grrr!";
console.log(petSound); // Grrr!
```

6. Multiple values can be outputted in the same **console.log**: Output both variables in *one* console.log:

```
console.log(pet, petSound); // dog Grrr!
```

7. For added clarity, you can 'label' console output:

```
console.log('pet:', pet, ' petSound:', petSound);
// pet: dog  petSound: Grrr!
```

---

number variables

A **number** can be an **integer** (int, for short) or a **float** (decimal).

- There are no commas in numeric values.
- Be it integer or float, a number's **datatype** is number.

8. Declare three numeric variables: integer, float, and four-digit int:

```
let price1 = 35; // integer
let price2 = 3.5; // float
let price3 = 3500; // no comma
console.log(price1, price2, price3); // 35 3.5 3500
```

9. Naturally, numeric variables can be used in mathematical operations, the result of which may also be assigned to a variable. Take these basic math operators for a spin: *+, -, , / :

```
let sum = price1 + price2 + price3;
console.log(sum); // 3538.5
console.log(price1 - price2); // 31.5
console.log(price2 * price3); // 12250
console.log(price3 / price1); // 100
```

---

boolean variables

A boolean is a variable with a value that is either **true** or **false**.

- boolean names often begin with *is* to emphasize the either-or concept
- **toggling** or **flipping** a boolean refers to changing its value

10. Declare 2 booleans, one **true**, one **false** :

```
let premiumMember = true;
let isOnline = false; // 'is' indicates that this is a boolean
console.log(premiumMember, isOnline); // true false
```

11. **Toggling** or **flipping** a boolean can be done either by direct assignment or by putting an exclamation point (**!**) in front. Flip **premiumMember** from true to false by direct assignment. Also flip **isOnline**, but do so by putting **!** in front of it:

```
premiumMember = false;
console.log('premiumMember', premiumMember); // premiumMember false
isOnline = !isOnline;
console.log('isOnline', isOnline); // isOnline true
```

The advantage of using **!** to toggle/flip a boolean is that you do not need to know the current value; whatever it is, **!** makes it the opposite.

---

undefined

A variable can be declared **without** a value being assigned. The assumption is that a value will be provided later. Until then, both value and datatype are **undefined**.

12. Declare a variable, but don't assign it a value:

```
let player1;
console.log('player1', player1); // player1 undefined
```

---

typeof() method

The **typeof()** method takes a variable as its argument and returns the **datatype**.

13. Declare variables of each of the 4 major **primitive** datatypes: string, number, boolean and undefined. Then log the name, value and datatype:

```
let ketchup = "Heinz";
console.log('ketchup', ketchup, typeof(ketchup)); // ketchup Heinz string

let varieties = 57;
console.log('varieties', varieties, typeof(varieties)); // varieties 57 number

let isFresh = true;
console.log('isFresh', isFresh, typeof(isFresh)); // isFresh true boolean

let total;
console.log('total', total, typeof(total)); // total undefined undefined
```

14. Multiple variables, separated by commas, can be declared in **one** line of code. Declare 3 variables with just one instance of the **let** keyword:

```
let x = 1, y = 2, z = 3;
console.log(x + y * z); // 7
```

15. Such "one-liner" variable declarations are more common when the vars that are not to be assigned an initial value. Declare 4 **undefined** (no value) variables with just one **let**:

```
let day, date, month, year;
console.log(day, date, month, year);
// undefined undefined undefined undefined
```

Undefined variables are **not** errors, but some programmers prefer to avoid them, anyway. As a workaround, you can assign a starter value with the intention of changing it later. This has the advantage of indicating the datatype.

16. Declare a string and a number, with starter values of empty string **""** and **0**:

```
let grade = "", score = 0;
```

---

declaring a variable equal to another variable

If you set a variable equal to another variable, the "copy" is its own independent entity. If you change the value of the "copy", the "original" remains unchanged.

17. Declare a variable, and set it equal to score from the previous step:

```
let greeting = "Hola";
let greeting2 = greeting;
greeting2 = "Howdy";
console.log(greeting2); // "Howdy"
console.log(greeting); // "Hola"
```

01.01 Lab

Lab Instructions:

- Open **01.01-Lab.html** in your Code Editor.
- Write your code in the **script** tags, where starter code is provided.
- Debug the following variables.
- Your solutions must be variables of the indicated **datatype**
- Wherever you see **VAR**, replace it with your variable.

Example:

Question 0:

0. **type: string**

```
// let first-name = Bob;
// console.log('0.', 'VAR', VAR, typeof(VAR));
```

Solution:

0. **type: string**

- Remove the hyphen from the variable name.
- Put the value of the string in quotes: "Bob" or 'Bob'.

```
let firstName = 'Bob';
console.log('0.', 'firstName', firstName, typeof(firstName));
// 0. firstName Bob string
```

On your own:

1. **type: number**

```
let admission Fee = $10;
console.log('1.', 'VAR', VAR, typeof(VAR));
```

2. **type: string**

```
let #1sportsCar = "Porsche";
console.log('2.', 'VAR', VAR, typeof(VAR));
```

3. **type: boolean**

```
let is Online = FALSE;
console.log('3.', 'VAR', VAR, typeof(VAR));
```

4. **type: number**

```
let %done = 22.5%;
console.log('4.', 'VAR', VAR, typeof(VAR));
```

5. **type: undefined**

```
let password "&123b45d";
console.log('5.', 'VAR', VAR, typeof(VAR));
```

6. **type: string**

```
let 26miles = marathon;
console.log('6.', 'VAR', VAR, typeof(VAR));
```

7. **type: string**

```
```

```
let $100000Bar = "candy bar";
console.log('7.', 'VAR', VAR, typeof(VAR));
```

8. **type: number**

```
let firstPrize = 7,500;
console.log('8.', 'VAR', VAR, typeof(VAR));
```

9. **type: boolean**

```
let i_Won! = True;
console.log('9.', 'VAR', VAR, typeof(VAR));
```

10. **type: undefined**

```
let greeting = Hola;
console.log('10.', 'VAR', VAR, typeof(VAR));
```

01.01 Lab Solution

---

1. **type: number**

- Variable names cannot have spaces.
- Value must be a number, so it cannot be $10, which, is a string requiring quotes: '$10'.

```
// let admission Fee = $10;
let admissionFee = 10;
console.log('1:', 'admissionFee', admissionFee, typeof(admissionFee));
```

2. **type: string**

- Variable names cannot have special characters (other than _ and $).

```
// let #1sportsCar = "Porsche";
let sportsCar1 = "Porsche";
console.log('2:', 'sportsCar1', sportsCar1, typeof(sportsCar1));
```

3. **type: boolean**

- Variable names cannot have spaces.
- Boolean values (true, false) are lowercase.

```
// let is Online = FALSE;
let isOnline = false;
console.log('3:', 'isOnline', isOnline, typeof(isOnline));
```

4. **type: number**

- Variable names cannot have special characters (other than _ and $).
- 22.5% percent as a decimal is .225, not 22.5

```
// let %done = 22.5%;
let pctDone = .225;
console.log('4:', 'pctDone', pctDone, typeof(pctDone));
```

5. **type: undefined**

- A variable without a value is undefined, so remove the value.

```
let password = "&123b45d";
let password;
console.log('5:', 'password', password, typeof(password));
```

6. **type: string**

- Variable names cannot start with a number.
- String values go in single or double quotes.

```
let 26miles = marathon;
let miles26 = "marathon";
console.log('5:', 'password', password, typeof(password));
```

7. **type: string**

- The $ is allowed, so the variable name is fine, as is.
- String values go in single or double quotes.

```
// let $100000Bar = candy bar;
let $100000Bar = "candy bar";
console.log('7:', '$100000Bar', $100000Bar, typeof($100000Bar));
```

8. **type: number**

- Numeric values cannot have commas.

```
// let firstPrize = 7,500;
let firstPrize = 7500;
console.log('8:', 'firstPrize', firstPrize, typeof(firstPrize));
```

9. **type: boolean**

- i_Won is legal, although iWon (camel-case) is better
- Variable names cannot include !
- boolean values (true, false) are lowercase

```
// let i_Won! = True;
let iWon = true;
console.log('9:', 'iWon', iWon, typeof(iWon));
```

10. **type: undefined**

- A variable without a value is undefined, so remove the value.

```
// let greeting = Hola;
let greeting;
console.log('10:', 'greeting', greeting, typeof(greeting));
```

- NaN, Number(), null
- undefined vs. not defined
- string concatenation

---

number-like string**

A "number-like string" is a number enclosed in quotes. So, whereas 5 is an actual number, "5" is a "number-like string".

1. Declare a variable with a "number-like string" value, and try to do addition with it:

```
let bill = 50;
let tip = '10';
let total = bill + tip;
console.log(total); // 5010
console.log(bill * tip); // 500
console.log(bill / tip); // 5
console.log(bill - tip); // 40
```

Addition with number-like strings fails, because the plus-sign defaults to concatenation. But for other operations with number-like strings -- subtraction, multiplication, division -- the math works, because there is no plus-sign to confuse things.

2. Declare a number-like string and do division with it:

```
let pizzas = "4";
let people = 8;
let pizzasPP = pizzas / people;
console.log(pizzasPP); // 0.5
```

NaN (Not a Number)

NaN (Not a Number) results from trying to do math with something that is neither a number nor a number-like string.

3. Try to do math with a price that includes a dollar sign:

```
let fullPrice = '$80';
let halfPrice = fullPrice * 0.5;
console.log('halfPrice', halfPrice); // NaN number
```

The string '$80' is in no way understood as 80, the number, so attempting to do math with '$80' fails.

Number() method

The **Number()** method takes a variable as its argument, and where possible, returns a number. It is ideal for converting "number-like strings" into actual numbers.

4. A number in quotes such as "4"` is a "number-like string". Convert it to a number:

```
console.log('pizzas', pizzas, typeof(pizzas));
// pizzas 4 string
pizzas = Number(pizzas);
console.log('pizzas', pizzas, typeof(pizzas));
// pizzas 4 number
```

If the string passed to it cannot be converted to a number, the **Number()** method returns **NaN**.

5. Try to convert "banana" to a number:

```
let fruit = "banana";
let baNaNa = Number(fruit);
console.log('baNaNa', baNaNa, typeof(baNaNa));
// baNaNa NaN number
```

Despite its name, **NaN** has a data type of number.

"Number-like strings" can't be used for addition, but you can convert them with the Number() method.

6. Convert "15" to an actual number, so that it can be used for addition:

```
bill = 70;
tip = '15';
total = bill + tip;
console.log(total); // 7015
total = bill + Number(tip);
console.log(total); // 85
```

not defined vs undefined

not defined means the variable does not exist. It is an error that usually arises from typos.

undefined means that the variable exists, but has no value.

7. To see the difference between 'undefined' and 'not defined', declare a variable with no value, and then misspell it:

```
let island;
console.log(island, typeof(island)); // undefined undefined
island = "Bali";
console.log(island, typeof(island)); // Bali string
// console.log(ixland); // error: ixland is not defined
```

---

null

null and **undefined** are both *falsey* (return false in a boolean context), but null is an actual value assigned to a variable. It has a data type of object, but it's just that null is an *empty* object.

8. Declare a variable, set it to null, and log it:

```
let user = null;
console.log('user', user, typeof(user)); // user null object
```

---

string concatenation

Variables and substrings can be joined together with plus-signs (**+**) to make one bigger string. The procedure is known as **string concatenation**.

9. Concatenate the **topic** variable with substrings:

```
let topic = "JavaScript";
let intro = "Let's learn " + topic + "!";
console.log(intro); // Let's learn JavaScript!
```

10. Concatenate with two variables:

```
let firstName = 'Brian';
let lastName = 'McClain';
let greeting = 'Hello, class! My name is ' + fullName + ' ' + lastName + '.';
console.log(greeting);
// Hello, class! My name is Brian McClain.
```

Concatenation is often used for making multiple versions of similar strings, such as a set of image paths that differ only slightly by file name.

11. Concatenate an image file path consisting of two variables and three substrings:

```
let kind = 'Jack';
let suit = 'Hearts';
let imgPath = 'images/' + kind + '-of-' + suit + '.jpg';
console.log(imgPath); // images/Jack-of-Hearts.jpg
```

12. Change the variable values to concatenate other image paths:

```
kind = 'Queen';
suit = 'Diamonds';
imgPath = 'images/' + kind + '-of-' + suit + '.jpg';
console.log(imgPath); // images/Queen-of-Diamonds.jpg

kind = 'King';
suit = 'Clubs';
imgPath = 'images/' + kind + '-of-' + suit + '.jpg';
console.log(imgPath); // images/King-of-Clubs.jpg

kind = 'Ace';
suit = 'Spades';
imgPath = 'images/' + kind + '-of-' + suit + '.jpg';
console.log(imgPath); // images/Ace-of-Spades.jpg
```

A plus-sign performs concatenation rather than addition if the expression includes a string.

13. Try adding numbers with a dollar-sign present. It reverts to string concatenation:

```
let food = 25;
let bev = 15;
let tip = 8;
let tot = 'Please Pay: $' + (food + bev + tip);
console.log(tot, typeof(tot)); // $25158 string
```

14. Remove the dollar sign, and run it again; this time the math works. Once the math is done, concatenate the dollar sign back on:

```
tot = food + bev + tip;
console.log(tot); // 48
console.log(tot, typeof(tot)); // 48 number
tot = '$' + tot;
console.log(tot, typeof(tot)); // $48 string
```

---

changing a variables's data type

Changing the datatype of a variable is possible, but it should make sense to do so, as it does in the above $48 example. Let's look at an example where it would make no sense to change a variable's data type:

15. Change `pet` from a string to a number. This makes no sense:

```
let pet = "cat";
pet = 255;
```

16. Change a three digit integer into a four-digit PIN by concatenating a leading zero. This result is a string, since a non-zero integer cannot start with a zero. The data type has changed from number to string, but in this case, the reason for changing datatypes make sense:

```
let num = 582;
console.log(num, typeof(num)); // 582 number
// make number 582 into the PIN 0582
let pin = "0" + num;
console.log(pin, typeof(pin)); // 0582 string
```

01.01 Lab

Lab Instructions:

- Open **01.02-Lab.html** in your Editor. Write the Lab code in the **script** tags.

1. Assign values to the provided variables and use **typeof()** to log output that matches the comment:

```
// example:
var flavor;
console.log('?'); // vanilla string

// solution:
var flavor = 'vanilla';
console.log(flavor, typeof(flavor));

// on your own:
var msg;
console.log('?'); // Hello World! string

var total;
console.log('?'); // 314 number

var isFresh;
console.log('?'); // true boolean

var price;
console.log('?'); // price: $ 29.95 number

var player1;
console.log('?'); // player1: undefined 'undefined'

var restaurantBill = 80;
var groupon = '$10';
var pleasePay;
console.log(pleasePay); // NaN 'number'
```

2. Use the Number() method so that the console.log output changes from NaN to matching the comment.

```
// example:
var x = '55';
var y = 6;
var z;
console.log(z); // expected: 330

// solution:
var x = '55';
var y = 6;
var z = Number(x) * y;
console.log(z); // 330

// on your own:
var x = '55';
var y = '33';
var z;
console.log(z); // expected: 22

var x = 55;
var y = '33';
var z;
console.log(z); // expected: 88

var x = '55';
var y = 5;
var z;
console.log(z); // expected: 11

var x = 55;
var y = '44';
var z;
console.log(z); // expected: 5544
```

string concatenation

Assign values to the variables and then use the variables in string concatenation to reproduce the **_messages shown_**:

3. message 1: Sally lives in New York City.

```
let msg1;
let name;
let city;
console.log(msg1);
```

4. message 2: Freddy went to France for 6 months.

```
let msg2;
let fName;
let country;
let months;
console.log(msg2);
```

5. message 3: My Bunny had a baby carrot for breakfast.

```
let msg3;
let pet;
let veg;
let meal;
console.log(msg3);
```

01.02 Lab Solution

---

1. Assign values to the provided variables and use typeof() to log output that matches the comment:

On your own:

```
```js
var msg = 'Hello World';
console.log(msg, typeof(msg)); // Hello World! string

var total = 314;
console.log(total, typeof(total)); // 314 number

var isFresh = true;
console.log(isFresh, typeof(isFresh)); // true boolean

var price = 29.95;;
console.log('price: $', price, typeof(price)); // price: $ 29.95 number

var player1;
console.log(player1, typeof(player1)); // undefined 'undefined'

var restaurntBill = 80;
var groupon= '$10';
var pleasePay = restaurantBill - groupon;
console.log(pleasePay, typeof(pleasePay)); // NaN 'number'
```
```

2. Use the Number() method so that the console.log output changes from NaN to matching the comment.

On your own:

```
var x = '55';
var y = '33';
var z = Number(x) - Number(y);
console.log(z); // 22
```

```
var x = '55';
var y = '33';
var z = Number(x) + Number(y);
console.log(z); // 88

var x = '55';
var y = 5;
var z = Number(x) / y;
console.log(z); // 11

var x = 55;
var y = '44';
var z = x + y;
console.log(z); // 5544
```

string concatenation

Assign values to the variables and then use the variables in string concatenation to reproduce the ***messages shown***:

3. message 1: Sally lives in New York City.

```
let msg1;
let name = "Sally";
let city = "New York City";
msg1 = name + " lives in " + city + "."
console.log(msg1);
```

4. message 2: Freddy went to France for 6 months.

```
let msg2;
let fName = "Freddy";
let country = "France";
let months = 6;
msg2 = fName + " went to " + country + " for " + months + " months.";
console.log(msg2);
```

5. message 3: My Bunny had a baby carrot for breakfast.

```
let msg3;
let pet = "Bunny";
let veg = "baby carrot";
let meal = "breakfast";
msg3 = "My " + pet + " had a " + veg + " for " + meal + ".";
console.log(msg3);
```

- resoloving nested quotes conflicts
- string interpolation with backticks ``
- const for declaring constants

---

resolving conflicting nested quotes

Strings can be enclosed in either double or single quotes, but nested quotes can result in erros if JS "gets confused" over where the ending quote of the string is supposed to occur.

1. Declare a string in single quotes with an internal single quote (apostrophe). Check the console. There's an error:

```
let song = 'Won't Get Fooled Again';
// Unexpected identifier 't'
```

---

unexpected identifier

An **unexpected identifier**, also known as an **unexpected token**, is a character that doesn't belong. The preceding error resulted from the apostrophe being mistaken for the end of the string. JS saw 'Won' as the entire string--and everything else as 'unexpected'.

---

mixing-and-matching quotes

If you have an inner single quote (apostrophe), avoid conflicts by wrapping the string in double quotes.

2. Resolve the nested quote conflict by switching to double quotes:

```
let song = "Won't Get Fooled Again";
console.log(song); // Won't Get Fooled Again
```

---

escaping quotes

Instead of mixing-and-matching quotes, you can "escape quotes" with a backslash, which causes the punctuation to be ignored.

3. Declare another variable with single quotes (apostrophes) nested inside single quotes, and then fix the error by escaping the apostrophes:

```
// song = 'Don't Stop Believin''; // error
song = 'Don\'t Stop Believin\'';
console.log(song);
```

string interpolation

Long string concatenation can get hard to read, with all its quote and plus-signs. A cleaner alternative is **string interpolation**, which has no quotes or plus signs. To convert concatenation to interpolation, follow these steps:

- Delete all quotes and plus signs.
- Wrap everything in a pair of backticks: ``
- Put variables in curly braces {}, with a dollar-sign in front: ${variable}: **${variable}**

4. Do a string concatenation, and then switch to string iterpolation:

```
// string concatenation
let firstName = 'Bob';
let greeting = 'Hi, ' + firstName + '! How are you?';
console.log(greeting); // Hi, Bob! How are you?

// string iterpolation
greeting = Hi, ${firstName}! How are you?;
console.log(greeting); // Hi, Bob! How are you?
```

The advantage of string interpolation is more apparent when there are multiple variables and substrings.

5. Declare a few more variables and make a longer, more complex concatenation:

```
let petName = "Fluffy";
let age = 3;
let food = "tuna";
let petSound = "Meow";

let catGreeting = petSound + "! My name is " + petName + "! I am " + age + " years old. My favorite food is " + food + ".";

console.log(catGreeting);
// Meow! My name is Fluffy! I am 3 years old. My favorite food is tuna.
```

This concatenation has four pairs of quotes and six plus signs, making it hard to read and write.

6. Refactor the above as string interpolation:

```
catGreeting = `${petSound}! My name is ${petName}. I am ${age} years old. My favorite food is ${food}`;

console.log(catGreeting);
// Meow! My name is Fluffy! I am 3 years old. My favorite food is tuna.
```

line breaks and tabs

With string concatenation, hitting enter to get a line break is ignored, as are tabs.

7. Do a string concatenation with line breaks and tabs:

```
let item1 = 'eggs';
let item2 = 'bread';
let item3 = 'milk';
let shoppingList = 'Grocery List:' +
   item1 +
   item2 +
   item3;
console.log(shoppingList);
// Shopping List:eggsbreadmilk
```

As the squished output indicates, the line breaks and tabs were ignored.

8. Try again, but this time include **\n** (new line)and **\t** (tab) symbols. Now it works:

```
list = 'Grocery List:\n\t' + item1 + '\n\t' + item2 +  '\n\t' + item3;
console.log(list);
/*
Shopping List:
   eggs
   bread
   milk
*/
```

- `\n` in string concatenation gives a line break.
- `\t` in string concatenation gives a tab. These symbols only work in the console, however. For web output, use tags for line breaks and or CSS margin / padding for extra space

9. Now, try it with string interpolation. Hitting enter gives you a line break. Htting tab works too, so no need for those pesky \n and \t symbols.

```
shoppingList = `Grocery List:
   ${item1}
   ${item2}
   ${item3}`;
console.log(shoppingList);
/*
Shopping List:
   eggs
   bread
   milk
*/
```

---

const means constant

- A **variable** is a named value that can vary (change).

- A **constant** is a named value that cannot change.

Constants, by convention, have UPPERCASE names, but an all-caps name alone does not make for a constant. Below, **let MOON_DIAM** is uppercase, but you can still change the value.

10. Declare an uppercase variable with **let**. Use uppercase. Since camel case is moot, use an underscore to provide separation between the words:

    ```
    let MOON_DIAM = 2159.1;
    console.log(MOON_DIAM); // 2159.1
    MOON_DIAM *= 10; // multiply by 10 (shorthand for MOON_DIAM = MOON_DIAM * 10)
    console.log(MOON_DIAM); // 21591
    ```

To declare a true constant, use the **const** keyword. As its name implies, **const** makes true constants. If you try to change it, you get an error.

11. Declare a true constant with **const**:

    ```
    const DOZEN = 12;
    DOZEN = 13; // Error: Assignment to constant variable
    ```

12. You cannot change a const, so it gives you an error. Comment it out:

    ```
    // DOZEN = 13;
    ```

13. Try redeclaring DOZEN. It won't work, not even if you change to **let** or **var**. It gives an error, so comment it out:

    ```
    // const DOZEN = 13; // SyntaxError: Identifier DOZEN has already been declared.
    ```

01.03 Lab

Instructions:

- Open **01.03-Lab.html** in your Editor. Write the Lab code in the **script** tags.

1. Debug these strings by mixing-and-matching quotes.

```
var cereal = 'Cap'n Crunch';
console.log(cereal);  // Cap'n Crunch

var questi = ""Who's there?" she asked.";
console.log(questi); // "Who's there?" she asked.
```

2. Debug these strings by escaping quotes with backslash:

```
var phrase = 'Isn't that amazing!';
console.log(phrase); // 'Isn't that amazing!

var quote = ""Show me the money!" is a famous movie line.";
console.log(quote); // "Show me the money!" is a famous movie line.
```

3. Use string interpolation to produce the messages in the comments:

```
var pet = 'dog';
var name = 'Fido';
var toy = 'frisbee';
var msg;
console.log(msg); // My dog Fido's favorite toy is a frisbee.

var animal;
var continent;
var msg;
console.log(msg); // The kangaroo is native to Australia.
```

4. Produce the restaurant bill output as shown in the multi-line comment, below. This one requires you to combine math operations, which you practiced earlier, with string interpolation:

```
var joint = "BOB'S DINER";
var food = 30;
var bev = 20;
var subTotal = food + bev;
var taxRate = 0.08; // 8%
var tipPct = 0.2; // 20%;
var tax;
var tip;
var grandTotal;
var guestCheck;
console.log(guestCheck);
/*
BOB'S DINER
GUEST CHECK:
Food: $30
Beverage: $20
Sub-Total: $50
Tax: $4
Tip: $10
TOTAL: $64
THANK YOU!
*/
```

01.03 Lab Solution

Instructions:

- Open **01.03-Lab.html** in your Editor. Write the Lab code in the **script** tags.

1. Debug these strings by mixing-and-matching quotes.

```
let cereal = "Cap'n Crunch";
console.log(cereal); // Cap'n Crunch

let questi = '"Who\'s there?" she asked.';
console.log(questi); // "Who's there?" she asked.
```

2. Debug these strings by escaping quotes with backslash:

```
let phrase = 'Isn\'t that amazing!';
console.log(phrase); // Isn't that amazing!

let quote = "\"Show me the money!\" is a famous movie line.";
console.log(quote); // "Show me the money!" is a famous movie line.
```

3. Use string interpolation to produce the messages in the comments:

```
let pet = 'dog';
let name = 'Fido';
let toy = 'frisbee';
let msg = `My ${pet} ${name}'s favorite toy is a ${toy}.`;
console.log(msg); // My dog Fido’s favorite toy is a frisbee.

let animal;
let continent;
let msg = `The ${animal} is native to ${continent}.`;
console.log(msg); // The kangaroo is native to Australia.
```

4. Produce the restaurant bill output as shown in the multi-line comment, below. This one requires you to combine math operations, which you practiced earlier, with string interpolation:

```
let joint = "JS CAFE";
let food = 30;
let bev = 20;
let subTotal = food + bev;
let taxRate = 0.08; // 8%
let tipPct = 0.2; // 20%;
let tax = subTotal * taxRate;
let tip = subTotal * tipPct;
let grandTotal = subTotal + tax + tip;
let guestCheck = `*** ${joint} ***
*** GUEST CHECK ***
    Food: $${food}
    Beverage: $${bev}
    Sub-Total: ${subTotal}
    Tax: ${tax}
    Tip: ${tip}
    TOTAL: $${grandTotal}
*** THANK YOU! ***`;
console.log(guestCheck);

/*
    *** JS CAFE ***
    *** GUEST CHECK ***
    Food: $30
    Beverage: $20
    Sub-Total: $50
    Tax: $4
    Tip: $10
    TOTAL: $64
    *** THANK YOU! ***
*/
```

- Mathematical Operators: +, -, *, /, **, +=, -=, *=, /=, ++, --
- Math.random(), Math.round(), Math.floor(), Math.ceil()
- Math.max(), Math.min(), Math.abs(), Math.pow(), Math.sqrt(), Math.PI
- Spread Operator: ...
- toFixed() for rounding a number to n digits

Naturally, we can do math with numeric variables.

Mathematical operators

```
+   add
-   subtract
*   multiply
/   divide
**  exponent (power of)
%   modulo (remainder)
```

1. Declare two number variables, and use them for some simple calculations:

```
let n1 = 10;
let n2 = 3;

console.log(n1 + n2); // 13
console.log(n1 - n2); // 7
console.log(n1 * n2); // 30
console.log(n1 / n2); // 3.333333333
console.log(n1 ** n2); // 1000 (10x10x10)
console.log(n1 % n2); // 1 (remainder of 10/3)
```

Results of mathematical calculations can be stored in variables.

2. Change the values of **n1** and **n2**, and do some more calculations, saving the results to variables:

```
n1 = 6;
n2 = 2;

let sum = n1 + n2;
console.log(sum); // 8

let diff = n1 - n2;
console.log(diff); // 4

let prod = n1 * n2;
console.log(prod); // 12

let quot = n1 / n2;
console.log(quot); // 3

let expon = n1 ** n2;
console.log(expon); // 36

let mod = n1 % n2;
console.log(mod); // 0
```

Order of Operations of Mathematical Expressions

- Do * and / before + and -
- Do * and / from left to right
- Do + and - from left to right
- Do exponents ** before * or /
- Do anything inside () first

3. Declare a third number variable, and do some math that shows how parentheses can affect the result:

```
n1 = 4;
n2 = 5;
let n3 = 8;

let tot = n1 + n2 * n3; // 4 + 40
console.log(tot); // 44

tot = (n1 + n2) * n3; // 9 * 8
console.log(tot); // 72
```

Math Object

JS has a built-in Math Object, which comes with many useful methods:

Math.random() generates a random float from 0-1, to 16 decimal points:

4. Generate a random number and log it:

```
let r = Math.random();
console.log(r); // 0.7492906781140873
```

Math.round() rounds off its argument to the nearest integer

5. Round off a number:

```
let x = Math.round(2.5);
console.log(x); // 3
```

Math.floor() rounds *down* its argument:

6. Round down a number:

```
let y = Math.floor(2.9999999);
console.log(y); // 2
```

Math.ceil() rounds *up* its argument:

7. Round up a number:

```
let z = Math.ceil(2.0000001);
console.log(z); // 3
```

random numbers

Math.random() generates a random float from 0-1, so to get a larger number, just multiply by some value.

8. Generate a random number and multiply it by 100:

```
let rando = Math.random() * 100;
console.log(rando)  // some number between 0-100
```

To get an integer, round, floor or ceil the random value.

9. Round down a random number multiplied by 100:

```
let randInt = Math.floor(Math.random() * 100);
console.log(randInt); // some integer between 0-99
```

To get a random integer in a range, multiply by the range span and then add the starting value.

10. Round up a random number multiplied by 50 and then add 50 to get a number in the 50-100 range:

```
let rand = Math.ceil(Math.random() * 50 + 50);
console.log(rand); // some value between 50-100
```

---

max(), min(), pow(), abs(), sqrt(), PI

Math.max() returns the greatest of the multiple values passed to it:

11. Find the maximum of a set of numbers:

```
let maxi = Math.max(3, 6, 8, 2, 12, 4, 10);
console.log(maxi); // 12
```

Arrays

Arrays are a topic for Unit 4, but here is a sneak preview. Arrays are variables that store multiple values inside square brackets, with each item separated by a comma.

12. Declare an array of numbers:

```
let nums = [3,6,8,2,12,4,10];
```

13. Pass the array to the Math.max() method, saving the result to a variable. Then log the variable:

```
let numsMax = Math.max(nums);
console.log('numsMax', numsMax); // NaN
```

It doesn't work. When we log the result we get NaN. This is because Math.max() requires a comma-separated listing of values. The numbers cannot be bundled up into an array.

Spread Operator ...

The spread operator ... in front of an array "destructures" the array, which means it essentially deletes the square brackets, leaving just the numbers.

14. Use the spread operator to destructur the array passed to Math.max(). This time it works.

```
let maximum = Math.max(...nums);
console.log('maximum', maximum); // 12
```

Math.min() returns the smallest of the multiple values passed to it.

15. Find the minimum of numsArr. As with Max.max() we need to use the spread operator to destructure the array:

```
let mini = Math.min(...nums);
console.log('mini', mini); // 2
```

Math.pow() takes two arguments: a number and a power to raise it to:

16. Raise a number to a power using the **Math.pow()** method:

```
let pwr = Math.pow(5, 4);
console.log(pwr); // 625 (5x5x5x5)
```

However, as we have seen, a ** b does the same thing as Math.pow(a,b)

17. Raise a number to a power using the ** operator:

```
let powr = 5 ** 4;
console.log(powr); // 625 (5x5x5x5)
```

Math.abs() returns the absolute value of its argument, meaning it makes it positive:

18. Use **Math.abs()** to get the absolute value of a negative number:

```
let absolut = Math.abs(-7);
console.log(absolut); // 7
```

Math.sqrt() returns the square root of its argument:

19. Use **Math.sqrt()** to find a square root:

```
let sqRt = Math.sqrt(81);
console.log(sqRt); // 9
```

Math.PI returns the famous constant. If you save it, it should be to a **const**, uppercase:

20. Get **PI** to 16 digits:

```
const PI = Math.PI;
console.log(PI); // 3.141592653589793
// PI = 'apple'; // ERROR
```

---

const means constant

const means constant so you cannot change the value.

21. Declare a constant and then try to change the values It gives you an error:

```
const NY_CAPITOL = "Albany";
// NY_CAPITOL = "Syracuse"; //Error: Assignement to constant variable
```

---

redeclaring with var let

As we have seen, with let you can change the datatype but not redeclare the variable.

22. Declare a let string and then change the data type to number. It works:

```
let pet = "Iguana";
console.log(pet); // Iguana
pet = 9;
console.log(pet); // 9
```

23. But you cannot redeclare a let. Try redeclaring pet, but then comment it out to get rid of the error:

```
// let pet = "Bunny";
// console.log(pet); // Error: pet has already been declared
```

24. With var, the old way of declaring a variable, you can redeclare the variable. Try it:

```
var car = "Chevy";
console.log(car); // Chevy
var car = "Dodge";
console.log(car); // Dodge
```

---

rounding off a number to N decimal places

toFixed() is a method called on a float. It returns a float with the number of decimal places in the argument:

25. Round PI to 2 digits. The rounded value is actually a string:

```
let pi2 = PI.toFixed(2);
console.log(pi2, typeof(pi2)); // 3.14 string
```

26. Try doing addition with **pi2**. The plus sign does concatenation, because it's working with a string:

```
let piPlusPi = pi2 + pi2;
console.log(ppiPlusPi); // 3.143.14
```

27. Round PI to 2 decimal places, but keep it as a number by passing PI.toFixed(2) to the Number() method:

```
let pi2num = Number(PI.toFixed(2));
console.log('pi2num', pi2num, typeof(pi2num));
// pi 3.14 number
```

parseFloat() method also converts a stringy number to actual number.

28. Pass the PI.toFixed(3) to parseFloat() so that the 3-digit PI stays a number:

```
let pi3num = parseFloat(PI.toFixed(3));
console.log('pi3num', pi3num, typeof(pi3num));
// pi 3.142 number
```

---

math shorthand operators: += -= *= /=

Math shorthand operators make math more concise by eliminating the need to repeat a variable:

29. Try these math shorthand operators:

```
x = 20;
console.log(x); // 20

x = x + 35; // addition
console.log(x); // 55

x += 15; // addition shorthand
console.log(x); // 70

x = x * 3; // multiplication
console.log(x); // 210

x *= 2; // multiplication shorthand
console.log(x); // 420

x = x - 80; // subtraction
console.log(x); // 340

x -= 100; // subtraction shorthand
console.log(x); // 240

x = x / 4; // division
console.log(x); // 60

x /= 3; // division shorthand
console.log(x); // 20
```

01.04 Lab

Instructions:

- Open **01.04-Lab.html** in your Editor. Write the Lab code in the **script** tags.

1. Set the **y** value to produce console output that matches the comment:

```
// Example:
let x = 20;
let y;

// y = ?;
console.log(x + y); // 23

// Solution:
x = 20;
y = 3;
console.log(x + y); // 23

// On your own:
x = 10;
y;

// y = ?;
console.log(x + y); // -10

// y = ?;
console.log(x - y); // -10

// y = ?;
console.log(x * y); // 2.5

// y = ?;
console.log(x / y); // 2.5

// y = ?;
console.log(x ** y); // 100000
```

```
// y = ?;
console.log(x % y); // 1
```

2. Calculate the total cost, as shown in the comment:

```
let unitCost = 50;
let numUnits = 12;
let shipping = 25;
let totalCost;
console.log(totalCost); // 625
```

3. Calculate the values of z in terms of w, x and y to get the value shown in the comment:

```
let w = 10;
x = 12;
y = 15;
let z;
console.log(z); // 190
console.log(z); // 105
console.log(z); // 18
console.log(z); // 8
console.log(z); // 7
```

4. Use the Math Object to generate a random integer between 10-19;

5. Use the Math Object to generate a random integer between 26-50;

6. Use the Math Object to get the maximum value from these numbers: 3, 5, 21, 7, 1, 3, 12, 6

7. Use the Math Object to get the square root of 81.

8. Use a Math Object method that takes 9.9999 as its argument and returns 9.

9. Supply values for x, y and expon so that the console.log output matches the comment next to it:

```
x;
y;
```

```
let expon;
console.log(expon); // 196
```

10. The area of a circle equals PI times the radius squared: **$A = \pi r^2$**. Given a circle of radius 4, use the Math Object to find the area of the circle.

11. The hypotenuse (c) of a right triangle is obtained by the formula: **$a^2+b^2=c^2$**, where a and b are the other two sides. Using the Math Object, find the hypotenuse of a triangle, where a=5 and b=12.

12. Generate two random floats, r1 and r2, in the 0-10 range. Round each of them off to 5 decimal places. Add them together. HINTS: You cannot do addition with strings. Use the Number() method to convert strings to numbers.

13. Given this baseball player and his statistics:

```
/*
Player:     Vladimir Guerrero Jr.
Team:       Toronto Blue Jays
Year:       2021

Stats:
PA: 698
AB: 604
R:  123
H:  188
2B: 29
3B: 1
HR: 48
RBI: 111

Calculate Guerrero's Slugging Percentage (SLG), which equals total
bases (TB) divided by at bats (AB): SLG = TB / AB

Total bases is the sum of a player's hits (H), plus their  doubles
(2B), plus twice their triples (3B), plus three times their home runs
(HR): TB = H + 2B + (3B * 2) + (HR * 3)

EXAMPLE:

A player has 100 hits, including 17 2B, 4 3B and 19 HR:
100 + 17 + (4 * 2) + (19 * 3) = 100 + 17 + 8 + 57 = 182 TB
```

```
It took the player 350 AB to amass these 182 TB. Therefore, they have
a SLG of: 100/350 = 0.520
*/
```

01.04 Lab Solution

Instructions:

- Open **01.04-Lab-Solution.html** in your Editor. Check your Lab code against the solution found in the **script** tags.

1. Set the **y** value to produce console output that matches the comment:

```
let x = 10;
let y = 0;

y = -20;
console.log(x + y); // -10

y = 20;
console.log(x - y); // -10

y = .25;
console.log(x * y); // 2.5

y = 4;
console.log(x / y); // 2.5

y = 5;
console.log(x ** y); // 100000

y = 3;
console.log(x % y); // 1
```

2. Calculate the total cost, as shown in the comment:

```
let unitCost = 50;
let numUnits = 12;
let shipping = 25;
let totalCost = unitCost * numUnits + shipping;
console.log(totalCost); // 625
```

3. Calculate the values of z in terms of w, x and y to get the value shown in the comment:

```
let w = 10;
x = 12;
y = 15;
let z;

z = x * y + w;
console.log(z); // 190

z = w * x - y;
console.log(z); // 105

z = x * y / w;
console.log(z); // 18

z = w + x - y;
console.log(z); // 8

z = w * x / y;
console.log(z); // 7
```

4. Use the Math Object to generate a random integer between 0-19;

```
let r = Math.floor(Math.random() * 20);
console.log(r);
```

5. Use the Math Object to generate a random integer between 26-50;

```
let ran = Math.ceil(Math.random() * 25 + 25);
console.log(ran);
```

6. Use the Math Object to get the maximum value from these numbers: 3, 5, 21, 7, 1, 3, 12, 6

```
let m = Math.max(3, 5, 21, 7, 1, 3, 12, 6);
console.log(m); // 21
```

7. Use the Math Object to get the square root of 81.

```
let sqR = Math.sqrt(81);
console.log(sqR); // 9
```

8. Use a Math Object method that takes 9.99 as its argument and returns 9.

```
let fl = Math.floor(9.99);
console.log(fl); // 9
```

9. Supply values for x, y and expon so that the console.log output matches the comment next to it:

```
x = 14;
y = 2;
let expon = x ** y;
console.log(expon); // 196
```

10. The area of a circle equals PI times the radius squared: $A = \pi r^2$. Given a circle of radius 4, use the Math Object to find the area of the circle.

```
// area = π r²
let area = Math.PI * Math.pow(4, 2);
console.log(area); // 50.26548245743669

// OR use the ** operator:
area = Math.PI * (4 ** 2);
console.log(area); // 50.26548245743669
```

11. The hypotenuse (c) of a right triangle is obtained by the formula: $a^2+b^2=c^2$, where a and b are the other two sides.

```
// Using the Math Object, find the hypotenuse of a triangle, where a=5
and b=12.

// a² + b² = c²
let a = 5;
let b = 12;
// let c2 represent the square of the hypotenuse
let c2 = (5 ** 2) + (12 ** 2);
let c = Math.sqrt(c2);
console.log(c);
```

12. Generate two random floats, r1 and r2, in the 0-10 range. Round each of them off to 5 decimal places. Add them together.

```
let r1 = Math.random() * 10;
let r2 =Math.random() * 10;

// round to 3 decimal places
r1 = r1.toFixed(3);
r2 = r2.toFixed(3);
console.log(r1); // 9.202
console.log(r2); // 6.845

// HINTS: You cannot do addition with strings.

let sum = r1 + r2; // 9.2026.845 -- Oops! String concatenation!

// Try again, first converting r1 and r2 back to numbers:
r1 = Number(r1);
r2 = Number(r2);

sum = r1 + r2;
console.log(sum); // 16.047
```

13. Given this baseball player and his statistics:

```
/*
Player:     Vladimir Guerrero Jr.
Team:       Toronto Blue Jays
```

```
Year:       2021

Stats:
PA: 698
AB: 604
R:  123
H:  188
2B: 29
3B: 1
HR: 48
RBI: 111

We don't need all the stats, but make vars of the ones we do need:
*/
let AB = 604;
let H = 188;
let _2B = 29;
let _3B = 1;
let HR = 48;

// Calculate Total Bases (TB):
let TB = H + _2B + (_3B * 2) + (HR * 3);
console.log(TB); // 363

// Calculate Slugging Percentage (SLG):
let SLG = TB / AB; // 363 / 604
console.log(SLG); // 0.6009933774834437

// Round to 3 decimal places
SLG = SLG.toFixed(3);
console.log(SLG); // 0.601

// EXTRA CREDIT: Drop the leading zero
SLG = SLG.toString().slice(1);

EXAMPLE / HINT (but only if you need it)

A player has 100 hits, including 17 2B, 4 3B and 19 HR, they have 182
TB: 100 + 17 + (4 * 2) + (19 * 3) = 100 + 17 + 8 + 57 = 182

It took the player 350 AB to amass these 182 TB. Therefore, they have
a slugging percentage of 0.520: 100 / 350 = 0.52
```

logical operators (==, <, >, <=, >=, ===, !=) conditional logic: if - else if - else variable scope let & const vs var in { code block}

---

single (=), double (==) and triple (===) equal signs

= is the Assignment Operator; it assigns a value to a variable

1. Using =, declare a variable with an initial value, and then change its value:

   ```
   let x = 8;
   x = 5;
   ```

== is the Comparison Operator == compares two values to see if they are equal in value The comparison returns a boolean (true or false)

2. Using ==, log x == 5 and x == 6:

   ```
   console.log(x == 5); // true
   console.log(x == 6); // false
   ```

3. Compare x, which is 5, to the number-like string "5". Even though 5 is not "5", the == sees them as equal in value.

   ```
   console.log(x == "5"); // true
   ```

=== is the Strict Comparison Operator. To return true, the comparsion must be equal in both value and data type.

4. Using ===, compare of x to "5". It's false, because 5 and "5", have different data types.

   ```
   console.log(x === "5"); // false
   ```

Other comparison operators:

```
>   greater than in value
<   less than in value
>=  greater than or equal to in value
<=  less than or equal to in value
!   not (opposite)
!=  not equal in value
!== not equal in both value and data type
```

5. Open the console and type these comparisons, hitting enter each time:

   ```
   8 == 8; // true
   8 == "8"; // true
   8 === "8"; // false
   ```

6. Try the ! (NOT) operator:

```
7 != 8; // true
7 != 7; // false
7 != '7'; // false
7 !== '7'; // true
```

7. Try the greater than / less than operators:

```
7 >= 8; // false
7 >= '7'; // true
7 <= 8; // true
7 <= '7'; // true
```

---

alphanumeric characters for comparison purposes

The "digits" of number-like strings are characters, so comparisons are alphabetic, not numeric.

The following character sets evaluate in ascending order:

- number-like strings ("0123456789")
- uppercase letters ("ABCDEFGHIJKLMNOPQRESUVWZYZ")
- lowercase letters ("abcdefghijklmnopqurstuvwxyz")

8. Open the Console, and type these comparisons:

```
'1' < 'A'; // true
'A' < 'a'; // true
'A' == 'a'; // false
'Z' > 'a'; // false
'z' > 'a'; // true
```

9. Still in the Console, compare some numbers as well as their number-like string equivalents:

```
1000 > 50; // true (numeric)
"1000" > "50"; // false (alphabetic)
10000 < 5; // false (numeric)
"10000" < "5"; // true (alphabetic)
```

---

conditional logic with "if"

An "if statement" compares two values in parentheses and returns a boolean (true / false).

- The comparison is called the condition
- If the condition is true, the code inside the curly braces runs

evaluating a number in an if statement

10. Write an if statement that checks if x equals 5. It runs, because the condition is true:

```
if(x == 5) {
console.log("x equals 5"); // runs
}
```

If the condition is false, the code inside the curly braces does not run

11. Try another if statement that checks if x equals 3. It doesn't run, because the comparison is false:

```
if(x == 3) {
    console.log("x equals 3"); // doesn't run
}
```

---

if else

Often, we will want to run one piece of code if the condition is true and some other piece of code if it is false. The "other part" is handled by "else". The "else part" does not evaluate a condition, and therefore has no parentheses.

evaluating a number in an if-else statement

12. Add an else part to run if the condition is false:

```
if(x == 3) {
    console.log("x is 3"); // doesn't run
} else {
console.log("x is not 3"); // runs
}
```

13. "x is not 3" doesn't tell us much about x, so concatenate x into the output:

```
if(x == 3) {
    console.log("x is 3"); // doesn't run
} else {
    console.log(`x is ${x}, not 3`); // runs
}
```

14. Try the **!=** (not equal) operator, where the logic is reversed:

```
if(x != 3) {
    console.log(`x is ${x}, not 3`); // runs
} else {
    console.log("x is 3"); // doesn't run
}
```

15. Try the **>** (greater than) operator:

```
if(x > 3) {
    console.log(`x is ${x}, which is greater than 3`); // runs
} else {
    console.log(`x is ${x}, which is not greater than 3`); // doesn't run
}
```

16. Try the < (less than) operator:

```
if(x < 3) {
    console.log(`x is ${x}, which is less than 3`); // doesn't run
} else {
    console.log(`x is ${x}, which is not less than 3`); // runs
}
```

17. Try the <= (less than or equal to) operator:

```
if(x <= 5) {
    console.log(`x is ${x}, which is less than or equal to 5`); // runs
} else {
    console.log(`x is ${x}, which is not less than or equal to 5`); // doesn't run
}
```

evaluating a string with if-else logic

18. Declare that the weather is "sunny", and do an if-else that checks if the weather is sunny:

```
let weather = "sunny";

if(weather == "sunny") {
    console.log("It's sunny! Let's go to the beach!"); // runs
} else {
    console.log("It's not sunny! No beach today!"); // doesn't run
}
```

19. Change the weather to "stormy", and run it again. This time, concatenate the variable into the output so we can see what the weather is:

```
weather = "stormy";

if(weather == "sunny") {
    console.log("It's sunny! Let's go to the beach!"); // doesn't run
} else {
    console.log(`It's ${weather}! No beach today!`); // runs
}
```

evaluating a boolean with if-else logic

20. Declare a boolean, and evaluate it in an if statment to see if it's true:

```
let raining = true;

if(raining == true) {
   console.log("It's raining!"); // runs
}
```

21. Get rid of the == comparison, and just evaluate the variable itself:

```
if(raining) {
   console.log("It's raining!"); // runs
}
```

22. Check if raining is false. Since raining is true, add an else part:

```
if(raining == false) { // if it is true that raining is false
   console.log("It's not raining!"); // doesn't run
} else { // else it is false that raining is false
   console.log("It's raining!"); // runs
}
```

Checking if a boolean is false involves "reverse logic":

- Is it true that it's true? If so, it's false that it's false.
- Is it true that it's false? If so, it's false that it's true.
- **!** (NOT operator) before a boolean checks for false (NOT true)

23. Get rid of the == comparison, and put ! before the variable:

```
if(!raining) {  // if raining is false (NOT true)
   console.log("It's not raining!"); // doesn't run
} else { // else raining is true (NOT false)
   console.log("It's raining!"); // runs
}
```

---

if-else if-else logic

else if adds another condition to evaluate. Think of it as a sandwich, where "if" and "else" are the bread and "if else" is the ingredient(s) between the bread.
Let's try an "if - else if - else", where we compare two numbers and do one of three things, depending on their relative values.

24. Declare two numeric variables:

```
let highScore = 15000;
let myScore = 10000;
```

25. Check if my score is greater than the high score:

```
if(myScore > highScore) {
    console.log("Congrats! You beat the high score!"); // doesn't run
}
```

26. Add an else part:

```
if(myScore > highScore) {
    console.log("Congrats! You beat the high score!");
} else {
    console.log("Sorry! You didn't beat the high score!"); // runs
}
```

The if-else checks for higher and lower, but what if you tied the high score?

27. Add an else if between the if and the else. Now we handle all three possible outcomes:

```
if(myScore > highScore) {
    console.log("Congrats! You beat the high score!");
} else if(myScore < highScore) {
    console.log("Sorry! You didn't beat the high score!"); // runs
} else {
    console.log("Wow! You tied the high score!");
}
```

28. Make myScore greater than highScore and run it. Now the "if part" runs ("Congrats!")

```
myScore = 18000;
```

29. Make myScore equal to highScore and run it. Now the "else part" runs ("Wow!")

```
myScore = 15000;
```

30. Revisiting the weather logic, let's add an "else if part":

```
weather = "cloudy";

if(weather == "sunny") {
    console.log("It's sunny!");
} else if(weather == "raining") {
    console.log("It's raining!");
} else { // it's neither sunny nor raining
    console.log(`It's ${weather}`); // It's cloudy
}
```

31. Change weather to "sunny" so that the "if part" runs:

```
weather = "sunny";
```

32. Change weather to "raining" so that the "else if" runs:

```
weather = "raining";
```

multiple "if else" conditions

Just as a sandwich can have multiple items between the bread, so too can there be mulitple "else if" blocks between the "if" and "else". Only ONE code block ever runs, however, regardless of how many "else if" blocks there may be.

33. Declare a numeric variable.

```
let score = 97;
```

34. If the score is at least 90, declare a grade variable with a value to "A":

```
if(score >= 90) {
   let grade = "A";
}
```

35. If the score is less than 90, make that grade of "F":

```
if(score >= 90) {
   let grade = "A";
} else {
   let grade = "F";
}
```

36. A failing grade for anything less than 90 is pretty harsh, so add an "else if" part that awards a "B" if the score is at least 80:

```
if(score >= 90) {
   let grade = "A";
} else if(score >= 80) {
   let grade = "B";
} else {
   let grade = "F";
}
```

37. We're still failing any score below 80, so add two more if else blocks for grades of "C" and "D":

```
if(score >= 90) {
   let grade = "A";
} else if(score >= 80) {
   let grade = "B";
} else if(score >= 70) {
   let grade = "C";
} else if(score >= 65) {
   let grade = "D";
} else {
   let grade = "F";
}
```

38. Concatenate the score and grade into a report card and check the Console:

```
let reportCard = `Score: ${score} | Grade: ${grade}`;
```

We get an error: grade is not defined. But why? Clearly, we declared the grade variable and assigned it a string value Comment out the reportCard line to get rid of the error. This brings us to the important topic of variable scope.

---

variable scope: global vs. block scope

Variable scope refers to where in the code it exists and is available. We need to consider global scope vs. block scope.

- **score** is declared in the **global scope** and is therefore available everywhere in the script. By global scope we mean that the variable is not declared inside curly braces.
- **grade**, however, is declared inside curly braces. The code inside curly braces is known as a code block. Variables declared inside a code block are **block-scoped**, that is, only available inside the code block.

39. Move the declaration of grade into the global scope, right after score. Since we do not have a grade yet, set it equal to an empty string. Let's change the score while we are at it:

```
score = 87;
let grade = "";
```

40. Rewrite the "if else if - else" logic without the "let". Now, the global grade variable is assigned the "B":

```
if(score >= 90) {
   grade = "A";
} else if(score >= 80) {
   grade = "B";
} else if(score >= 70) {
   grade = "C";
} else if(score >= 65) {
   grade = "D";
} else {
   grade = "F";
}
```

41. Make a new reportCard and log it:

```
let reportCard = `Score: ${score} | Grade: ${grade}`;
console.log(reportCard); // Score: 87 | Grade: B
```

02.01 Lab

Instructions:

- Open **02.01-Lab.html** in your Editor. Write the Lab code in the **script** tags.

1. Declare a variable called lucyIsOnline and set its value to false. Write an if-else statement:

- if lucyIsOnline is true, console.log "Lucy is online",
- if lucyIsOnline is false, console.log "Lucy is not online"

2. Declare a variable price, and set it equal to 88. Write an if-else statement:

- if price is greater than or equal to 100, console.log 'Expensive'
- if price is less than 100, console.log 'Cheap'

3. Add an "else if" clause to the statement:

- if price is greater than 100, console.log 'Expensive'
- if price is between 50-99, console.log 'Reasonable'
- if price is less than 50, console.log 'Cheap'

4. Declare two variables: stars and review. Set stars equal to 4 and review equal to an empty string. Write an if else-else if-else statemetn:

- if 5 stars, review is "Great"
- if 4 stars, review is "Good"
- if 3 stars, review is "Meh"
- if 2 stars, review is "Bad"
- if 1 star, review is "Awful"
- console.log review below the whole thing

5. Debug the following:

```
let animal = 'cow';
let sound = '';

if (animal = 'dog') {
    sound = Woof;
} elseif (animal = 'cat') {
sound = Meow;
} elseif (animal = 'cow') {
    sound = Moo;
} else (Animal and sound both unknown) {
console.log('sound + !');

// desired output: Moo!
```

6. Given these variables, write an if-else with three "else if" blocks to evaluate multiple temperature ranges:

```
    let fahrenheit = 95;
    let weather = "";
```

- above 90 is hot
- 70-89 is warm
- 50-69 is cool
- 32-49 is cold
- below 32 is freezing
- **END Lab 02.01**
- **SEE Lab 02.01 Solution**

02.01 Lab Solution

1. Declare a variable called lucyIsOnline and set its value to false. Write an if-else statement:

```
let lucyIsOnline = false;

if (lucyIsOnline) {
    console.log("Lucy is online");
} else {
    console.log("Lucy is not online");
}
```

2. Declare a variable price, and set it equal to 88. Write an if-else statement:

- if price is greater than or equal to 100, console.log 'Expensive'
- if price is less than 100, console.log 'Cheap'

```
let price = 88;

if(price > 100) {
    console.log('Expensive');
} else {
    console.log('Cheap');
}
```

3. Add an "else if" clause to the statement:

- if price is greater than 100, console.log 'Expensive'
- if price is between 50-99, console.log 'Reasonable'
- if price is less than 50, console.log 'Cheap'

```
if(price > 100) {
    console.log('Expensive');
} else if (price > 50) {
    console.log('Reasonable');
} else {
    console.log('Cheap');
}
```

4. Declare two variables: stars and review. Set stars equal to 4 and review equal to an empty string. Write an if else-else if-else statement:

- if 5 stars, review is "Great"
- if 4 stars, review is "Good"
- if 3 stars, review is "Meh"

- if 2 stars, review is "Bad"
- if 1 star, review is "Awful"
- console.log review below the whole thing

```
let stars = 4;
let review = "";

if (stars == 5) {
    review = "Great";
} else if (stars == 4) {
    review = "Good";
} else if (stars == 3) {
    review = "Meh";
} else if (stars == 2) {
    review = "Bad";
} else {
    review = "Awful";
}
```

5. Debug the following:

```
let animal = 'cow';
let sound = '';

if (animal == 'dog') {
    sound = 'Woof';
} else if (animal == 'cat') {
    sound = 'Meow';
} else if (animal == 'cow') {
    sound = 'Moo';
} else {
    console.log('Animal and sound both unknown');
}
console.log(sound + '!'); // expected: Moo!
```

6. Given these variables, write an if-else with three "else if" blocks to evaluate multiple temperature ranges:

```
let fahrenheit = 95;
let weather = "";
```

- above 90 is hot
- 70-89 is warm
- 50-69 is cool
- 32-49 is cold
- below 32 is freezing

```
    if(fahrenheit > 90) {
        weather = "hot";
    } else if (fahrenheit > 70) {
        weather = "warm";
    } else if (fahrenheit > 50) {
        weather = "cool";
    } else if (fahrenheit > 32) {
        weather = "cold";
    } else { // 32 and below
        weather = "freezing";
    }
    console.log('The weather is ' + weather);
```

- nested if-else logic
- truthy-falsey values

nested if-else logic

Consider if-else logic involving a planned activity:

- if it's rainy, let's go to the museum.
- else if it's sunny, let's go to the beach
- else it's neither rainy nor sunny, so let's go to the park.

But what if there are also wind conditions that will determine what activities we engage in once we reach our destination. This requires nested if-else logic:

1. Declare two variables: a string and a boolean:

```
let weather = "rainy";
let windy = true;
```

2. Set up the non-nested logic, where we consider the weather, but not the wind conditions:

```
if(weather == "rainy") {
    console.log("Let's go to the museum!");
} else if (weather == "sunny") {
    console.log("Let's go to the beach!");
} else {
    console.log("Let's go to the park!");
}
```

3. Make the weather "sunny" so that the "else if" part runs:

```
weather = "sunny";
```

4. Make the weather "cloudy" so that the "else" part runs:

```
weather = "cloudy";
```

nested if-else

The wind don't much matter at the museum, but does at the beach and park. If it's windy at the beach, we will go windsurfing; otherwise we'll play frisbee.

5. Add a nested if-else to specify what to do at the beach, based on the wind:

```
if(weather == "rainy") {
console.log("Let's go to the museum!");
} else if(weather == "sunny") {
console.log("Let's go to the beach!");
if(windy) {
    console.log("Let's go windsurfing!");
} else {
    console.log("Let's play frisbee!");
}
} else {
console.log("Let's go to the park!");
}
```

6. Moving on to the last part, specify what to do at the park based on the wind:

```
if(weather == "rainy") {
console.log("Let's go to the museum!");
} else if(weather == "sunny") {
console.log("Let's go to the beach!");
if(windy) {
    console.log("Let's go windsurfing!");
} else {
    console.log("Let's play frisbee!");
}
} else {
console.log("Let's go to the park!");
if(windy) {
    console.log("Let's fly a kite!");
} else {
    console.log("Let's have a picnic!");
}
}
```

7. Change the values of weather and windy to get different output.

8. CHALLENGE: Given these variables:

Write nested if-else logic that considers whether today is a weekday, weekend or holiday, as well or if you're a student or not. Depending on the variable values, the logic should log one of six different strings:

- if it's a weekday and student is true: "Go to school!"
- if it's a weekday and student is false: "Go to work!"
- if it's the weekend and student is true: "Go to party!"
- if it's the weekend and student is false: "Mow the lawn!"
- if it's a holiday and student is true: "Road trip!"
- if it's a holiday and student is false: "Backyard BBQ!"

truthy and falsey values

Truthy and falsey values are not literally true or false, but they return true or false in an "if-statement".

These are the falsey values:

- NaN (Not a Number)
- undefined
- null
- 0
- "", '' (empty string)

All other values are truthy. These include:

- non-empty strings: "Hi", "5"
- non-zero numbers: 5, -5
- objects and arrays, even empty ones: [], {}

non-zero numbers are truthy

9. Pass any non-zero number to an if statement. Positive or negative, it returns true:

```
if(-3) {
    console.log("non-zero numbers are truthy");
}
```

non-empty strings are truthy

10. Pass a non-empty string to an if statement. It returns true:

```
if("hi") {
console.log("non-empty strings are truthy");
}
```

zero is falsey

11. Pass 0 to an if statement. Since zero is falsey, add an else part to get output:

```
if(0) {
    console.log("zero is truthy");
} else {
    console.log("zero is falsey");
}
```

empty strings are falsey

12. Pass "" to an if statement. Since empty strings are is falsey, add an else part to get output:

```
if("") {
    console.log("empty strings are truthy");
} else {
    console.log("empty strings are falsey");
}
```

array.length with truthy-falsey

Recall that arrays have a length property. If an array is empty, its length is 0. Since 0 is falsey, we can run an if statement that is true only if the array contains at least one item.

13. Declare an array of strings:

```
const fruitBowl = ["apple", "banana", "cherry"];
```

14. Log the array and its length property

```
console.log(fruitBowl); // ["apple", "banana", "cherry"]
console.log(fruitBowl.length); // 3
```

15. Log a message if the length is not zero:

```
if(fruitBowl.length) {
    console.log("fruit bowl has at least 1 fruit");
}
```

16. Declare another array, but leave its square brackets empty for an array of 0 items.

```
const pets = [];
```

17. Log the pets array and its length property:

```
console.log(pets); // []
console.log(pets.length); // 0
```

18. Check if an empty array is truthy or falsey:

```
if(pets) {
console.log('empty arrays are truthy');
```

```
} else {
console.log('empty arrays are falsey');
}
```

19. Log something if there is at least one item in pets, and something else if pets has a length of 0:

```
if(pets.length) {
console.log("pet array has at least 1 item");
} else {
console.log("pets array is empty--No pets!");
}
```

Reverse logic with the ! (NOT operator) may also be used to check for truthy-falsey values.

20. Check if pets.length is falsey, that is, equal to 0:

```
if(!pets.length) {
console.log("No pets!"); // runs
}
```

21. Try evaluating !fruitBowl.length, which returns false since the fruitBowl array contains 3 items. Add an else part to get output:

```
if(!fruitBowl.length) {
console.log("fruit bowl contains zero fruit!");
} else {
console.log("fruit bowl has at least 1 fruit");
}
```

NaN (Not a Number) is falsey

An impossible numeric or mathematical operation will result in NaN. The Number() method converts number-like strings like "5" to actual numbers, but if you pass it "banana", it returns NaN.

22. Try turning "banana" into a number, and log the result along with its data type. Oddly enough, the data type of NaN is number.

```
let baNaNa = Number('banana');
console.log('baNaNa', baNaNa, typeof(baNaNa)); // baNaNa NaN number
```

23. Pass the baNaNa to an if else statement:

```
if(baNaNa) {
    console.log('baNaNa is truthy');
} else {
    console.log('baNaNa is falsey'); // runs
}
```

Multiplication, division and subtraction involving number-like string(s) works, because JS is able to "figure out" that these should behave like actual numbers.

24. Try doing multiplication with a number and a number-like string. It works:

```
let price = 19.95; // number
let nycSalesTax = "8.875"; // number-like string
let total = price + (price * nycSalesTax); // math works
console.log('total', total); // total 197.00625
```

But if the string is not convertible to a number, say because it contains a unit like a dollar sign or percent sign, the math fails and we get NaN.

25. Try the preceding math problem but with the inclusion of "%":

```
price = 19.95; // number
nycSalesTax = "8.875%"; // string not convertible to number
total = price + (price * nycSalesTax); // math fails
console.log('total', total); // total NaN
```

Addition involving a number-like string fails, although it does not return NaN. Instead, the plus-sign results in concatenation.

26. Try adding a number to a number-like string. The result is concatenation--not addition:

```
let priceA = 15.75;
let priceB = "24.49";
let totPrice = priceA + priceB;
console.log('totPrice', totPrice); // totPrice 15.7524.49
```

undefined is falsey

undefined is the value as well as the data type of a variable that has not been assigned a value.

27. Do a test to confirm that undefined is falsey:

```
let user;
console.log('user', user, typeof(user)); // user undefined undefined
```

```
if(user) {
    console.log('undefined user (no value) is truthy');
} else {
console.log('undefined user (no value) is falsey'); // runs
}
```

null is falsey

null is an empty object. Unlike undefined, null is an assigned value.

28. Do a test to confirm that null is falsey:

```
let score = null;
console.log('score', score, typeof(score)); // score null object

if(score) {
    console.log('null values are truthy');
} else {
    console.log('null values are falsey'); // runs
}
```

Lab 03.02 - Instructions:

1. Write a nested if-else block, that satisfies the following conditions:
    - If the student score is greater than or equal to 90, if the student is in high school, they get a grade of "A", but if they are in college they get a grade of 4.
    - If the student score is greater than or equal to 80 and less than 90, if the student is in high school, they get a grade of "B", but if they are in college they get a grade of 3.
    - If the student score is greater than or equal to 70 and less than 80, if the student is in high school, they get a grade of "C", but if they are in college they get a grade of 2.
    - If the student score is greater than or equal to 65 and less than 70, if the student is in high school, they get a grade of "D", but if they are in college they get a grade of 1.
    - Else the grade is "FAIL" for both high school and college.
2. Write a nested if-else block, that calculates the price of a bet, based on the following conditions:
    - If the pet is a cat, increase the pet price by 20%, unless it is a kitten (baby cat), in which case decrease the price by 10%.
    - If the pet is a dog, increase the pet price by 30%, unless it is a puppy (baby dog), in which case decrease the price by 15%.
    - If the pet is anything besides a dog or a cat, double the price, unless it is a baby version of the pet, in which case cut the price in half.

02.02 Lab Solution

1. Write a nested if-else block, that satisfies the following conditions:

2. Write a nested if-else block, that satisfies the following conditions:

    - If the student score is greater than or equal to 90, if the student is in high school, they get a grade of "A", but if they are in college they get a grade of 4.
    - If the student score is greater than or equal to 80 and less than 90, if the student is in high school, they get a grade of "B", but if they are in college they get a grade of 3.
    - If the student score is greater than or equal to 70 and less than 80, if the student is in high school, they get a grade of "C", but if they are in college they get a grade of 2.
    - If the student score is greater than or equal to 65 and less than 70, if the student is in high school, they get a grade of "D", but if they are in college they get a grade of 1.
    - Else the grade is "FAIL" for both high school and college.

```
let score = 94;
let highSchool = true;
let grade;

if(score >= 90) {
    if(highSchool) {
        grade = "A";
    } else {
        grade = 4;
    }
} else if(score >= 80) {
    if(highSchool) {
        grade = "B";
    } else {
        grade = 3;
    }
} else if(score >= 70) {
    if(highSchool) {
        grade = "C";
    } else {
        grade = 2;
    }
} else {
    grade = "FAIL";
}

console.log(grade);
```

3. Write a nested if-else block, that calculates the price of a bet, based on the following conditions:

    - If the pet is a cat, increase the pet price by 20%, unless it is a kitten (baby cat), in which case decrease the price by 10%.

- If the pet is a dog, increase the pet price by 30%, unless it is a puppy (baby dog), in which case decrease the price by 15%.
- If the pet is anything besides a dog or a cat, double the price, unless it is a baby version of the pet, in which case cut the price in half.

```
let pet = "cat";
let price = 100;
let baby = true;

if(pet == "cat") {
    if(baby) {
        price *= 0.9; // to decrease by 10%, multiply by 0.9
    } else {
        price *= 1.2; // to increase by 20%, multiply by 1.2
    }
} else if(pet == "dog") {
    if(baby) {
        price *= 0.85; // to decrease by 15%, multiply by 0.85
    } else {
        price *= 1.3; // to increase by 30%, multiply by 1.3
    }
} else {
    if(baby) {
        price *= 0.5; // to decrease by half (50%), multiply by 0.5
    } else {
        price *= 2; // to double a number, multiply it by 2
    }
}

console.log(price);
```

- ternary expression
- && (AND) operator, || (OR) operator
- switch - case - break - default

ternary expression

A ternary is a concise alternative to an if-else statement. // What takes an "if else" five lines to do, a ternary does in one.

1. Declare three number variables.

```
let x = 5;
let y = 8;
let z = 0;
```

2. If x < y, add them; else multiply them; save the answer to z:

```
if(x < y) {
    z = x * y;
} else {
    z = x + y;
}
console.log('z', z); // 40
```

Now to convert the "if else" to a ternary:

3. Get rid of the "if()", "else" and curly braces:

```
//   x < y
//     z = x * y;
//     z = x + y;
```

4. Put a question mark in front of the if part:

```
//   x < y
//     ? z = x * y;
//     z = x + y;
```

5. Put a colon in front of the else part:

```
//   x < y
//     ? z = x * y;
//     : z = x + y;
```

6. Lose the first semi-colon; this makes it one line of code:

```
//   x < y
//     ? z = x * y
//     : z = x + y;
```

7. Back everything up onto the same line, and log z:

```
x < y ? z = x * y : z = x + y;
console.log(z); // 40
```

8. Instead of "z =" twice, put "z =" at the beginning:

```
z = x < y ? x * y : x + y;
console.log(z); // 40
```

CHALLENGE:

8. Convert the following "if else" to a ternary:

```
let n = 5;

if(n == 7) {
    n = 0;
} else {
    n++;
}
console.log('n', n); // 6

// TERNARY:

console.log('n', n); // 6
```

9. Also convert this "if else" to a ternary:

```
let day = "nice";
let beach;

if(day == "nice") {
    beach = true;
} else {
    beach = false;
```

```
}
console.log('beach:', beach); // beach: true

// TERNARY:

console.log('n', n); // 6
```

&& (AND), || (OR) operators

Two or more conditions can be evaluated in an if statement:

- && (AND) operator requires both conditions to be true
- || (OR) operator requires either condition to be true

&& (AND) operator

Let's try an if statement with && (AND), where both conditions must be true.

10. Declare two variables, a string and a number:

```
let weather = "sunny";
let temperature = 78;
```

11. Using &&, write an "if statement" where:

- weather must be "sunny" AND
- temperature must be at least 70
- if BOTH conditions are true, go to the beach:

```
if(weather == "sunny" && temperature >= 70) {
    console.log("It's warm and sunny! Let's go to the beach!");
}
```

12. Add an "else part"; it will run if EITHER or BOTH conditions are false:

```
if(weather == "sunny" && temperature >= 70) {
    console.log("It's warm and sunny! Let's go to the beach!");
} else {
    console.log("We're not going to the beach..");
}
```

There are 3 "no beach" scenarios:

- It's not sunny, nor is it warm enough.

- It's sunny, but it's not warm enough.
- It's warm enough, but it's not sunny.

13. Inside the "else", add a nested "if - else if - else" to handle the 3 "no beach" scenarios:

```
if(weather == "sunny" && temperature >= 70) {
    console.log("It's warm and sunny! Let's go to the beach!");
} else {
    console.log("We're not going to the beach..");
    if(weather != "sunny" && temperature < 70) {
        console.log("It's not sunny, nor is it warm enough.");
    } else if(weather == "sunny") {
        console.log("It's sunny, but it's not warm enough.");
    } else {
        console.log("It's warm enough, but it's not sunny.");
    }
}
```

14. Changing the values of weather and temperature, keep running it until you get all four possible outcomes.

15. Declare a few variables:

```
let cash = 150;
let credit = 300;
let buyStuff;
```

16. Write an if statement where we will buy stuff only if we have at least 100 in BOTH cash AND credit. Both conditions are true, so the "if part" runs:

```
if(cash >= 100 && credit >= 100) {
    buyStuff = true;
} else {
    buyStuff = false;
}
console.log('buy stuff:', buyStuff); // buy stuff: true
```

17. Reduce cash to 50 and run it again. Since only one condition is now true, the "else part" runs:

```
cash = 50;
credit = 300;

if(cash >= 100 && credit >= 100) {
    buyStuff = true;
} else {
    buyStuff = false;
```

```
}
console.log('buy stuff:', buyStuff); // buy stuff: false
```

|| (OR) operator

Let's try an if statement with || (OR), where only one condition must be true.

18. Change the && to ||. Now, only one condition needs to be true, so the "if part" runs:

```
cash = 50;
credit = 300;

if(cash >= 100 || credit >= 100) {
    buyStuff = true;
} else {
    buyStuff = false;
}
console.log('buy stuff:', buyStuff); // buy stuff: true
```

19. Set credit to below 100, so that neither conditions is true. The else part runs:

```
cash = 50;
credit = 30;

if(cash >= 100 || credit >= 100) {
    buyStuff = true;
} else {
    buyStuff = false;
}
console.log('buy stuff:', buyStuff); // buy stuff: false
```

&& and || in one if statement

20. Declare a few variables:

```
let meal = "dinner";
let potato = "baked";
let orderSteak;
```

21. Order steak if BOTH of these conditions are met: the meal is dinner AND The potato is baked

```
if(meal == "dinner" && potato == "baked") {
    orderSteak = true;
} else {
```

```
    orderSteak = false;
}

console.log("Order steak:", orderSteak); // Order steak: true
```

22. Change potato to mashed and run it again. We no longer order the steak, because only one condtion has been met:

```
potato = "mashed";

if(meal == "dinner" && potato == "baked") {
    orderSteak = true;
} else {
    orderSteak = false;
}

console.log("Order steak:", orderSteak); // Order steak: false
```

To combine && and || in the same if statement, use parentheses to compound the logic.

23. Add an || part, so that we order the steak if the meal is dinner AND the potato is either baked OR mashed. The two potatos must be wrapped in parentheses, like tin foil:

```
if(meal == "dinner" && (potato=="baked" || potato=="mashed")) {
    orderSteak = true;
} else {
    orderSteak = false;
}

console.log("Order steak:", orderSteak); // Order steak: true
```

24. Change the potato to "French fries", making the OR condition false. Since both meal and potato must be true, we don't order the steak:

```
potato = "French fries";

if(meal == "dinner" && (potato=="baked" || potato=="mashed")) {
    orderSteak = true;
} else {
    orderSteak = false;
}

console.log("Order steak:", orderSteak); // Order steak: false
```

CHALLENGES:

A.
Write an "if else" such that we will watch a movie if we have at least 2 hours of free time AND the genre is comedy. Otherwise, we will NOT watch a movie. After you get it to work so that you watch the movie, change the variable(s) so that you don't watch the movie.

```
```js
let watchMovie;
let genre = "comedy";
let hoursFree = 2.5;

if(hoursFree >= 2 && genre == "comedy") {
    watchMovie = true;
} else {
    watchMovie = false;
}
console.log("Watch movie:", watchMovie); // Watch movie: true

genre = "drama";
if(hoursFree >= 2 && genre == "comedy") {
    watchMovie = true;
} else {
    watchMovie = false;
}
console.log("Watch movie:", watchMovie); // Watch movie: false
```
```

B.
Write an "if else" such that we will wear our baseball cap if the temperature is less than 60 degrees OR if it is sunny OR if it is rainy. Only ONE of these three conditions need be true for us to wear the cap. If NONE of these three conditions are true, we will NOT wear the cap. Run it so that we DO wear the cap, and then change it so that we do NOT wear the cap.

```
```js
let wearCap;
let degrees = 50;
weather = "sunny";

if(degrees < 60 || weather == "sunny" || weather == "rainy") {
    wearCap = true;
} else {
    wearCap = false;
}
console.log("Wear cap:", wearCap); // Wear cap: true

degrees = 70;
weather = "cloudy";
```
```

```
if(degrees < 60 || weather == "sunny" || weather == "rainy") {
    wearCap = true;
} else {
    wearCap = false;
}
console.log("Wear cap:", wearCap); // Wear cap: false
```

C.
Write an "if else" such that we will drive to the mall if we have a car AND if the mall is less than 10 miles away OR traffic is light. In other words:

- If we have a car and the mall is 5 miles away, we go, even if the traffic is heavy.
- If we have a car and the mall is 15 miles away, we go only if the traffic is not heavy.
- If we have a car but the mall is 15 miles away the traffic is heavy, we don't go.
- If we don't have a car, we don't go regardless of distance to mall or traffic conditions.
- Run it a couple of times with different variable values, so that you go to the mall and don't go to the mall.

```
let hasCar = true;
let distance = 5;
let heavyTraffic = true;
let goToMall;

if(hasCar == true && (distance < 10 || heavyTraffic == false)) {
    goToMall = true;
} else {
    goToMall = false;
}
console.log("Go to mall:", goToMall); // Go to mall: true
```

A shorter version of above: no == comparison for booleans:

```
```js
if(hasCar && (distance < 10 || !heavyTraffic)) {
    goToMall = true;
} else {
    goToMall = false;
}
console.log("Go to mall:", goToMall); // Go to mall:: true
```
```

Making it so we don't go to the mall:

```js
distance = 15;
if(hasCar && (distance < 10 || !heavyTraffic)) {
    goToMall = true;
} else {
    goToMall = false;
}
console.log("Go to mall:", goToMall); // Go to mall:: false
```

D.
Expanding upon C, above:
If the mall is within walking distance (less than 2 miles), we will go there, regardless of whether or not you have a car.

```js
distance = 1.5;

if(hasCar && (distance < 10 || !heavyTraffic)) {
    goToMall = true;
} else {
    if(distance < 2) {
        goToMall = true;
    } else {
        goToMall = false;
    }
}
console.log("Go to mall:", goToMall); // Go to mall: true
```

E.
Expanding upon D, above:
Once we get to the mall, if we have at least $100, we will buy shoes. If we can't buy shoes, but we do have at least $50, we will buy a hoodie. If we can't buy a hoodie, assume that we have enough money to buy a T-shirt, and just buy that.

F.
Ternary Review Convert the following from "if else" to a ternary.

```js
let ca$h = 50;
let cred = 300;
let buyIt;

if(ca$h > 99 || cred > 99) {
```

```
    buyIt = true;
} else {
    buyIt = false;
}

// ternary version:
ca$h > 99 || cred > 99 ? buyIt = true : buyIt = false;
// Or even more concise:
buyIt = ca$h > 99 || cred > 99 ? true : false;

console.log('buy stuff:', buyStuff); // buy stuff: true
```

G.

Ternary Review

Convert the following from an "if else" to a ternary.

```
```js
let score = 98;
let grade = "";

if(score >= 65) {
    grade = "Pass";
} else {
    grade = "Fail";
}

console.log("Grade: " + grade);

// TERNARY:
score >= 65 ? grade = "Pass" : grade = "Fail";

console.log('Ternary Grade: ' + grade);
```
```

switch - case - break - default:

an alternative to "if-else if-else"

Given: two variables:

```
```js
let moneySymbol = "GBP";
let currency = "";
```
```

Given: currency is set based on moneySymbol:

```
```js
if(moneySymbol == "USD") {
    currency = "US Dollar";
} else if(moneySymbol == "GBP") {
    currency = "British Pound";
} else if(moneySymbol == "JPY") {
    currency = "Japanese Yen";
} else {
    currency = "Not Dollar, not Pound, not Yen";
}
console.log(`${moneySymbol}; ${currency}`);
```
```

3. Convert the "if-else if-else" to "switch-case-default":

- In a switch-case-break-default:
- the switch is compared to the case:
- switch("GBP") is compared to case: "USD";
- if they don't match, the next case is considered:
- switch("GBP") is compared to case: "GBP";
- if they match, the code up until the break runs.
- if no case matches the switch, the default runs.
- the default is like the else part of an "if else"

```
```
switch (moneySymbol) {

    case "USD": // condition test for (moneySymbol)
    // condition test is true, set value of currency variable
    currency = "US Dollar";
    break; // ends this part like a closing curly brace

    case "GBP":
    currency = "British Pound";
    break;

    case "JPY":
    currency = "Japanese Yen";
    break;

    default: // no case is true, so do this (like an else part)
    currency = "Not Dollar, not Pound, not Yen";
    break;
```
```

```
}
console.log(`${moneySymbol}; ${currency}`);
```

4. Change the moneySymbol to something that matches no case:

```
moneySymbol = "RMB";

switch (moneySymbol) {

    case "USD": // condition test for (moneySymbol)
    // condition test is true, set value of currency variable
    currency = "US Dollar";
    break; // ends this part like a closing curly brace

    case "GBP":
    currency = "British Pound";
    break;

    case "JPY":
    currency = "Japanese Yen";
    break;

    default: // no case is true, so do this (like an else part)
    currency = "Not Dollar, not Pound, not Yen";
    break;
}
console.log(`${moneySymbol}; ${currency}`);
```

02.03 Lab

Multiple "if conditions"

1. Given three variables, R, G, B, write an "if else" where:
    - if all R, G, B values are the same:
        - set 'allSame' to true
        - else set 'allSame' to false.

R, G, B values must be integers in the 0-255 range

```
```js
let R = 0, G = 0, B = 0;
let allSame;

// ANS:

console.log('allSame:', allSame);
```
```

2. Expanding upon #1 above, given a color variable:

```
let color = "";
```

Set the color value according to these rules:

- if R, G and B are ALL the same exact number:
    - if R, G, B are all 0, set color to "black",
    - if R, G, B are all 255, set color to "white",
    - otherwise, set color to "gray",
- if R, G and B are NOT all the same:
    - set color to "not gray"

Log both variables: allSame and color.
Try different values of R, G and B to get all possible results.

```
```js
// ANS:

console.log('allSame:', allSame, ' - color:', color);
```
```

3. Expanding upon #2, above, add logic where:

- if gray R,G,B values are in the 1-55 range, log "dark gray"
- if gray R,G,B values are in the 56-199 range, log "medium gray"
- if gray R,G,B values are in the 200-254 range, log "light gray"

```
// ANS:

console.log('allSame:', allSame, ' - color:', color);
```

Converting if-else to ternary:

4. Write an if-else and then convert it to a ternary;

- if x is less than or equal to y, divide x by y
- if x is greater than y, divide y by x
- in either case, set z equal to the answer

Try different x and y values so that the if and else parts both run.
The logic is such that z can never be greater than 1.

```
```js
let x = 45;
let y = 15;
let z = 0;

// ANS:

console.log(z); // 0.5

// ANS:
// TERNARY:

console.log(z); // 0.5
```
```

5. Given these variables:

```
let msg = "";
let str1 = 'apple';
let str2 = 'banana';
```

Using "if - else if - else" logic, compare str1 and str2, alphabetically, such that:

- if str1 comes before str2 (str1 = "apple", str2 = "banana")
    - set msg to "apple before banana"

- if str2 comes before str1 (str1 = "dog", str2 = "cat")
    - set msg to "cat before dog"
- if they are identical (str1 = "kiwi", str2 = "kiwi")
    - set msg to: "double kiwi"

Assume str1 and str2 are lowercase, non-empty strings.
Test the logic for all 3 outcomes.

```
```
// ANS:

console.log(msg);
```
```

6. Expanding upon #5, given this additional variable:

```
let half = "";
```

In a separate piece of "if - else if - else" logic, set the value of half, according to these rules:

- if str1 and str2 both start with "a" - "m" (str1 = "bat", str2 = "man")
    - set half to "both 1st half"
- if str1 and str2 both start with "n" - "z" (str1 = "north", str2 = "south")
    - set half to "both 2nd half"
- if str1 and str2 start with letters from different halves (str1 = "east", str2 = "west")
    - set half to "different halves"
- if str1 and str2 are the same
    - don't change half; it stays an empty string

Concatenate msg and half into a single string and log it:
Examples:

- str1 = "apple", str2 = "cat" logs "apple before cat - both 1st half"
- str1 = "noodle", str2 = "rice" logs "noodle before rice - both 2nd half"
- str1 = "up", str2 = "down" logs "down before up - different halves"
- str1 = "papaya", str2 = "papaya" logs "double papaya"

```
// ANS:

console.log(msg);
```

7. Given these variables, num and doubleDigit, where we may assume that num is a positive integer, write an "if else" where:

- if num is 2-digits (10-99), set doubleDigit to true
- if num is NOT 2-digits (0-9, 100+), set doubleDigit to false

```
let num = 8;
let doubleDigit;

// ANS:

console.log('doubleDigit:', doubleDigit);
```

8. Convert the "if else" for #7 to a ternary

```
// ANS:

console.log('doubleDigit:', doubleDigit);
```

9. Convert the following "if - else if - else" to "switch - case - break - default"; here is a recap of the syntax:

- compare value in switch parentheses to the case value:
- switch("apple") case: "apple"
- if they match, the code after case runs, stopping at break
- if they do not match, go on to consider the next case
- if end is reached with no switch-case match, default runs

```
let country = "Ghana";
let continent = "";

if (country === "Canada") {
    continent = "North America";
} else if (country === "China") {
    continent = "Asia";
} else if (country === "Ghana") {
    continent = "Africa";
} else if (country === "Bolivia") {
    continent = "South America";
}  else if (country === "France") {
    continent = "Europe";
} else {
    continent = "continent unknown";
}
```

```
console.log(`${country} is in ${continent}`);

// ANS ("switch - case - break - default" version):
country = "Paraguay";
```

02.03 Lab

Multiple "if conditions"

1. Given three variables, R, G, B, write an "if else" where:
    - if all R, G, B values are the same:
        - set 'allSame' to true
        - else set 'allSame' to false.

R, G, B values must be integers in the 0-255 range

```
```js
let R = 0, G = 0, B = 0;
let allSame;

// ANS:
if(R == G && R == B) {
    allSame = true;
} else {
    allSame = false
}

console.log('allSame:', allSame);
```
```

2. Expanding upon #1 above, given a color variable:

```
let color = "";
```

Set the color value according to these rules:

- if R, G and B are ALL the same exact number:
    - if R, G, B are all 0, set color to "black",
    - if R, G, B are all 255, set color to "white",
    - otherwise, set color to "gray",
- if R, G and B are NOT all the same:
    - set color to "not gray"

Log both variables: allSame and color.
Try different values of R, G and B to get all possible results.

```
```js
// ANS:
if(R == G && R == B) {
    allSame = true;
```
```

```
    if(R == 0) {
        color = "black";
    } else if(R == 255) {
        color = "white";
    } else {
        color = "gray";
    }
} else {
    allSame = false;
    color = "not gray";
}

console.log('allSame:', allSame, ' - color:', color);
```

3. Expanding upon #2, above, add logic where:

- if gray R,G,B values are in the 1-55 range, log "dark gray"
- if gray R,G,B values are in the 56-199 range, log "medium gray"
- if gray R,G,B values are in the 200-254 range, log "light gray"

```
// ANS:
if(R == G && R == B) {
    allSame = true;
    if(R == 0) {
        color = "black";
    } else if(R == 255) {
        color = "white";
    } else {
        // color = "gray";
        if(R <= 55) {
            color = "dark gray";
        } else if(R <= 199) {
            color = "medium gray";
        } else {
            color = "light gray";
        }
    }
} else {
    allSame = false;
    color = "not gray";
}

console.log('allSame:', allSame, ' - color:', color);
```

Converting if-else to ternary:

4. Write an if-else and then convert it to a ternary;

- if x is less than or equal to y, divide x by y
- if x is greater than y, divide y by x
- in either case, set z equal to the answer

Try different x and y values so that the if and else parts both run.
The logic is such that z can never be greater than 1.

```
```js
let x = 45;
let y = 15;
let z = 0;

// ANS:
if (x <= y) {
    z = x / y;
} else {
    z = y / x;
}

console.log(z); // 0.5

// ANS:
// TERNARY:
z = x <= y ? x / y : y / x;

console.log(z); // 0.5
```
```

5. Given these variables:

```
let msg = "";
let str1 = 'apple';
let str2 = 'banana';
```

Using "if - else if - else" logic, compare str1 and str2, alphabetically, such that:

- if str1 comes before str2 (str1 = "apple", str2 = "banana")
    - set msg to "apple before banana"
- if str2 comes before str1 (str1 = "dog", str2 = "cat")
    - set msg to "cat before dog"
- if they are identical (str1 = "kiwi", str2 = "kiwi")
    - set msg to: "double kiwi"

Assume str1 and str2 are lowercase, non-empty strings.
Test the logic for all 3 outcomes.

```js
// ANS:
    if (str1 < str2) {
    msg = `${str1} before ${str2}`;
} else if(str1 > str2) {
    msg = `${str2} before ${str1}`;
} else {
    msg = `double ${str1}`;
}

console.log(msg);
```

6. Expanding upon #5, given this additional variable:

```
let half = "";
```

In a separate piece of "if - else if - else" logic, set the value of half, according to these rules:

- if str1 and str2 both start with "a" - "m" (str1 = "bat", str2 = "man")
    - set half to "both 1st half"
- if str1 and str2 both start with "n" - "z" (str1 = "north", str2 = "south")
    - set half to "both 2nd half"
- if str1 and str2 start with letters from different halves (str1 = "east", str2 = "west")
    - set half to "different halves"
- if str1 and str2 are the same
    - don't change half; it stays an empty string

Concatenate msg and half into a single string and log it:
Examples:

- str1 = "apple", str2 = "cat" logs "apple before cat - both 1st half"
- str1 = "noodle", str2 = "rice" logs "noodle before rice - both 2nd half"
- str1 = "up", str2 = "down" logs "down before up - different halves"
- str1 = "papaya", str2 = "papaya" logs "double papaya"

```
// ANS:
if(str1 < 'n' && str2 < 'n') {
    half = "both 1st half";
} else if(str1 >= 'n' && str2 >= 'n') {
    half = "both 2nd half";
} else {
    half = "different halves";
}
```

```
console.log(msg);
```

7. Given these variables, num and doubleDigit, where we may assume that num is a positive integer, write an "if else" where:

- if num is 2-digits (10-99), set doubleDigit to true
- if num is NOT 2-digits (0-9, 100+), set doubleDigit to false

```
let num = 8;
let doubleDigit;

// ANS:
if(num >= 10 && num <= 99) {
    doubleDigit = true;
} else {
    doubleDigit = false;
}

console.log('doubleDigit:', doubleDigit);
```

8. Convert the "if else" for #7 to a ternary

```
// ANS:
doubleDigit = (num >= 10 && num <= 99) ? true : false;

console.log('doubleDigit:', doubleDigit);
```

9. Convert the following "if - else if - else" to "switch - case - break - default"; here is a recap of the syntax:

- compare value in switch parentheses to the case value:
- switch("apple") case: "apple"
- if they match, the code after case runs, stopping at break
- if they do not match, go on to consider the next case
- if end is reached with no switch-case match, default runs

```
let country = "Ghana";
let continent = "";

if (country === "Canada") {
    continent = "North America";
} else if (country === "China") {
```

```
    continent = "Asia";
} else if (country === "Ghana") {
    continent = "Africa";
} else if (country === "Bolivia") {
    continent = "South America";
}  else if (country === "France") {
    continent = "Europe";
} else {
    continent = "continent unknown";
}

console.log(`${country} is in ${continent}`);


// ANS ("switch - case - break - default" version):
country = "Paraguay";

switch (country) {

    case "Canada":
    continent = "North America";
    break;

    case "China":
    continent = "Asia";
    break;

    case "Ghana":
    continent = "Africa";
    break;

    case "Bolivia":
    continent = "South America";
    break;

    case "Frane":
    continent = "Europe";
    break;

    default: // no case is true, so do this (like an else part)
    continent = 'continent unknown';
    break;
}

console.log(`${country} is in ${continent}`);
```

02.04 LAB

1. Continuing with the "timely greeting", adjust the logic so that:

- if the time is 6:00 pm - 10:59 pm, greeting is "Good Evening"
- if the time is 11:00 pm - 3:59 am, greeting is "Hey, Night Owl!"
- if the time is 4:00 pm - 11:59 am, greeting is "Good Morning!"
- if the time is 12:00 - 5:59 pm, greeting is "Good Afternoon!"
- all other greeting times are unchanged

```
let dt;
let greeting = 'Good ';
```

Output the greeting to the tag on the web page with an id="greeting"

```
```js
console.log(greeting);
```
```

2. Make a "Activityies" link for each greeting

- Each activity should link to the provided URL
- The a-tag is already on the thml page
- Set the href and the text for the a-tag in the same
- if-else block that makes the greeting
- If the greeting is "Hey, Night Owl!", the link should say:
    - "Night Owl Activities" and should link to:
    - https://newyorksimply.com/nyc-things-to-do-in-new-york-city-at-night
- If the greeting is "Good Morning!", the link should say:
    - "Morning Activities" and should link to:
    - https://nymag.com/guides/everything/early-morning-2014-1/
- If the greeting is "Good Afternoon!", the link should say:
    - "Afternoon Activities" and should link to:
    - https://www.theinfatuation.com/new-york/guides/best-afternoon-tea-nyc
- If the greeting is "Good Evening!", the link should say:

    - "Evening Activities" and should link to:
    - https://websterhall.com/shows/

```
let nightOwlURL = "https://newyorksimply.com/nyc-things-to-do-in-new-york-city-at-night";

// hr = 1; // testing hr

console.log(greeting);
```

Test the output by hard-coding hr values. Sample hr values and their expected output

- When hr = 0, greeting is "Good Evening!"
- When hr = 3, greeting is "Hey, Night Owl!"
- When hr = 5, greeting is "Good Morning!"
- When hr = 15, greeting is "Good Afternoon!"
- When hr = 20, greeting is "Good Evening!"

02.04 LAB

1. Continuing with the "timely greeting", adjust the logic so that:

- if the time is 6:00 pm - 10:59 pm, greeting is "Good Evening"
- if the time is 11:00 pm - 3:59 am, greeting is "Hey, Night Owl!"
- if the time is 4:00 pm - 11:59 am, greeting is "Good Morning!"
- if the time is 12:00 - 5:59 pm, greeting is "Good Afternoon!"
- all other greeting times are unchanged

```
let dt = new Date();
let hr = dt.getHours();
let greeting = 'Good ';

if(hr < 4) {
    greeting = "Hey, Night Owl!";
} else if(hr < 12) {
    greeting = "Good Morning!";
} else if(hr < 18) {
    greeting = "Good Afternoon!";
} else if(hr < 22) {
    greeting = "Good Evening!";
} else { // hr is 23 (11:00-11:59pm)
    greeting = "Hey, Night Owl!";
}
```

Output the greeting to the tag on the web page with an id="greeting"

```
```js
const greetingTag = document.getElementById('greeting');
console.log(greeting);
greetingTag.textContent = greeting;
```
```

2. Make a "Activityies" link for each greeting

- Each activity should link to the provided URL
- The a-tag is already on the thml page
- Set the href and the text for the a-tag in the same
- if-else block that makes the greeting
- If the greeting is "Hey, Night Owl!", the link should say:

    - "Night Owl Activities" and should link to:
    - https://newyorksimply.com/nyc-things-to-do-in-new-york-city-at-night

- If the greeting is "Good Morning!", the link should say:

    - "Morning Activities" and should link to:
    - https://nymag.com/guides/everything/early-morning-2014-1/

- If the greeting is "Good Afternoon!", the link should say:

    - "Afternoon Activities" and should link to:
    - https://www.theinfatuation.com/new-york/guides/best-afternoon-tea-nyc

- If the greeting is "Good Evening!", the link should say:

    - "Evening Activities" and should link to:
    - https://websterhall.com/shows/

```
let aTag = document.querySelector('a');

let nightOwlURL = "https://newyorksimply.com/nyc-things-to-do-in-new-
york-city-at-night";

let morningURL = "https://nymag.com/guides/everything/early-morning-
2014-1/";

let afternoonURL = "https://www.theinfatuation.com/new-
york/guides/best-afternoon-tea-nyc";

let eveningURL = "https://websterhall.com/shows/";

hr = 1; // testing hr

if(hr < 4) {
    greeting = "Hey, Night Owl!";
    aTag.href = nightOwlURL;
    aTag.text = "Night Owl Activities";
} else if(hr < 12) {
    greeting = "Good Morning!";
    aTag.href = morningURL;
    aTag.text = "Morning Activities";
} else if(hr < 18) {
    greeting = "Good Afternoon!";
    aTag.href = afternoonURL;
    aTag.text = "Afternoon Activities";
} else if(hr < 22) {
    greeting = "Good Evening!";
    aTag.href = eveningURL;
    aTag.text = "Evening Activities";
} else { // hr is 23 (11:00-11:59pm)
    greeting = "Hey, Night Owl!";
    aTag.href = nightOwlURL;
    aTag.text = "Night Owl Activities";
```

```
}

console.log(greeting);
greetingTag.textContent = greeting;
```

Test the output by hard-coding hr values. Sample hr values and their expected output

- When hr = 0, greeting is "Good Evening!"
- When hr = 3, greeting is "Hey, Night Owl!"
- When hr = 5, greeting is "Good Morning!"
- When hr = 15, greeting is "Good Afternoon!"
- When hr = 20, greeting is "Good Evening!"

Lesson 03.01

functions

- A function is a block of code that runs only when it is invoked (called).
- A function usually (but not always) has a name to call it by.
- The function can be called by an event on the web page, such as a button click, however a function can also be called directly in the JS code.

defining (declaring) a function.

To declare or define a function, follow these steps:

1. Start with the keyword: function
2. Give the function a name. Variable naming rules apply. A function carries out some action, so it is customary for the name to be, or at least begin, with a verb, play, swapImage or greetWorld:function greetWorld
3. Tack on a pair of parentheses. These are for passing data into the function:
4. Next comes a pair of curly braces, inside of which goes the code which runs when the function is called. Let's just log "Hello World":

```
function greetWorld() {
    console.log("Hello World!");
}
```

Some functions have empty parentheses. We'll start by leaving them empty.

5. Run this example and check the Console.

Nothing appears. Where's the Hello World message? Remember, a function must be called, but all we have done so far is define the function. We do not have a button to click, or other web page element for calling the function, so's just call it directly in the code:

6. Call the function by writing its name, followed by the parentheses:

```
greetWorld();
```

7. Re-run the page and check the Console. We should see Hello World.
8. Check the data type of the function--it's function:

```
console.log('greetWorld() data type:', typeof(greetWorld));
// greetWorld() data type: function
```

variable scope

global variables

- A var, let or const declared outside of any curly braces are global in scope.
- A global variable is available everywhere in its script.

block scoped (local) variables

- A var, let or const declared inside the curly braces of a function is block scoped, that is, local to that function. Outisde of the function, the variables do not exist.
- A let or const declared inside the curly braces of an "if statement" are also block scoped, but a var so declared is global in scope.

9. As a review of variable scope, declare two variables inside the curly braces of an "if statement":

```
let meal = "dinner";

if(meal == "dinner") {
    var food = "burger";
    let bev = 'ginger ale';
    console.log(`I'll have a ${food} and ${bev}, please.`);
}
```

10. Try to access the food variable, which was declared inside the "if block":

```
console.log(`I'll have a ${food}, please.`); // I'll have a burger,
please.
```

This works, because var variables declared are not confined by the curly braces.

11. Try to access the bev variable, which was declared inside the "if block":

```
console.log(`I'll have a ${entree}, please.`);
```

This throws an error, because the let variable is confined to the curly braces.

12. Declare another function with variables declared inside the function:

```
function welcomePlayer() {
    var user = "Pika2";
    let score = 500;
    console.log(`Welcome, ${user}! Your score is ${score}.`);
}
```

13. Call the function and check the console. We get the expected output: Welcome, Pika2! Your score is 500.

14. Try to access the user variable:

```
console.log(`Welcome ${user}!`);
```

This throws an error, because var, let and const declared inside a function are "function scoped", which means that they are available only inside of the function.

function parameters and arguments

- Parameters (params for short) are inputs of a function that go in the parentheses.
- Inside the function, parameters are local variables.
- When a function is called, the params are assigned values as arguments passed into the function parentheses.

15. Write a function with a parameter:

```
function greetPlayer(username) {
    console.log(`Greetings, ${username}!`);
}
```

16. Call the function twice, with different arguments each time

```
greetPlayer("Brian123"); // Greetings, Brian123!
greetPlayer("Sally789"); // Greetings, Sally789!
```

If a function expects an argument, but none is provided, the missing value will be 'undefined'.

17. Call the function again, but this time omit the argument:

```
greetPlayer(); // Greetings, undefined!
```

18. Write a function with two parameters:

```
function greetWithScore(username, score) {
    console.log(`Hey, ${username}! Your score is ${score}.`);
}
```

19. Call the function, passing in both expected arguments:

```
greetWithScore("Dan12", 4543); // Hey, Day12! Your score is 4543.
```

20. Call the function again, but this time omit the score argument:

```
greetWithScore("Ida34"); // Hey, Ida34! Your score is undefined.
```

default parameter values

- A parameter can be assigned a value when the function is declared.
- That way, if no argument is supplied for it, it uses the default.

21. Write a function but a default value:

```
function greetWScore(user, highScore=3000) {
    console.log(`Welcome ${user}! Your high score is ${highScore}!`);
}
```

22. Call the function, passing in both expected arguments:

```
greetWScore("Xyz1", 12300); // Hey, Xyz1! Your score is 12300.
```

23. Call the function again, but this time omit the second argument. This time, rather than resulting in undefined, the second argument will use its default value:

```
greetWScore("Abc2"); // Hey, Abc2! Your score is 3000.
```

Global variables are available everywhere, including to all functions:

24. Write a function called addNumbers, and give it two parameters. These are the number that are added together by the function:

```
function addNumbers(num1, num2) {
    console.log(num1 + num2);
}
```

25. Call the function and provide the expected numbers to add:

```
addNumbers(85, 79); // 164
```

function return values

So far, our functions have produced only console output. This is useful for tesing and debugging, but the result not been saved to any variable. In the addNumbers example, our "164" made it to the console, but otherwise went poof. To "preserve the result" of a persist, don't just log it--return it. This "exports the result" out of the function, where you may "capture it" by setting the function call itself equal to a variable. To return a "result", put the keyword 'return' in front of the value. The return also terminates the function, so make return the last thing the function does.

26. Define another function, but with a big difference: this one has a return statement:

```
function multiplyNumbers(n1, n2) {
    let product = n1 * n2;
    return product;
}
```

27. Call the function, setting the function call itself equal to a variable. This "captures" the product -- the return value:

```
let prod = multiplyNumbers(12, 15);
console.log('prod:', prod); // prod: 180
```

28. Save a step by directly returning the math expression, as opposed to first saving the product to a variable and then returning the variable:

```
function multiplyNums(n1, n2) {
    return n1 * n2;
}

prod = multiplyNums(15, 5); // 75
```

29. Refactor the greeting-style function by having it return a value, rather than just log a message to the console.

```
function getGreetingAndScore(username, score) {
    return `Welcome ${username}! Your score is ${score}.`;
}

let greeting = getGreetingAndScore("Viper", 7340);
console.log('greeting:', greeting); // Welcome, Viper! Your score is
7340.
```

30. Try logging the function call itself, rather than saving it to a variable:

```
console.log(getGreetingAndScore("Kit26", 890)); // Welcome, Kit26!
Your score is 890!
```

calculate cafe bill function

31. Declare a calcCafeBill function with four parameters: food, bev, taxPct and tipPct, the last two with default values of 8.875 and 18, respectively.

```
function calcCafeBill(food, bev, taxPct=8.875, tipPct=18) {
```

The function adds the food and bev together to obtain the sub-total The tax equals the sub-total times the tax percent. The tip equals the sub-total times the tip percent. The final total equals the sub-total plus the tax plus the tip.

````
32. Add food and bev food and bev together to obtain the subTot

```js
let subTot = food + bev;
```

33. Calculate the tip and tax. Since these are percents, divide by 100:

```js
let tip = subTot * tipPct / 100;
let tax = subTot * taxPct / 100;
```

34. Add everything together to get the final total:

```js
let tot = subTot + tip + tax;
```

35. Concatenate and return the itemized guest check. Round the monetary
values to 2 digits (cents) with toFixed(2):

```js
return `
    *** JavaSlurp Cafe ***
    *** Guest Check ***
    Food: $${food}
    Bev: $${bev}
    Sub-total: $${subTot}
    Tip Pct: ${tipPct}%
    Tip: $${tip.toFixed(2)}
    Tax Pct: ${taxPct}%
````
````

```
    Tax: $${tax.toFixed(2)}
    Please Pay: $${tot.toFixed(2)}
    *** Thank you! ***`;
```

} // function calcCafeBill

````
36. Call the function, passing in the four expected arguments, and save
the resulting string to a variable:

    ```js
    let bill = calcCafeBill(56, 35, 18, 6);
    console.log(bill);
    ```

37. Call the function again, but this time, omit the last two arguments.
The default tip (18) and tax (8.875) will be used. log the func directly
w/o saving to a var cuz the return value IS the func call itself

    ```js
    bill = calcCafeBill(156, 135);
    console.log(bill);
    ```

### CHALLENGE:
A.
- Most restaurant bills show the date and time, so add these to the bill.
- See Lesson 02.04 Date Object to refresh your memory on how to do this
- Do this as a new function by the name of **calcCafeBillDateTime**.
- To keep the date time reasonably simple:
    - no seconds needed
    - no conversion to AM-PM; just use the default military time
    - no day of the week; just output the date as day/month/year

    ```js
    function calcCafeBillDateTime(food, bev, taxPct=8.875, tipPct=18) {

        let subTot = food + bev;
        let tip = subTot * tipPct / 100;
        let tax = subTot * taxPct / 100;
        let tot = subTot + tip + tax;

        // the new stuff for the date time
        let dt = new Date();
        let hour = dt.getHours();
        let minutes = dt.getMinutes();
        let date = dt.getDate();
        let month = dt.getMonth();
        let year = dt.getFullYear();

        return `
````
````

```
            *** JavaSlurp Cafe ***
            *** Guest Check ***
            *** Date: ${date}/${month}/${year}
            *** Time: ${hour}:${minutes}
            Food: $${food}
            Bev: $${bev}
            Sub-total: $${subTot}
            Tip Pct: ${tipPct}%
            Tip: $${tip.toFixed(2)}
            Tax Pct: ${taxPct}%
            Tax: $${tax.toFixed(2)}
            Please Pay: $${tot.toFixed(2)}
            *** Thank you! ***`;

    } // function calcCafeBillDateTime

    bill = calcCafeBillDateTime(265, 180);
    console.log(bill);
    ```
```
```

Lesson 03.01

1. The weight of anything on the moon is approximately one-sixth its weight on earth. A person who weighs 60 kg on earth would weigh 10 kg on the moon.
   Write a function called calcMoonWeight with a parameter that:

- takes an earth weight as its argument,
- converts it to moon weight
- returns the moon weight.
- call the function, passing in an earth weight number, saving the call to a variable
- log the variable (the answer as moon weight)

```
function calcMoonWeight(earthWt) {
    // convert earth weight to moon weight and return result
}

let moonWt = calcMoonWeight(150);
console.log(moonWt); // 25
```

2. Complete this function, which convert feet to meters, rounds to two decimal places, and returns the result. Conversion units:

- 39.37 inches = 1 meter
- 12 inches = 1 foot When you call the function, save the return value to a variable, and then log the variable.

```
function convertFeetToMeters(feet) {
    // convert feet to meters and return result
}

let meters = convertFeetToMeters(408);
console.log(meters); // 124.36
```

3. Bonus: Make the convert function work in BOTH directions:

- if input is in feet, convert to meters
- if input is in meters, convert to feet
- HINT: Use if-else logic and a boolean to keep track of whether the input unit is feet or meters

4. Extra Bonus: Refactor the if-else into a ternary. This time only return a number; add the units later:

5. Make a function called squareNum that:

    - takes in one number
    - squares the number and returns the result
    - input 4, get back 16.

    - Run the function twice, with different values.

6. Write a function called squareOrCubeNum that:

    - takes a positive integer, n, as its argument
    - if n is even, square it (input 6, get back 36)
    - if n is odd, cube it (input 5, get back 125)
    - return the answer

    Run the function with odd and even inputs HINT: if a number divided 2 and has a remainder of 0, the number is even (% operator finds remainder)

7. Refactor the above as a ternary, instead of if-else:

8. Declare a function called introducePet, that has four params: pet, name, age and sound

- Function returns a message, such that if the arguments are:
    - For "cat", "Fluffy", 3, "Meow", the returned message is:
    - Meow! My name is Fluffy! I am a 3-year-old cat!.
- Run the function three times, with different inputs:
    - "cat", "Fluffy", 3, "Meow"
    - "dog", "King", 2, "Woof";
    - "bunny", "Bunny", 4, "Thump";

9. Declare a function with two parameters: num1 and num2.

- The function call passes in two arguments, both numbers.
- The function does math and returns the answer:
- if num1 is greater than num2, subtract num2 from num1
- if num1 is less than num2, add the numbers together.
- if num1 and num2 are equal, square it.
- if only one number is passed in, square it.
- Run the function four times, once for each scenario.

10. Write a function findHypotenuse, having two parameters a and b, representing the two shorter sides of a right triangle

- function calculates and returns the hypotenuse, c
- Pythagorean Theorem is $a^2 + b^2 = c^2$
- function returns the hypotenuse Call the function 3 times, each time passing in two sides of a Pythagorean Triple, so named because all 3 sides are integers:

11. Write a function called addUpCoinValue that

- takes in whole numbers of pennies, nickels, dimes and quarters
- adds up the total value of all coins
- returns the total as dollars and cents, with a dollar-sign

12. Define a function that:

- takes the radius of a circle as its input

- calculates the area of the circle by the formula: A = πr²
- returns the area

13. Given these array of :

```
const nums1 = [4, 16, 8, 12, 3, 5, 18, 9];
const nums2 = [54, 46, 82, 27, 10, 65, 23, 39];
```

- write a function that takes in an array of positive integers
- returns the difference between the array's min and max values
- for nums1, the difference is 15 (18 - 3 = 15)
- for nums2, the difference is 72 (82 - 10 = 72)
- HINT: Use the Spread Operator (...) and the Math object

Lesson 03.01

1. The weight of anything on the moon is approximately one-sixth its weight on earth. A person who weighs 60 kg on earth would weigh 10 kg on the moon.
   Write a function called calcMoonWeight with a parameter that:

- takes an earth weight as its argument,
- converts it to moon weight
- returns the moon weight.
- call the function, passing in an earth weight number, saving the call to a variable
- log the variable (the answer as moon weight)

```
function calcMoonWeight(earthWt) {
    // convert earth weight to moon weight and return result
    return earthWt / 6;
}

let moonWt = calcMoonWeight(150);
console.log(moonWt); // 25
```

2. Complete this function, which convert feet to meters, rounds to two decimal places, and returns the result. Conversion units:

- 39.37 inches = 1 meter
- 12 inches = 1 foot When you call the function, save the return value to a variable, and then log the variable.

```
function convertFeetToMeters(feet) {
    // convert feet to meters and return result
    return (feet * 12 / 39.37).toFixed(2);
}

let meters = convertFeetToMeters(408);
console.log(meters); // 124.36
```

3. Bonus: Make the convert function work in BOTH directions:

- if input is in feet, convert to meters
- if input is in meters, convert to feet
- HINT: Use if-else logic and a boolean to keep track of whether the input unit is feet or meters

```

```

```
function convertDistance(dist, isFeet) {
    if(isFeet) {
        return `${(dist * 12 / 39.37).toFixed(2)} m`;
    } else {
        return `${(dist * 12 / 39.37).toFixed(2)} ft`;
    }
}

let dist = convertDistance(50, true);
console.log('50 ft = ', dist); // 50 ft = 15.24 m
```

4. Extra Bonus: Refactor the if-else into a ternary. This time only return a number; add the units later:

```
function convertDist(dist, isFeet) {
    return isFeet ? dist * 12 / 39.37 : dist * 39.37 / 12;
}

dist = convertDist(30, false);
console.log(`30m = ${dist.toFixed(2)}'`); // 30m =  98.42' ft
```

5. Make a function called squareNum that:

    - takes in one number
    - squares the number and returns the result
    - input 4, get back 16.
    - Run the function twice, with different values.

```
function squareNum(n) {
    // square n and return it
    return n ** 2;
}

let sq = squareNum(4);
console.log(sq); // 16

sq = squareNum(14);
console.log(sq); // 196
```

6. Write a function called squareOrCubeNum that:

    - takes a positive integer, n, as its argument
    - if n is even, square it (input 6, get back 36)
    - if n is odd, cube it (input 5, get back 125)
    - return the answer

    Run the function with odd and even inputs HINT: if a number divided 2 and has a remainder of 0, the number is even (% operator finds remainder)

```
function squareOrCubeNumber(n) {
    if(n % 2 == 0) { // if n is even
        return n ** 2; // square it
    } else { // else n is odd
        return n ** 3; // cube it
    }
}

console.log(squareOrCubeNumber(8)); // 64
console.log(squareOrCubeNumber(3)); // 27
```

7. Refactor the above as a ternary, instead of if-else:

```
function squareOrCubeNum(n) {
    return n % 2 == 0 ? n ** 2 : n ** 3;
}

console.log(squareOrCubeNum(12)); // 144
console.log(squareOrCubeNum(6)); // 216
```

8. Declare a function called introducePet, that has four params: pet, name, age and sound

- Function returns a message, such that if the arguments are:
    - For "cat", "Fluffy", 3, "Meow", the returned message is:
    - Meow! My name is Fluffy! I am a 3-year-old cat!.
- Run the function three times, with different inputs:
    - "cat", "Fluffy", 3, "Meow"
    - "dog", "King", 2, "Woof";
    - "bunny", "Bunny", 4, "Thump";

```
function introducePet(pet, name, age, sound) {
    return `${sound}! My name is ${name}! I am a ${age}-year-old
${pet}`;
}

console.log(introducePet("cat", "Fluffy", 3, "Meow"));
console.log(introducePet("dog", "King", 2, "Woof"));
console.log(introducePet("bunny", "Bunny", 4, "Thump"));
```

9. Declare a function with two parameters: num1 and num2.

- The function call passes in two arguments, both numbers.
- The function does math and returns the answer:

- if num1 is greater than num2, subtract num2 from num1
- if num1 is less than num2, add the numbers together.
- if num1 and num2 are equal, square it.
- if only one number is passed in, square it.
- Run the function four times, once for each scenario.

```
function doMath(num1, num2) {
    if(num1 > num2) {
        return num1 - num2;
    } else if(num1 < num2) {
        return num1 + num2;
    } else { // num1 must be equal to num2 or num2 is undefined
        return num1 ** 2;
    }
}

console.log(doMath(16, 7)); // 9
console.log(doMath(7, 16)); // 23
console.log(doMath(7, 7)); // 49
console.log(doMath(16)); // 256
```

10. Write a function findHypotenuse, having two parameters a and b, representing the two shorter sides of a right triangle

- function calculates and returns the hypotenuse, c
- Pythagorean Theorem is $a^2 + b^2 = c^2$
- function returns the hypotenuse Call the function 3 times, each time passing in two sides of a Pythagorean Triple, so named because all 3 sides are integers:

```
function findHypotenuse(a, b) {
    return Math.sqrt(a ** 2 + b ** 2);
}

let c = findHypotenuse(3, 4);
console.log(c); // 5

c = findHypotenuse(6, 8);
console.log(c); // 10

c = findHypotenuse(5, 12);
console.log(c); // 13
```

11. Write a function called addUpCoinValue that

- takes in whole numbers of pennies, nickels, dimes and quarters
- adds up the total value of all coins
- returns the total as dollars and cents, with a dollar-sign

```
function addUpCoinValue(p, n, d, q) {
    let cents = p + (n*5) + (d*10) + (q*25); // add up total cents
    let dollars = `${(cents/100)}`; // divide by 100
    return ` $${dollars}`;
}

let money = addUpCoinValue(17, 8, 16, 20);
console.log(money); // $7.17

money = addUpCoinValue(175, 85, 165, 205);
console.log(money); // $73.75
```

12. Define a function that:

- takes the radius of a circle as its input
- calculates the area of the circle by the formula: A = πr²
- returns the area

```
function areaOfCircle(r) {
    return Math.PI * r ** 2;
}

let area = areaOfCircle(12);
console.log(area1); // 452.3893421169302

area = areaOfCircle(16);
console.log(area); // 804.247719318987
```

13. Given these array of :

```
const nums1 = [4, 16, 8, 12, 3, 5, 18, 9];
const nums2 = [54, 46, 82, 27, 10, 65, 23, 39];
```

- write a function that takes in an array of positive integers
- returns the difference between the array's min and max values
- for nums1, the difference is 15 (18 - 3 = 15)
- for nums2, the difference is 72 (82 - 10 = 72)

- HINT: Use the Spread Operator (...) and the Math object

```
function findMinMaxDiff(nums) {
    return Math.max(...nums) - Math.min(...nums);
}

console.log(findMinMaxDiff(nums1)); // 15
console.log(findMinMaxDiff(nums2)); // 72
```

Lesson 03.02 START

This page consists of several UI elements which call functions and produce output.

Get the various DOM elements:

1. Get the output box:

```
const outBox = document.getElementById('out-box');
```

2. Get the text input fields, where the user enters their name:

```
const firstName = document.getElementById('first-name');
const lastName = document.getElementById('last-name');
```

3. Get the Greet World and Greet User buttons:

```
const greetWorldBtn = document.getElementById('greet-world-btn');
const greetUserBtn = document.getElementById('greet-user-btn');
```

4. Instruct the buttons to call their respective functions:

```
greetWorldBtn.addEventListener('click', greetWorld);
greetUserBtn.addEventListener('click', greetUser);
```

5. Get the "Pick a Fruit" menu:

```
const fruitMenu = document.getElementById('fruit-menu');
```

6. Whenever the fruit menu is changed, call a function:

```
fruitMenu.addEventListener('change', pickFruit);
```

7. Get the numeric input fields:

```
const num1 = document.getElementById('num1');
const num2 = document.getElementById('num2');
```

8. Get the menu for choosing the math operator:

```
const mathMenu = document.getElementById('math-menu');
```

9. Whenever a number changes or a math operator is chosen, call the calculateAnswer function:

```
mathMenu.addEventListener('change', calculateAnswer);
num1.addEventListener('change', calculateAnswer);
num2.addEventListener('change', calculateAnswer);
```

Extra functions for running alternate code scenarios

We want to try out some cool, new moves, so we'll need alternate versions of the pickFruit and calculateAnswer functions.

10. Make calls to these alternate functions, but keep them commented out for the time being; these functions won't exist for a while, but we'll come back to them:

```
fruitMenu.addEventListener('change', getFruitFood);
mathMenu.addEventListener('change', calculate);
num1.addEventListener('change', calculate);
num2.addEventListener('change', calculate);
```

Define the functions: Whenever you mention a function in a listener, the function must actually exist or else you get an error. Comment out all listeners except for the one that calls greetWorld.

11. Define the **greetWorld()** function:

```
function greetWorld() {
  // 12. Output the greeting to the web console:
  console.log("Hello World!");
  // 13. Now output greeting to the DOM--the web page:
  outBox.textContent = "Hello World!";
}
```

14. Define the **greetUser()** function:

```
function greetUser() {

  // 15. Get the values from the name fields and save them to local
variables:
  let fName = firstName.value;
  let lName = lastName.value
```

```
  // 16. Concatenate a greeting from the inputted names:
  let greeting = `Greetings, ${fName} ${lName}!`;
  outBox.textContent = greeting;

  // 17. If no names were inputted, we get a wonky-looking "Greetings, !".
Let's refactor with if-else logic: if both names are blank, output "Hey,
you!":
  if(!fName && !lName) { // if both names are falsey empty strings
      outBox.textContent = "Hey, you!";
  } else { // if at least one name is NOT an empty string, greet by
name(s):
      outBox.textContent = `Hi, ${fName} ${lName}!`;
  }
}
```

CHALLENGE:

A. Make it a Timely Greeting, so that instead of "Hey, you!", we get "Good morning!" or other appropriate greeting for the time of day. If at least one name field is NOT blank, output the timely greeing along with the name(s):

B. Refactor the if-else as a ternary

this and event.target

The this keyword, fruitMenu and event.target all refer to the same thing: the object that called the function; in this case, that object is the fruit menu:

18. Define the **pickAFruit()** function:

```
function pickFruit(event) {

  // 19. Log the value of the fruit menu in each of the three ways:
  console.log('this.value:', this.value);
  console.log('fruitMenu.value:', fruitMenu.value);
  console.log('event.target.value:', event.target.value);
  // 20. Get rid of the cross out line, by passing event to the function
  // 21. Output the chosen fruit in UPPERCASE:
  outBox.textContent = `You picked ${fruitMenu.value.toUpperCase()}`;
}
```

"data-" attribute

Additional data may be attached to any html element by means of the "data -" attribute, where "data-myVar" means that a myVar variable is associated with this tag and will be available to JS as the dataset.myVar property.

menu.options

The select object has an options property, which stores all the options as an array. We will get deeper into arrays later on, but we already know that an array is a variable which stores multiple items.

index

The items in an array are stored by numeric position, called index, with the first item at index 0.

menu.selectedIndex property

The select object has a selectedIndex property, which is the index of the selected item. So, if the 10th menu item is selected, the selectedIndex is 9.

menu.options[menu.selectedIndex]

The selected option can be gotten from the options array by looking it up by its index.

menu.option.dataset

Once we have the selected option, we can get any data contained in the "data-" attribute via the option's dataset property. Most tags won't have a "data-" attribute, but the fruit menu options all have a "data-food" attribute, the value of which is accessible as menu.option.dataset.food:

menu.option.dataset.food

```
<option value="mango" data-food="mango salsa">Mango</option>
```

- **menu** : the select menu as a whole
- **option** : the selected menu option
- **dataset** : property of an element option that has a "data-" attribute
- **food** : the property of dataset via the "data-food" attribute

Let's define a new function for tapping into the "data-food" attribute:

22. Comment out the listener that calls the pickFruit function, and activate the listener that calls getFruitFood.

23. Define the **getFruitFood()** function, which is an alternate function to the "Pick a Fruit" menu.

```
function getFruitFood(event) {

  // 24. Save the value of the fruit menu to a variable:
  let fruit = fruitMenu.value;

  // 25. Save the selectedIndex to a variable. This is the index of
selected option:
  let indx = fruitMenu.selectedIndex;

  // 26. Look up the selected option in the options array and save it to
an object:
  let optn = fruitMenu.options[indx];
```

```
  // 27. Save the food property of the option's dataset property to a
variable:
  let food = optn.dataset.food;

  // 28. Output the chosen food:
  outBox.textContent = `Try the ${food}!`;
}
```

The Calculator

The calculator works as follows:

- **calculateAnswer()** does the math
- the function is called by a **change** event
- the change event occurs whenever:
    - the user enters numbers(s) and hits Enter
    - the user uses the arrows to increase-decrease numbers
    - the user chooses a math operator from the menu
- **calculateAnswer()** function does the following:
    - gets the values from the number input boxes
    - converts these "number like strings" to actual numbers
    - gets the operator from the math menu
    - does the math in accordance with the math operator We have two ways of doing the math, which is why we have two calculate functions (the second one is MUCH shorter).

switch-case-break revisited

We need to run a big chunk of logic that does the math based on that operator. We could use a long if-else if-else, but instead, let's get take the opportunity to get in a little more practice with switch-case-break:

29. Define the **calculateAnswer()** function:

```
function calculateAnswer() {

  // 30. Get the values from the number boxes and convert them to actual
numbers
  let n1 = Number(num1.value);
  let n2 = Number(num2.value);

  // 31. Get the math operator, which is the value of the math menu:
  let mathOperation = mathMenu.value;

  // 32. Declare a variable to hold the answer:
  let ans = 0;

  // 33. Start with switch, which takes what is to be compared to the
case:
  switch(mathOperation) {

      // 34. Write the case to compare to mathOperation:
```

```
        case 'add':
            // 35. If the switch-case comparision is true, run the case
code:
            ans = n1 + n2;
        // 36. After the case code, break before going on to the next case:
        break;

        // 37. Repeat the above for all the other operations:
        case 'subtract':
            ans = n1 - n2;
        break;

        // 38. Since each case executes only one line, roll it up onto the
same line:
        case 'multiply': ans = n1 * n2;
        break;
        case 'divide': ans = n1 / n2;
        break;
        case 'modulo': ans = n1 % n2;
        break;
        case 'exponent': ans = n1 ** n2;
        break;
    }

    outBox.textContent = ans;

}
```

eval()

The eval() method takes a a string as its argument and considers it as a math operation. If it can detect numbers with a math operator in between, it will perform the calculation.

- We will get the operator directly the text of the selected math menu option
- Then, we will concatenate a string from the numbers and the operator
- Then, we will pass the string to eval(), which will perform the calculation.

39. Start by defining a new **calculate()** function, and switch over to calling this function, instead of calculateAnswer:

```
function calculate() {

    // 40. Get the number:
    let n1 = Number(num1.value);
    let n2 = Number(num2.value);

    // 41. Get the index of selected option and save it to a variable:
    let i = mathMenu.selectedIndex;
    console.log('i:', i);

    // 42. Get the text property of the selected option, which we look up
```

```
by index in the math menu's options array, and save it to a variable:
    let op = mathMenu.options[i].text;
    console.log('op:', op);

    // 43. Do the math as a concatenated string using the op variable:
    let ans = `${n1} ${op} ${n2}`;

    // 44. Output the answer; all we get is the literal string, but at
least it looks like a math problem:
    outBox.textContent = ans;

    // 45. Try again this time wrapping the string in eval():
    ans = eval(`${n1} ${op} ${n2}`);

    // 46. Output the answer again. This time it should work as expected:
    outBox.textContent = ans;
}
```

*** CHALLENGE SOLUTION START: *** ### ***

A. Make it a Timely Greeting, so that instead of "Hey, you!", we get "Good morning!" or other appropriate greeting for the time of day. If at least one name field is NOT blank, output the timely greeing along with the name(s):

```
let dt = new Date();
let hr = dt.getHours();
// hr = 19; // testing hr
let g = "";
if(hr < 12) {
    g = `Good Morning`;
} else if(hr < 18) {
    g = `Good Afternoon`;
} else {
    g = `Good Evening`;
}
```

B. Refactor the if-else as a ternary

```
outBox.textContent = !fName && !lName ? `${g}!` : `${g}, ${fName}
${lName}!`;

// *** ### *** CHALLENGE SOLUTION END *** ### ***
```

Google Unit Converter Clone Lite

Make a frontend JavaScript app (as html page) called **Google Units Converter Clone Lite**

You are provided w comps (graphic design and specs) for the app.
This is the attached jpg. You can also try out the real app by Googling Unit Converter.
Your job is to make it actually work!

UI (User Interface)

select menu with options of:

- **Temperature**
- **Distance**
- **Length**
- **Weight**
- **Time**

Conversion should work in both directions:

Temperature **(C <--> F)**
Distance **(Foot <--> Mile)**
Length **(CM <--> IN)**
Weight **(KG <--> LB)**
Time **(Sec <--> Min)**

Temperature is the default menu that appears on page load

under the menu are two input fields (text boxes), one for each unit in the conversion.

the **default** boxes which appear on page load boxes are :
Fahrenheit
Celsius
The boxes have starter-placeholder values of:
212 Fahrenheit
100 Celsius

UX (User Experience)

- The user chooses a unit from the select menu
- The user types a number in one box or the other.
- The user clicks the Convert button or hits enter
- The conversion / answer appears in the other box
- The user chooses another unit from the select menu and repeats

---

Conversion Formulas:

Temperature: Fahrenheit (F), Celsius (C)
F = C * 9/5 + 32
C = (F - 32) x 5/9

Distance: **mile (mi), feet (ft)**
1 mi = 5280 ft

Length: **inch (in), centimeter (cm)**
1 in = 2.54 cm

Weight **kilogram (kg), pound (lb), ounce (oz)**
1 kg = 2.20462 lb

Time **second (sec), minute (min), hour (hr), day**
1 min = 60 sec

Google Unit Converter Solution

There is no one way to do this project, but here is a pretty good solution.
Refer to the project prompt for background on what we are doing here.

Fetch the DOM objects:

1. Get the conversion type category menu:

```
const categoryMenu = document.getElementById('category-menu');
```

2. Get the two numeric input fields:

```
const v1 = document.getElementById('val1');
const v2 = document.getElementById('val2');
```

3. Get the menus below the input boxes. These un-selectable menus serve as the labels of the input boxes:

```
const unit1 = document.getElementById('unit1');
const unit2 = document.getElementById('unit2');
```

4. Get the h2 for displaying the conversion formula:

```
const formula = document.getElementById('formula');
```

5. Have a change to the category menu call the changeCategory function:

```
unitMenu.addEventListener('change', changeCategory);
```

Call convert() function whenever:

- user hits Enter after entering number
- user changes unmber by clicking stepper Call changeCategory() function whenever:
- user chooses from the category menu

6. Have the number fields call the convert function on change:

7. Define the convert() function:

```
function changeUnit() {

    // 8. Pass this.id -- the id of the the number field that called
the function -- and save it simply as id:
    let id = this.id;

    // 9. Set up the switch par of a switch-case-break, where the
category menu value is compared to various cases, looking for a match:
    switch(categoryMenu.value) {

        // 10. Check if the categoryMenu value is 'temperature':
        case 'temperature': // value1: fahrenheit; value2: celsius

        // 11. If the num1 box called the function:
        if(id == 'num1') {

            // 12. Set the nun2 box to the converted value:
            n2.value = ((n1.value - 32) * 5 / 9).toFixed(5);

        } else {

            // 13. Set the nun1 box to the converted value:
            n1.value = (n2.value * 9 / 5 + 32).toFixed(5);
        }
        break;

        // 14. Run the case-break on the other menu choices:
        case 'distance': // num1: mi; num2: ft
            if(id == 'num1') {
                n2.value = (n1.value * 3.28084).toFixed(5);
            } else {
                n1.value = (n2.value / 3.28084).toFixed(5);
            }
        break;

        case 'length': // num1: in; num2: cm
            if(id == 'num1') {
                n2.value = (n1.value * 2.54).toFixed(5);
            } else {
                n1.value = (n2.value / 2.54).toFixed(5);
            }
        break;

        case 'weight': // num1: kg; num2: lb
            if(id == 'num1') {
                n2.value = (n1.value * 2.20462).toFixed(5);
            } else {
                n1.value = (n2.value / 2.20462).toFixed(5);
            }
        break;

        case 'time': // num1: min; num2: hr
            if(id == 'num1') {
```

```
                n2.value = (n1.value * 60).toFixed(5);
            } else
                n1.value = (n2.value / 60).toFixed(5);
            }
        break;
    }
}
```

Call the changeCategory() function whenever:

- user chooses from the category menu

15. Define the changeCategory() function:

```
function changeCategory() {

    // 16. Set up a switch that considers the chosen category:
    switch(categoryMenu.value) {

        // 17. If the chosen category is 'temperature':
        case 'temperature':

            // 18. Display the starter temperature values:
            n1.value = '212'; // 212 F
            n2.value = '100'; // 100 C

            // 19. Set the text of the menu "labels". Since there is
only the one option, we can access it at index [0]:
            unit1[0].text = 'Fahrenheit';
            unit2[0].text = 'Celsius';

            // 20. Display the Formula text:
            formula.textContent = "F = C * 9/5 + 32";

        break;

        // 21. Move on to see if the chosen category is 'distance':
        case 'distance':

            // 22. Set the default, which is 1 as the left number:
            n1.value = '1'; // 1 meter
            n2.value = '3.28084'; // 3.28084 ft

            // 23. Set value and text of the menu "labels":
            unit1[0].text = 'Meter';
            unit2[0].text = 'Foot';

            // 24. Update the formula display
            formula.textContent = "1 m = 3.28084 ft";
        break;

        // 25. Repeat the above for the other three cases
```

```
        case 'length':
            n1.value = '1'; // 1 inch
            n2.value = '2.54'; // 2.54 cm
            unit1[0].text = 'Inch';
            unit2[0].text= 'Centimeter';
            formula.textContent = "1 in = 2.54 cm";
        break;

        case 'weight':
            n1.value = '1'; // 1 kg
            n2.value = '2.20462'; // 2.20462 lb
            unit1[0].text = 'Kilogram';
            unit2[0].text = 'Pound';
            formula.textContent = "1 kg = 2.20462 lb";
        break;

        case 'time':
            n1.value = '1'; // 1 hour
            n2.value = '60'; // 60 min
            unit1[0].text = 'Hour';
            unit2[0].text = 'Minute';
            formula.textContent = "1 hour = 60 min";
        break;

    }
}
```

Lesson 03.03

- forms and form elements
- event.preventDefault()
- form attributes: action and method
- "GET" vs. "POST"

We have a Login form for which the html and css are all ready. We will look at two different ways to process the form:

- on the page where the form itself is located
- on a separate processor page; this is the default

We don't have a database, so we can't compare the username and password typed into the form to any actual credentials in a database. Instead, we will just use local variables.

1. Open the html page in the browser; it's a typical login form, where the user enters their username and password and clicks a button to submit the form. The html page is currently using FINAL.js, so the form should work.

2. Fill out the form. The correct username is "Amy". The password is "abc".

3. Click the Log In button. This calls a function that compares what you entered to the correct username and password. If they match, you get a "Welcome" message; else the message is "Login failed".

4. In the html, notice that :

- `<form>` tag has no attributes. This is fine for now, but later, we will add attributes to it.
- `<input>` tags have id's for JS to get them.
- `<button>` inside a form submits the form by default on click.

5. Switch to using PROG.js, so that we can code all the functionality from scratch.

6. In PROG.js, declare variables for the username and password. Use any values you like. We'll use uppercase consts, since these values will not be changed:

```
const USER = 'amy';
const PSWD = 'abc';
```

7. Get the DOM elements: button, username and password inputs, and the h2:

```
const btn = document.querySelector('button');
const userBox = document.getElementById('username');
const pswdBox = document.getElementById('password');
const h2 = document.querySelector('h2');
```

8. Have the button call the login function on click:

```
btn.addEventListener('click', login);
```

event object

The **event** object is a default parameter passed to all functions. It is there even if not explicitly added as a parameter. The default name is **event**, but you can call the event object anything you like. Common aliases are **evt** and **e**, but renaming the event object requires that it be explicitly passed in as an argument of the function.

event.preventDefault()

When a button inside a form is clicked, it submits the form by default. This causes the page to redirect to the url specified by the form tag's **action** attribute, where a processor script is expected. If there is no action attribute, the page will reload in place. which resets the form and clears all its elements. We want to process the form on this page, so we need to prevent the default behavior which reloads the page and erases the username and password. To prevent the default, use the **event.preventDefault()** inside the login function.

9. Define the **login()** function, passing in the **event** argument and calling the **event.preventDefault()** method:

```
function login(event) {
    event.preventDefault();
```

10. Get the **username** and **password** from the form input fields. These are the **values** of the input objects:

```
 let user = userBox.value;
 let pswd = pswdBox.value;
```

11. Compare the username and password from the form to the correct username and password. There are two conditions to evaluate, so use the **&&** operator. If the credentials match, output a "Welcome" message and hide the input elements; else output "Login Failed":

```
    if(user === USER && pswd === PSWD) {
        h2.textContent = "Welcome " + USER;
        userBox.style.display = "none"; // hide username field
        pswdBox.style.display = "none"; // hide password field
        btn.style.display = "none"; // hide Log In button
    } else {
        h2.textContent = "Login Failed";
    }
} // end login function
```

leaving out event.preventDefault()

12. Comment out the **event.preventDefault()** line, and fill out the form again. It doesn't work, because the page reloads, which clears the username and password fields before JS has a chance to compare the values to the correct credentials.

form action: send form variables to a processor page

A form action attribute:

- has as its value the page where the form vars are sent: action="form-processor.html"
- since we are redirecting to another page on button click, the button no longer needs a click event to call a function

13. Comment out the listener for the button so that it no longer calls the login function on click.

14. In the form tag, add the action attribute: `**<form action="login-processor.html">**`

form method attribute

15. Also add a method="GET" attribute. This means that the form variables will travel via the URL to the login-processor.html processor page.
```
<form action="login-processor.html"
method="GET">
```

16. Fill out the form, and submit it again. We are redirected to login-processor.html.

17. Check the URL up in the browser address bar. Notice that the url ends with a question mark **"?"**. The part of the url that begins with the **?** is called the **querystring**.

name attribute of form elements

There are supposed to be name=value pairs following the question mark, specifically: **"?username=amy&password=abc"**. The reason these are absent is because these only occur for form elements that have a name attribute.

18. Go back to the html form and add name attributes to the two input elements.
    The name values can be anything, but it makes sense that they match the id's:
```
<input type="text" id="username" name="username" placeholder="username">
<input type="password" id="password" name="password"
placeholder="password">
```

19. Run the form again. This time up in the browser address bar at login-processor.html, we should see that the url ends in the querystring: **"?username=amy&password=abc"**

The code for accessing querystring variables involves arrays and loops--topics we have not covered yet--so instead, we will use another way:

sessionStorage for saving Session Variables

Session Variables are variables that we store in the browser, as oppposed in the current script page. This makes the vars available to any page in the application that is opened in the browser. Over in the form page,

we will save the username and password as session variables. This way, we will be able to access them on the processor page.

mouseover event

The **mouseover** event occurs whenever the cursor hovers over some DOM element, such as a button, div or image. As with any event, **mouseover** can call a function. The reason we want mouseover rather than click is because we want to make the session variables before the form is submitted on click.

20. Below the commented-out listener that calls the login function on click, add a listener that invokes a function called setSessionVars and runs on mouseover:

```
btn.addEventListener('mouseover', setSessionVars);
```

21. Define the **setSessionVars()** function:

```
function setSessionVars() {
    // 20. Call the setItem() method on the sessionStorage property,
passing it two arguments: a variable name in quotes and a value, with
the values being the username and password from the input boxes:
    sessionStorage.setItem("user", userBox.value);
    sessionStorage.setItem("pswd", pswdBox.value);
    console.log("Session Vars: user", userBox.value, "pswd",
pswdBox.value);
}
```

22. The JS for checking the login credentials is now going to be in the processor page, where we are redirected on submit. Switch to that page.

Form Processor Page

The following JS code is imported by **form-processor.html**, which processes the form, that is, checks the user log in.

- it gets the session variables set in the form page
- compares the session variables vars to the "real" username and password
- since we do not have a real database in which to look up the correct username and password, we have hard-coded those as constants, USER and PSWD

23. Get the session variables, and save them to "regular" vars:

```
let user = sessionStorage.getItem("user");
let pswd = sessionStorage.getItem("pswd");
console.log("session var user:", user);
console.log("session var pswd:", pswd);
```

24. Hard-code the correct username and password:

```
const USER = 'amy';
const PSWD = 'abc';
```

25. Get the h2 that displays the login response message:

```
const h2 = document.querySelector('h2');
```

Compare the session vars to the correct username and password:

26. First make sure the session vars exist; if they do not, the form was submitted with one or both inputs left blank in which case the login fails

```
if(user && pswd) {
    // 27. If the vars are set, next check to see if they match the
correct login credentials
    if(user == USER && pswd == PSWD) {
        // 28. Output a personalized welcome message on login success
        h2.textContent = `Welcome back, ${user}`;
    } else {
        // 29. Else, utput a "Login failed"
        h2.textContent = 'Login failed.';
    }
// 30. If no session vars exist, login also fails
} else {
    h2.textContent = 'Login failed.';
}
```

27. Back in the html log in page, in the form, change the method from "get" to "post": `<form action="login-processor.html" method="post">`

28. Re-run the form. Notice that now in the url of the processor page, there is no querystring, that is no **"?username=amy&password=abc"** This is because the post method transmits the form variables to the processor invisibly.

same-page processing vs. processor-page

To fully appreciate the difference between a form that is processed on the same page, as opposed to a form that redirects to a processor page, lets' go back to the same-page processin version.

33. Comment out the form tag with the action and method attributes and add replace it with an empty

    tag.

34. In the JS file, reactivate the button listener that calls the login function on click: **btn.addEventListener('click', login)**

35. In the **login(event)** function, reactivate the **event.preventDefault()** line.

36. Fill out and sumbit the form. This time, you stay on the same page.

form methods: GET vs POST

When a form is submitted, from variables as name=value pairs are sent to a script for processing. In the case of a login form the from variables would be something like username and password, or perhaps user and pswd, he values of which would be whatever the user entered into the login form.

"GET" : if a form has **method="GET"**, it means that:

- the form variables are transmitted to the processing page vias the URL querystring.
- right after the url of the page itself, there will be a question mark **?**, followed by one or more **name=value** pairs, connected by ampersands **&**:
- the **?** separates the url of the page itself from the querytstring
- **"GET"** is commonly used for HTTP requests that GET or request something, such as search results

"POST" : if a form has **method="POST"**, it means that the form variables are transmitted to the processing page invisibly -- they do not appear in the URL. **"POST"** is commonly used for HTTP requests that POST or write something, such as a registration form that logs a new user to the database.

"GET" and "POST" will become enormously important concepts when you get into **Node.JS**, where you will constantly be making "GET" and "POST" requests as you build server-side applications with database connectivity.

Lesson 03.04

RESTAURANT BILL TAX AND TIP CALCULATOR

The restaurant bill calculator consists of a web form:

- inputs for Food and Beverage subtotals
- select menus for Tip and Tax percent
- a Calculate button to call the **calcBill** function
- an h2 to display the itemized bill made by the function

1. Get the form objects:

```
let food = document.getElementById('food');
let bev = document.getElementById('bev');
let taxMenu = document.getElementById('tax-menu');
let tipMenu = document.getElementById('tip-menu');
let calcBtn = document.getElementById('calc-btn');
let response = document.getElementById('response');
```

2. Have the button listen for a click and call the calcBill function when it is clicked:

```
calcBtn.addEventListener('click', calcBill);
```

3. Start writing the function. First thing it does is prevent the form from doing a default page reload:

```
function calcBill(event) {
    event.preventDefault();
```

4. Get the values from the form.

- Use the Number() method to convert the "number-like strings" to real numbers. Add the food and bev together to obtain the sub-total:

```
let foodTot = Number(food.value);
let bevTot = Number(bev.value);
let tipPct = Number(tipMenu.value);
let taxPct = Number(taxMenu.value);
let subTot = foodTot + bevTot;
```

5. Calc the tax and tip totals by mutiplying the subTot by the tax and tip rates. Since 15% is 0.15 (not 15) divide by 100:

```
let taxTot = subTot * taxPct / 100;
let tipTot = subTot * tipPct / 100;
```

6. Add the tax and tip to the subtotal and round off tipTot, taxTot and total to two decimal places with toFixed(2). Be sure to do toFixed() AFTER the math is done, because toFixed() converts number to strings:

```
let total = subTot + taxTot + tipTot;
tipTot = tipTot.toFixed(2);
taxTot = taxTot.toFixed(2);
total = total.toFixed(2);
```

7. Get the date and time, so that they can be added to the bill. This is all review of Unit 02.04 - Date Object, so we won't discuss the code again here:

```
let d = new Date();
let hr = d.getHours();
let min = d.getMinutes();
let sec = d.getSeconds();
if(min < 10) min = '0' + min;
if(sec < 10) sec = '0' + sec;
let timeIs = `${hr}:${min}:${sec}`;
let mo = d.getMonth(); // 0-11
let date = d.getDate(); // 1-31
let yr = d.getFullYear();
let dateIs = `${mo+1}/${date}/${yr}`;
```

8. Concatenate the itemized bill and output it. The string includes
 tags, so use **innerHTML**, rather than **textContent**:

```
// concatenate bill
let bill = `
    ***JS CAFE***
    <br>***GUEST CHECK***
    <br>Date: ${dateIs}
    <br>Time: ${timeIs}
    <br>Food: $${foodTot}
    <br>Beverage: $${bevTot}
    <br>Sub-Total: $${subTot}
    <br>Tax %: ${taxPct}
    <br>Tax: $${taxTot}
    <br>Tip Pct %: ${tipPct}
    <br>Tip: $${tipTot}
    <br>***PLEASE PAY:***
    <br>TOTAL: $${total}
    <br>***THANK YOU!***`;
```

```
    // output the bill to the web page (DOM)
    response.innerHTML = bill;
} // end calcBill function
```

Lesson 03.05

- Hoisting: Function Declarations vs. Expressions
- Anonymous Functions
- Inline Anonymous Callback Functions
- onclick property of DOM objects

hoisting

- Hoisting refers to functions being lifted automitically to the top of their scope.
- Hoisting lets functions be called from lines above where they are declared.
- Hoisting does not apply to variables--only functions.
- To use a variable, it must have already been declared.

1. Declare a variable, fruit, and try to log it from the line above:

```
console.log(fruit); // Error:  Cannot access 'fruit' before
initialization
let fruit = "kiwi";
```

We get an error, because "regular variables" are not hoisted.

2. Comment out the error, and log fruit after it is declared. Now, it works:

```
console.log(fruit);
```

3. Declare a function, and then call it from above and below the declaration:

```
console.log(sayHello('down there!'));

function sayHello(name) {
    return `Hey, ${name}!`;
}

console.log(sayHello('up there!'));
```

Both function calls work, because the function itself has been hoisted.

function experessions and anonymous functions

- A variable with a function for its value is known as a **function expression**.
- As a variable, a **function expression** doesn't get hoisted.
- A **function expression** must be defined before it can be called or "listened for".
- To make a **function expression**, set a variable equal to an unnamed function.
- A function without a name is known as an **anonymous function**.

4. Write a function expression, and then call it from above and below:

```
console.log(welcomeUser('Jane1')); // ReferenceError: welcomeUser is
not defined

const welcomeUser = function(user) {
    return `Welcome ${user}!`;
}

console.log(welcomeUser('Jane2'));
```

We get an error for the first attempt to call welcomeUser, because a function expression is first and foremost a variable--and variables don't get hoisted.

callbacks: functions as arguments of other functions

A function can take any kind of argument: string, number, boolean--even another function. A function passed to another function as its argument is known as a **callback function**, or simply a **callback**. We have already been working with callbacks without referring to them as such. The addEventListener method takes two arguments: an event, such as "click", and a callback to execute when said event occurs.

5. Get the four buttons, each of which will call a different type of function:

- function declaration: starts with keyword "function" (btn1)
- function expression: variable set equal to anonymous function (btn2)
- inline function: an anonymous function as callback (btn3)
- onclick: a DOM object property set equal to an anonymous function (btn4)

```
const btn1 = document.getElementById('btn-1');
const btn2 = document.getElementById('btn-2');
const btn3 = document.getElementById('btn-3');
const btn4 = document.getElementById('btn-4');
```

6. Get the four span tags, each of which will hold output from one of the four functions:

```
const out1 = document.getElementById('out-1');
const out2 = document.getElementById('out-2');
const out3 = document.getElementById('out-3');
const out4 = document.getElementById('out-4');
```

Button 1: Calling Function Declaration

7. Instruct the first button to call a function when clicked. The function to call on click, that second argument, is a **callback**:

```
btn1.addEventListener('click', onBtn1Click);
```

8. Declare the onBtn1Click function, which outputs a message to the first span tag:

```
function onBtn1Click() {
    out1.textContent = 'Function Declaration says "Hey!"';
}
```

9. Reload the page, and click BTN 1 to get the message.

Button 2: Calling Function Expression

10. Write a function expression by declaring a variable by the name of onBtn2Click and setting it equal to an anonymous function, which outputs a message to the second span tag:

```
const onBtn2Click = function() {
    out2.textContent = 'Function Expression says "Hola!"';
}
```

11. Instruct the second button to call the function when clicked. Notice that we had to do the listener AFTER the expression, since function expressions are variables--and variables cannot be refernced before they are declared.

```
btn2.addEventListener('click', onBtn2Click);
```

12. Reload the page, and click BTN 2 to get the message.

Inline Anonymous Functions

In a function expression, a variable is set equal to a function. Since the variable provides the name, the function value itself is anonymous. An anonymous function can also be a callback, that is, the argument of another function. The addEventListener method takes a callback as its second argument.

We will instruct the third button to run an anonymous function inline, that is, right there inside the addEVentListener method. This is in contrast to our usual practice of having the addEVentListener call an external function by name.

Button 3: Running an Inline Anonymous Function

13. Have the third button listen for a click. When the click occurs, run an **inline anonymous function** right there on the spot:

```
```

```
btn3.addEventListener('click', function() {
    out3.textContent = 'Inline anon function says "Hi!"';
})
```

14. Reload the page, and click BTN 3 to get the message.

onclick property

Any **event** that can be called on a DOM object co-exists as a property of that object. A button can run a function on "click", and therefore has an **onclick** property. If you set the onclick property equal to an anonymous function, when you click the button, the function will run.

Button 4: onclick property = anonymous function

15. Set the fourth button's onclick property equal to an anonymous function:

```
btn4.onclick = function() {
    out4.textContent = 'onclick anon function says "Yo!"';
}
```

16. Reload the page, and click BTN 4 to get the message.

Lesson 03.06 - PROG

Number Guessing Game

In this lesson we will write a number guessing game program:

- Player clicks PLAY button to call **playGame** function
- **playGame** function hides the PLAY button and
- shows the input box for entering guess
- shows the GUESS button for submitting guess
- shows an h2 for displaying feedback
- Program generates a mystery random integer from 1-100
- Player inputs a number by typing or using stepper arrows
- Player clicks GUESS button to call **evalGuess** function
- **evalGuess** function compares guess to mystery number
- feedback is displayed: Too High or Too Low or Congrats
- function keeps track of total guesses
- Once the mystery number is guessed:
- the score (GamesPplayed and Guess Avg) is updated
- the **resetGame** function is called automatically

resetGame function:

- updates the score
- shows playBtn; hides guessBox, guessBtn and feedback
- We are ready for a new game: click PLAY AGAIN

1. Declare game play variables. These need to be in the global scope, since the vars are used by more than one function.

```
let r = guess = guesses = games = avg = 0;
```

The above shorthand requires that the variables have the same initial value.

2. Get the PLAY and GUESS buttons (GUESS is hidden on page load):

```
const playBtn = document.getElementById('play-btn');
const guessBtn = document.getElementById('guess-btn');
```

3. Have the buttons call their respective functions:

```
playBtn.addEventListener('click', playGame);
guessBtn.addEventListener('click', evalGuess);
```

4. Get the guess input box and feedback h2, both of which are also hidden on page load:

```
const guessBox = document.querySelector('input');
const feedback = document.getElementById('feedback');
```

5. Get the footer spans for keeping score:

```
const gamesSpan = document.getElementById("games-span");
const avgSpan = document.getElementById("avg-span");
```

The playGame function runs when the PLAY button is clicked.

6. Declare the playGame function:

```
function playGame() {
```

7. Generate a random mystery number from 1-100:

```
r = Math.ceil(Math.random() * 100);
console.log('mystery number:', r);
```

8. Hide the PLAY button

```
this.style.display = "none";
```

9. Show the guess input box, GUESS button and feedback h2:

```
guessBox.style.display = "inline";
guessBtn.style.display = "inline";
feedback.style.display = "inline-block";
```

10. Output a prompt to the feedback h2:

```
    feedback.textContent = "Guess the mystery number from 1-100";

} // end function playGame()
```

The evalGuess function runs when GUESS button is clicked.

- if the guess is too low, output: "Guess is too LOW"
- else if the guess is too high, output: "Guess is too HIGH"
- else guess is correct, so "Congrats!" (Game Over)
- the resetGame() function is called when the game is over

11. Declare the playGame function:

```
function evalGuess() {
```

12. Increase guesses by 1

```
guesses++;
```

13. Get the player's guess from the input box and convert it to an actual number:

```
guess = Number(guessBox.value);
```

14. If the guess is less than the mystery number:

```
if(guess < r) {
    feedback.textContent = "Guess is too LOW!";
```

15. If the guess is greater than the mystery number:

```
} else if(guess > r) {
    feedback.textContent = "Guess is too HIGH!";
```

16. If neither too high nor too low, the guess is correct:

```
} else {
    feedback.innerHTML = `Congrats! You guessed<br>the mystery
number: ${r}`;
```

17. Game's over, so call the resetGame function:

```
        resetGame();
    }

} // end function evalGuess()
```

The resetGame function runs automatically when the player guesses correctly.

18. Declare the playGame function:

```
function resetGame() {
```

19. Reset the player guess and guess input box to 0:

```
guess = 0;
guessBox.value = 0;
```

20. Increment total games played by 1

```
games++;
```

21. Hide the guess input box and guess button:

```
guessBox.style.display = "none";
guessBtn.style.display = "none";
```

22. Show playBtn; this time have it say PLAY AGAIN:

```
playBtn.style.display = "inline-block";
playBtn.textContent = "PLAY AGAIN";
```

23. Update the score average (guesses per game):

```
avg = guesses / games;
```

24. Update the score Games Played and Guess Avg:

```
    gamesSpan.textContent = games;
    avgSpan.textContent = avg.toFixed(2);

} // end function resetGame()
```

Lesson 03.07

Apartment Rent Estimator

In this form, we present the user with two select menus and two checkboxes, from which they choose their desired apartment configuration. Each choice has a price: the more bedrooms, bathrooms and amenities chosen, the higher the rent. The choices are:

- Number of Bedrooms, chosen from a select menu.
- Number of Baths, also chosen from a select menu.
- Doorman Building checkbox
- Parking Space checkbox
- Fitness Center checkbox
- Skyline View checkbox

The html and css are already done, but of course the form doesn't work yet. We will write the JS that powers the form.

The select menus are for choosing numbers of Bedrooms and Baths. The greater the choice, the higher the value (rent).

- The 1 Bath choice has a value of 0, so choosing that will not raise the rent.
- One bedroom is pre-selected by default, even though it is not the first menu choice.

checkboxes

The fees associated with the checkboxes will, of course, only be charged if the checkbox is checked. We will first assess the flat fees and add those to the rent. Then we will calculate the percent surchargess on top of the rent.

- **Parking** (id="parking") value = 350, for the $350 parking fee
- **Fitness Center** (id="gym") value = 100, for the $100 gym fee
- **Skyline View** (id="view") value = 0.25, for the 25% surcharge for a skyline view.
- **Doorman Building** (id="doorman") value = 0.1, for the 10% surcharge for a doorman building.
- The html also includes a **CALCULATE RENT** button and an h3 for posting the answer.

1. Get the button and have it call a function when clicked:

```
const calcBtn = document.querySelector('button');
calcBtn.addEventListener('click', calculateRent);
```

2. Get the "feedback" tag:

```
const feedbackTag = document.getElementById('feedback');
```

3. Declare the function. Start with the preventDefault() method, so that the button click does not reload the page and reset the form:

```
function calculateRent(event) {
```

4. Prevent the button from submitting the form, which would reset it:

```
event.preventDefault();
```

5. Get the values of the select menus. We don't need the whole objects--just their values. The values come in as strings, so convert them to numbers:

```
let bdrms = Number(document.getElementById('bdrms').value);
let baths = Number(document.getElementById('baths').value);
let clsts = Number(document.getElementById("closets").value);
```

6. Add the values from the menus to get the rent, not including any applicable checkbox fees:

```
let rent = bdrms + baths + clsts;
```

7. Get the checkboxes

```
const doormanCB = document.getElementById('doorman');
const parkingCB = document.getElementById('parking');
const viewCB = document.getElementById('view');
const gymCB = document.getElementById('gym');
const balconyCB = document.getElementById("balcony"); // +$100
const petCB = document.getElementById("pet"); // +5%
```

Checkboxes have a .checked property, which returns a boolean. Write if-statements to see if the checkboxex are checked.
Start with the flat-fee checkboxes:

8. If the parking checkbox is checked, add $350 to the rent:

```
if(parkingCB.checked) {
    rent += Number(parkingCB.value); // +350
}
```

9. If the gym checkbox is checked, add $100 to the rent:

```
if(gymCB.checked) {
    rent += Number(gymCB.value); // +100
}

if(petCB.checked) {
    rent += Number(petCB.value); // +50
}
```

Next, evaluate the percent fee checkboxes. Calculate the individual fees, and then add them all to the rent:

10. Declare variables to store the percent fees:

```
let doormanFee = viewFee = balconyFee = 0;
```

11. A doorman building incurs a 10% surcharge, so to get the doorman fee, multiply the rent by the value of the doorman checkbox, which is 0.1 (10%):

```
if(doormanCB.checked) {
    doormanFee = rent * doormanCB.value; // +10%
}
```

12. An apartment with a skyline view is 25% more expensive than one without such a view, so to get the view fee, multiply the rent by the value of the view checkbox, which is 0.25 (25%):

```
if(viewCB.checked) {
    viewFee = rent * viewCB.value; // +25%
}

if(balconyCB.checked) {
    balconyFee = rent * balconyCB.value; // +25%
}
```

13. Add the percent fees to the rent and output the final rent, rounded off to the nearest integer:

```
rent += doormanFee + viewFee + balconyFee;
feedbackTag.textContent = `Estimated rent: $${Math.round(rent)}

} // end function calculateRent(event)
```

14. Run the page in the browser and test various configurations, from the cheapest to the most expensive, and a few in between.

CHALLENGE: ADD 2 checkboxes and a select menu

- Checkboxes: **Balcony (+10%)**, **Pet-Friendly (+$100)**
- Select Menu: **Closets:**: 2 closets standard (no fee); Additional closets (up to 5) cost $150 extra each

Lesson 03.08

Keyboard Events

- The pressing of a key is a **keydown** event.
- The release of a key is a **keyup** event.
- As with any event, a keyboard event can call a function.
- Keyboard events are listened for by the document.
- Syntax: **document.addEventListener('keyup', getKey)**
- The **event.key** property returns the name of the key
- If the key is the one we are listening for, run some function

1. Get the DOM elements: the container, the two boxes in the upper corners for displaying output, and the spaceship:

```
const container = document.querySelector('.container');
const keyBox = document.getElementById('key-box');
const pinBox = document.getElementById('pin-box');
const spaceShip = document.getElementById('space-ship');
```

2. Set the left position of the spaceship to equal half the window width, minus half the width of the spaceship. This puts the spaceship is the middle of the screen to start:

```
let leftPos = window.innerWidth / 2 - 128;
spaceShip.style.left = leftPos + 'px';
```

3. Set booleans to keep track of the font and dark mode states, which are toggled by pressing "f" and "d", respectively:

```
let serif = false;
let dark = false;
```

4. Set the speed of the spaceship. Each time the left or right arrow is pressed, the spaceship:

```
let speed = 20;
```

5. Have the document listen for the keyup event. On keyup, run a function that outputs the key and code to the keyBox. Since the event object is used by the function, pass the event object into the function as its argument. Since the function is so short, write it as an inline anonymous function, as opposed to an external, named function:

```
document.addEventListener('keyup', function(event) {
keyBox.innerHTML = `Key: ${event.key}<br>Code: ${event.code}`;
});
```

6. Have document listen for the keydown event. When the event takes place -- which is when ANY key is pressed -- call the onKeyPress function.

```
document.addEventListener('keydown', onKeyPress);
```

7. Define the onKeyPress function, passing in the event object:

```
function onKeyPress(event) {
```

8. Output the key and code that was pressed:

```
keyBox.innerHTML = `Key: ${event.key}<br>Code: ${event.code}`;
```

Check if the key is 'c', 'd', 'p', 'n' or the left or right arrow.

setting random background color

9. Check if the key is "c" for "color", or if the spacebar was pressed:

```
if(event.key == 'c' || event.code == 'Space') {

    // 10. Generate three integers in the 0-255 range
    let R = Math.floor(Math.random() * 255);
    let G = Math.floor(Math.random() * 255);
    let B = Math.floor(Math.random() * 255);
```

10. Concatenate the numbers into the rgb() method:

```
let randRGB = `rgb(${R}, ${G}, ${B})`;

// 12. Set the body background color to the random RGB color:
document.body.style.backgroundColor = randRGB;
```

Alternatively, we ca generate the random color as a hex value, in which case we only need one random number.

13. Comment out steps 10-12, and make a new random integer in the 0 - 16777215 range (256 x 256 x 256):

```
let r = Math.floor(Math.random() * (256 ** 3));

// 14. Convert the random number to a base-16 (0-9, A-F) hexadecimal
value by calling the toString(16) method on the number.
// let randHex = "#" + r.toString(16); // hexify the number
let randHex = getRandHexColor();

// 15. Set the body background color to the random hex color:
// document.body.style.backgroundColor = randHex;
document.body.style.backgroundColor = randHex;

// 16. Set the pinBox text and color to the random color:
// pinBox.style.color = randHex;
pinBox.style.color = randHex;
// pinBox.textContent = randHex;
pinBox.textContent = randHex;
```

Toggling between Dark Mode and Light Mode for the container

17. Check if the key is "d" for "dark":

```
} else if(event.key === 'd') {

    // 18. If dark boolean is currently false:
    if(!dark) {

    // 19. Switch to dark mode by adding and removing classes:
    container.classList.remove('light-mode');
    container.classList.add('dark-mode');

    // 20. Else, dark mode is already true
    } else {

    // 21. Switch to light mode:
    container.classList.remove('remove-mode');
    container.classList.add('light-mode');
    }

    // 22. Flip the dark boolean:
    dark = !dark;
```

18. Check if the key is "f" for "font":

```
} else if(event.key === 'f') {
```

```
    // 24. If serif boolean is currently false:
    if(!serif) {

    // 25. Set the body font family to 'serif':
    document.body.style.fontFamily = 'serif';

    // 26. Else, serif boolean is already true:
    } else {

    // 27. Set the body font family to 'sans-serif':
    document.body.style.fontFamily = 'sans-serif';
    }

    // 28. Flip the serif boolean:
    serif = !serif;
```

19. Check if the key is "p" for "pin" or "n" for "number":

```
} else if(event.key === 'p' || event.key === 'n') {

    // 30. Generate a random number in the 0-9999 range:
    let r = Math.floor(Math.random() * 10000);

    // All pin numbers need to be four digits, so we need to add
leading zero(es) to numbers in the 0-999 range.

    // 31. Declare a variable to store the pin as a string:
    let pin;

    // 32. If r is 0, set pin to be four zeroes:
    if(r == 0) {
    pin = "0000";

    // 33. else if r is less than 10, add three leading zeroes:
    } else if(r < 10) {
    pin = "000" + r;

    // 34. else if r is less than 100, add two leading zeroes:
    } else if(r < 100) {
    pin = "00" + r;

    // 35. else if r is less than 1000, add one leading zero:
    } else if(r < 1000) {
    pin = "0" + r;

    // 36. else r is a four-digit number, so use r as the pin:
    } else {
    pin = r;
    }

    // 37. Output the pin to the "pin box":
    pinBox.innerHTML = 'PIN:<br>' + pin;
```

20. If the Left Arrow was pressed:

```
```js
} else if(event.code == "ArrowLeft") {  // OR: event.key == "ArrowLeft"

    // 39. If the spaceship isn't already all the way left:
    if(leftPos > 0) {

    // 40. Reduce the leftPos value by the speed (default is 20):
    leftPos -= speed;

    // 41. Move the spaceship left by the speed value:
    spaceShip.style.left = leftPos + 'px';

    }
```
```

42. If the Right Arrow was pressed:

```
```js
} else if(event.code === "ArrowRight") { // OR: event.key == "ArrowRight"

    // 43. If the spaceship isn't already all the way to the right (250px
from the right end of the window):
    if(leftPos < window.innerWidth - 250) {

    // 44. Increase the leftPos value by the speed (default is 20):
    leftPos += speed; // subtract 10 from leftPos

    // 45. Move the spaceship right by the speed value:
    spaceShip.style.left = leftPos + 'px';
    }
```
```

46. If the key pressed is NOT 'c', 'd', 'p', 'n', left arrow or right arrow

```
```js
} else {

    // 47. Output a message to the pin box, so at least something happens:
    pinBox.textContent = "Nothing doing!";
    }
```
```

```
}
```

Random Hex Color

- A base 10 number uses the digits 0123456789 only
- A base-16 value uses the digits 0123456789ABCDEF only
- A hexadecimal color is a base 16 value
- The **toString()** method converts a number to a string
- **toString(16)** converts a base 10 number to a base 16 string
- There exist 16,777216 RGB colors (256 x 256 x 256)
- Call toString(16) on an int from 0-16777215 to get a hex color
- Put **'#'** before the number to complete the hex color

48. Declare a function that does ONE thing: makes a random hex color:

```
function getRandHexColor() {

    // 49. Generate a random 16-digit float from 0-16777215
    let r = Math.floor(Math.random() * 256 ** 3);

    // 50. Make a base-16 hexadecimal color from the random number and
return it
    let randHex = '#' + r.toString(16);

    // 51. return the hexadecimal color value
    return randHex;

}
```

Switch to using the **getRandHexColor()** function:

52. Comment out steps 10-13

53. In step 14, set **randHex = getRandHexColor()**

54. In steps 15 and 16, set all three values to **randHex**

Lesson 03.09 - Image Zoom, IEFE

- Image Zoom, IIFE, mousemover

Image Zoom

In this project, we will make an Image Zoomer:

- The UI consists of two divs and an image
- The user mouses over the image
- An empy framed box called "zoom-window" follows the mouse
- The other div shows the closeup from the "zoom-window"
- The function for this will be an IIFE

IIFE (Immediately Invoked Function Expression)

An IIFE (Immediately Invoked Function Expression):

- automatically runs as soon as it is declared.
- has no name: it's just function()
- is entirely wrapped in parentheses
- is called by another set of parentheses at the end
- runs immediately--but only once
- cannot be called again later, since it has no name

1. Declare the IIFE: everything is wrapped in parentheses, and then the function is called by a set of empty parentheses, right at the end:

```
/*   (function() {
        // do stuff;
    })()

*/

(function() {
```

2. Get the image:

```
const img = document.getElementById("image");
```

3. Have the image call a function on mousemove, that is, whenever the mouse is moved over the image, a function is called:

```
img.addEventListener("mousemove", moveView);
```

4. Get the "zoom-window" div. This is the div that follows the cursor as a little framed box:

```
const zoomWindow = document.getElementById("zoom-window");
```

5. Have the zoomWindow also call the same function. Having the image and the framed zoom window box both call the function makes the zoom effect smoother:

```
zoomWindow.addEventListener("mousemove", moveView);
```

6. Get the zoom image div for displaying the zoomed-in close-up of the image:

```
const zoomImage = document.getElementById("zoom-image");
```

7. Declare variables for the offset width and the offset height. These are for defining the zommed-in portion of the image:

```
let oW = zoomImage.offsetWidth / zoomWindow.offsetWidth;
let oH = zoomImage.offsetHeight / zoomWindow.offsetHeight;
```

8. Set the zoomImage div background image to be the image we want to zoom in on. The image should be MUCH larger than the zoom image div that contains it, since the zoom image shows a detail.

```
zoomImage.style.backgroundImage = "url(images/_chinese-day-bed-
2.jpg)";
```

9. Set the background size (don't worry about what the math is doing). The backgroundSize property has two values: width and height:

```
zoomImage.style.backgroundSize = (img.width * oW) + "px " +
(img.height * oH) + "px";
```

position properties: pageX, pageY, pageXOffset, scrollX

- the event object has pageX and pageY properties pageX and pageY store the (x, y) coordinates of the cursor
- pageXOffset property, equal to the scrollX property returns how far the document has been scrolled from the top of the window.
- the offsetWidth property returns the viewable width of an element (in pixels) including padding, border and scrollbar, but not margin

getBoundingClientRect()

- the getBoundingClientRect() method is called on a DOM object and returns an object of 8 properties, pertaining to the object's size and position: **left, top, right, bottom, x, y, width, height**

Understanding (or not understanding) the math:

These various properties are ingeniously tapped to create the zoom effect. You are free to analyze the math on your own, but suffice it to say, there are many such "widget" projects where some or most of the math and logic may be hard to grasp. The thing to know is not the nuances of the math, but the fact that such widgets as the image zoomer exist, are Googleable, and can, for the most part be copy-pasted and then customized as needed--we don't usually need to mess with the core math.

10. Declare the moveView function that is called when the mouse if over the image and/or the zoom window that follows the cursor:

```
function moveView(event) {
```

11. Make the zoomWindow (the box that follows the mouse) appear:

```
zoomWindow.style.visibility = "visible";
```

12. Get the 8 properties of

```
bounds = img.getBoundingClientRect();
```

13. Declare a variable, x, set equal to a fairly complicated math expression:

```
let x = event.pageX - bounds.left - window.pageXOffset -
(zoomWindow.offsetWidth / 2);
```

14. If x is greater than the image's width, subtract the zoom window's offset width from x:

```
if (x > img.width - zoomWindow.offsetWidth) {
    x = img.width - zoomWindow.offsetWidth;
}
```

15. If x less than 0, set x to 0:

```
if (x < 0) {
    x = 0;
```

```
}
```

16. Set the left position of the zoom window to be x pixels:

```
zoomWindow.style.left = x + "px";
```

17. Repeat the "x calculations" for y:

```
let y = event.pageY - bounds.top - window.pageYOffset -
(zoomWindow.offsetHeight / 2);

if (y > img.height - zoomWindow.offsetHeight) {
    y = img.height - zoomWindow.offsetHeight;
}

if (y < 0) {
    y = 0;
}

zoomWindow.style.top = (y+60) + "px";
```

18. Set the zoom image background position as negative x and y values:

```
    zoomImage.style.backgroundPosition =
    "-" + (x * oW) + "px -" + (y * oH) + "px";
}

})();
```

Lesson 04.01

arrays - index, array.length

const vs. let

array methods: push(), sort(), pop(),

nested / 2D arrays

arrays

Arrays are variables that can hold more than one value at a time. Arrays have a datatype of **object**. Here are some key things to know about arrays:

- array items exist as a list, surrounded by square brackets: **[item1, item2, item3]**
- each item is assigned a position number, or **index**, with the first item at index 0
- array items can be of any data type
- items of different data types can be in the same array: **['apples', 4, true, 3.5]**

using const to declare an array

As we have learned, primitive variables (strings, numbers, booleans) declared with **const** cannot be modified (changed) in any way.

1. Declare a constant with **const**, and try to change it. We get an error:

```
const DOZEN = 12;
DOZEN = 13; // ERROR: Assigment to constant variable
```

const for objects

Unlike primitives, arrays declared with **const** *can* be modified--to an extent. Items can be changed, added or deleted, but the array bject itself cannot be **mutated** to a different datatype. So, with **const**, once an array, always an array.

To declare an array, start with **let** or **const**, followed by a name. We will start with **let** in order to later demonstrate our point about mutability. The array value consists of a pair square brackets containing items separated by commas.

It is a good practice to pluralize array names: **fruits** not **fruit**. Adding **Arr** further clarifies that this is an array: **fruitsArr**.

2. Declare an array called **fruitsArr**, with three items:

```
let fruitsArr = ['apple', 'banana', 'cherry'];
console.log(fruitsArr);
```

3. Check the Console. Expand the arrow to see the numbered items, with 'apple' at index 0.

index

Array items are stored in a numeric order, called **index**.

- The first item in an array is at index 0 the second item is at index 1, and so on.
- Use square bracket syntax to get individual array items: **array[0]**

4. Get the first item at index 0 and the last item at index 2:

```
console.log(fruitsArr[0]); // apple
console.log(fruitsArr[2]); // cherry
```

setting array values by index

To set the value of an array item, refer to it by index:

5. Replace the first item, 'apple', with 'peach':

```
fruitsArr[0] = 'peach';
console.log(fruitsArr);
// ['peach', 'banana', 'cherry']
```

One way to add an item at the end of the array is to assign it by index:

6. Add a fruit to the end of the array, at index 3:

```
fruitsArr[3] = 'mango';
console.log(fruitsArr);
// ['peach', 'banana', 'cherry', 'mango']
```

mutating an array (changing its datatype)

A downside of **let** for arrays is that it does not protect the data type. You could inadvertently change the array to a string or any other datatype. Use 'const' to prevent such a mistake from occurring.

7. Change fruitsArr to a string. Just like that, no more array:

```
fruitsArr = "oops";
console.log(fruitsArr); // oops
```

8. Declare another fruits array, this time with 'const'. We need a new name, as we cannot redeclare an existing 'const' or 'let':

```
const fruits = ['apricot', 'pear', 'kiwi', 'grape'];
```

9. Try to mutate the array into a string. You get an error:

```
fruits = "whoops";
// ERROR: Assignment to constant variable
```

array.length

The **length** property of an array returns the number of items in the array.

10. Get the length of the array:

```
console.log(fruits.length); // 4
```

array.length - 1

- The first item in an array is at index 0.
- **array[0]** gets the first item.
- The last item is at index **array.length - 1**.
- **array[arrray.length - 1]** gets the last item.

11. Save the last item in fruits to a variable:

```
let lastFruit = fruits[fruits.length - 1];
console.log(lastFruit); // grape
```

12. Try that again, but without subtracting 1. We get 'undefined', because there is no index 4. Indexing starts at 0, so the last of the 4 items is at index 3:

```
lastFruit = fruits[fruits.length];
console.log(lastFruit); // undefined
```

random array items

To select an item at random from an array:

- generate a random integer from 0 to array.length-1
- pass that number to the array square brackets

13. Get a random fruit from fruits:

```
let r = Math.floor(Math.random() * fruits.length);
console.log(r, fruits[r]); // 2 kiwi
```

Run it a few times to see that the fruit keeps changing.

array methods

The array object has numerous methods, some of which we will learn about now:

array.push(item)

The 4th and last item of the fruits array is at index 3. Therefore, a 5th item would occupy index 4.

14. Add a 5th fruit to the array:

```
fruits[4] = 'banana';
console.log(fruits); // ['apricot', 'pear', 'kiwi', 'grape', 'banana']
```

That worked, but hard-coding numbers is to be avoided, since the length of the array has just changed. Next time, the new item would be at index 5, and so on. More dynamically, we could use **array.length** as the index.

15. Add a 6th and 7th fruit using **array.length** as the dynamic index of the new, last item:

```
fruits[fruits.length] = 'pineapple';
console.log(fruits); // ['apricot', 'pear', 'kiwi', 'grape', 'banana',
'pineapple']
fruits[fruits.length] = 'papaya';
console.log(fruits); // ['apricot', 'pear', 'kiwi', 'grape', 'banana',
'pineapple', 'papaya']
```

With fruits.length, we avoid hard-coding the index as we add more items. But there's yet a better way: array.push(item)

The **push()** method takes one or more items as its argument and adds it/them to the end of the array.

16. Using the **push()** method, add 'lime' to the end of the array.

```
fruits.push('lime');
console.log(fruits); // ['apricot', 'pear', 'kiwi', 'grape', 'banana',
'pineapple', 'papaya', 'lime]
```

17. Push more than one item at a time:

```
fruits.push('lemon', 'blueberry');
console.log(fruits); // ['apricot', 'pear', 'kiwi', 'grape', 'banana',
'pineapple', 'papaya', 'lime', 'lemon', 'blueberry']
```

array.sort()

Called on an array of strings, the **sort()** method puts the items in alphabetical order. Sort does not return (make) a new array; it operates in place on the existing array:

18. Sort the fruits array:

```
fruits.sort();
console.log(fruits);
// ['apricot',  'banana', 'blueberry', 'grape' 'kiwi',  'lemon',
'lime', 'papaya', 'pear', 'pineapple']
```

array.pop()

The **pop()** method removes the last item from the array. It also returns the item, so it can be saved to a variable.

19. Pop the last item from the array. Log the array to see that 'pineapple' is gone:

```
fruits.pop();
console.log(fruits);
// ['apricot',  'banana', 'blueberry', 'grape' 'kiwi',  'lemon',
'lime', 'papaya', 'pear']
```

20. Pop again, but this time save the popped item to a variable and then log it:

```
let poppedItem = fruits.pop();
console.log(poppedItem); // pear
console.log(fruits);
// ['apricot',  'banana', 'blueberry', 'grape' 'kiwi',  'lemon',
'lime', 'papaya']
```

declaring an empty array

An empty array is declared as just a pair of empty square brackets--no items.

21. Declare a new, empty array:

```
const veggies = [];
```

22. Push a few items into the array:

```
veggies.push('carrot', 'beet', 'arugula', 'cucumber');
veggies.push('kale', 'broccoli', 'lettuce', 'celery');
console.log(veggies); // ['carrot', 'beet', 'arugula', 'cucumber',
'kale', 'broccoli', 'lettuce', 'celery']
```

23. Sort the veggies array:

```
fruits.sort();
console.log(veggies); // ['arugula', 'beet', 'broccoli', 'carrot',
'celery', 'cucumber', 'kale', 'lettuce']
```

arrays of numbers

The simple sort() method, without arguments, works on strings and so regards numbers as strings and alphabetizes them. As a string, "1" is less than "2", as expected, but "100" is also less than "2", just as "app" is less than "b".

24. Declare an array of numbers:

```
const nums = [3, 5, 220, 2, 17, 105, 45, 65, 1008, 25];
```

25. Sort the nums array. Rather than being put in numeric order, the items are alphabetized:

```
nums.sort();
console.log(nums); // [1008, 105, 17, 2, 220, 25, 3, 45, 5, 65]
```

sort callback function

Sorting an array of numbers requires that a callback be passed to the sort method. Recall from a previous lesson that a callback is a function passed to another function as its argument.

- The callback takes two arguments of its own: a and b.
- The callback returns a - b.
- Under the hood, the method compares a to b, and if necessary, swaps their position.
- Working iteratively (again and again on a loop), this compare-and-swap procedure continues until there are no more swaps to be made, meaning that the items are sorted.

26. Call the sort method on the nums array, passing in the callback function. The result is properly numbers:

```
nums.sort(function(a,b) {
    return b - a; // b - a is descending
});
console.log(nums); // [2, 3, 5, 17, 25, 45, 65, 105, 220, 1008]
```

array.reverse()

The reverse method reverses the order of array items, so if they are first sorted, we get a reverse sort.

27. Reverse the fruits array, which is already sorted in alphabetical order:

```
fruits.reverse();
console.log(fruits); // ['papaya', 'lime', 'lemon', 'kiwi', 'grape',
'blueberry', 'banana','apricot']
```

The reverse() method can also be called on the numbers array once it is sorted, but it can also be sorted in reverse to begin with by returning **b - a** instead of **a - b**:

28. Push a few numbers into **nums**, so that it is no longer in ascending order, and then sort the array in descending (reverse) order:

```
nums.push(21, 325, 1234, 7, 88, 4);
console.log(nums); // [2, 3, 5, 17, 25, 45, 65, 105, 220, 1008, 21,
325, 1234, 7, 88, 4]
```

29. Sort the array in descending (reverse) order by flipping the return statement to **b - a**:

```
nums.sort(function(a,b) {
    return b - a; // b - a is descending
});
console.log(nums); // [1234, 1008, 325, 220, 105, 88, 65, 45, 25, 21,
17, 7, 5, 4, 3, 2]
```

2D (nested) arrays

An array can have arrays for its items. The terms to describe such an array include: **matrix, 2D array** and **nested array**

30. Make a 2x4 array of two items, each an array of four items:

```
const twoByFour = [[1,2,3,4], [5,6,7,8]];
console.log(twoByFour); // [1,2,3,4], [5,6,7,8]
```

```
console.log(twoByFour.length); // 2
console.log(twoByFour[0]); // [1,2,3,4]
console.log(twoByFour[1]); // [5,6,7,8]
```

To access individual number from this array of arrays, use double-square brackets.

31. Get the first and last items from each inner array:

```
console.log(nestedArr[0][0]); // 1
console.log(nestedArr[0][3]); // 4
console.log(nestedArr[1][0]); // 5
console.log(nestedArr[1][3]); // 8
```

32. Make a 3x3 tic-tac-toe "board", where all 9 squares start with a value of **null**:

```
let ticTacToe = [
    [null, null, null],
    [null, null, null],
    [null, null, null]
];
```

33. Game on! **X** chooses the middle, and **O** chooses the lower left square:

```
ticTacToe[1][1] = "X";
ticTacToe[2][0] = "O";
```

34. Log **ticTacToe** to see the game progress:

```
console.log(ticTacToe);
/*
(3) [Array(3), Array(3), Array(3)]
0: (3) [null, null, null]
1: (3) [null, 'X', null]
2: (3) ['O', null, null]
*/
```

04.01 Lab

1. Declare an array called **gems** and give it five items of your choice.
2. Log the entire array and also the first item.
3. Add 'sapphire' to the end of the array, using the index of the new item. If sapphire is already in the array, add another gem.
4. Add 'amethyst' to the end of the array using one of the array methods we have learned. If amethyst is already in the array, add some other gem.
5. Use an array method to put the gems in alphabetcial order.
6. Log the number of items of the array. The result should be 5.
7. Get the next-to-last gem, without deleting it from the array or hard-coding the index number. Save it to a variable and log it.
8. Use one array method to remove the last gem from the array, and then another array method to put the item right back in at the end. HINT: Save the removed gem to a new variable.
9. Given the 12 months: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec.

- Make a nested array called 'months', consisting of 4 items, each an array
- The child arrays have 3 items each, so Jan, Feb, Mar are in the first array. Using double-square brackets, log all months that start with 'J' and 'M'.

10. Given this list of numbers, 111, 23, 202, 12, 23, 1000, 2006, 1, 2, 14, 28, put them into an array called nums and then sort the array in descending order. HINT: You need to use an array method that takes a callback function.

04.01 Lab Solutions

1. 1. Declare an array called gems and give it three gems of your choice.

```
const gems = ['diamond', 'emerald', 'ruby'];
```

2. Log the entire array and also the first item.

```
console.log(gems); // ['diamond', 'emerald', 'ruby']
console.log(gems[0]); // diamond
```

3. Add 'sapphire' to the end of the array, using the index of the new item. If sapphire is already in the array, add another gem.

```
gems[3] = 'sapphire';
console.log(gems); // ['diamond', 'ruby', 'emerald', 'sapphire']
```

4. Using an array method, add 'amethyst' to the end of the array If amethyst is already in the array, add some other gem.

```
gems.push('amethyst');
console.log(gems); // ['diamond', 'ruby', 'emerald', 'sapphire',
'amethyst']
```

5. Use an array method to put the gems in alphabetcial order.

```
gems.sort();
console.log(gems); // ['amethyst', 'diamond', 'emerald', 'ruby',
'sapphire']
```

6. Log the number of items of the array. The result should be 5.

```
console.log(gems.length); // 5
```

7. Get the next-to-last gem, without deleting it from the array or hard-coding the index number. Save it to a variable and log it. HINT: length property

```

```

```
let nextToLastGem = gems[gems.length-2];
console.log(nextToLastGem); // ruby
```

8. Use one array method to remove the last gem from the array, and then another array method to put the item right back in at the end. HINT: Save the removed gem to a new variable.

```
let poppedGem = gems.pop();
console.log(poppedGem); // sapphire
console.log(gems); // ['amethyst', 'diamond', 'emerald', 'ruby']
gems.push(poppedGem);
console.log(gems); // ['amethyst', 'diamond', 'emerald', 'ruby',
'sapphire']
```

9. Given the 12 months: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec.

- Make a nested array called 'months', consisting of 4 items, each an array
- The child arrays have 3 items each, so Jan, Feb, Mar are in the first array.
- Using double-square brackets, log all months that start with 'J' and 'M'.

```
const months = [ ['Jan', 'Feb', 'Mar'],
                 ['Apr', 'May', 'Jun'],
                 ['Jul', 'Aug', 'Sep'],
                 ['Oct', 'Nov', 'Dec']
               ];
console.log(months[0][0]); // Jan
console.log(months[0][2]); // Mar
console.log(months[1][1]); // May
console.log(months[1][2]); // Jun
console.log(months[2][0]); // Jun
```

10. Given this list of numbers: **111, 23, 202, 12, 23, 1000, 2006, 1, 2, 14, 28**. Put them into an array called nums and then sort the array in descending order. HINT: You need to use an array method that takes a callback function.

```
const nums = [111, 23, 202, 12, 23, 1000, 2006, 1, 2, 14, 28];

nums.sort(function(a,b) {
    return b - a;
});

console.log(nums);
```

Lesson 04.02 - PROG

Objects

There are four categories of JS objects that interest us, three of which we are already quite familiar with:

- A. Built-in Objects: JS objects that do not pertain to the DOM.
    - Math object, with its methods: floor(), sqrt(), etc. The Math object does not need to be instantiated--just use it
    - Date object, with its methods: getHours(), getDate(), etc. The Date object must be instantiated: let dt = new Date()
- B. DOM objects: JS "versions" of HTML elements: body, div, image, button, select, input, h1, h2, p, etc.
    - DOM objects have properties that "belong" to the object.
    - HTML elements are brought into JS by various methods, viz: getElementById(), querySelector(), querySelectorAll()
    - Properties are accessed via "dot-syntax": pic.src
    - Properties exist as key-value pairs: pic.src = "cat.jpg" pic is the object, src is the key, "cat.jpg" is the value
    - Properties may be specific to particular DOM objects: image.src, checkbox.checked, input.value, select.selectedIndex
    - Other properties are applicable to practically any DOM object: image.id, checkbox.id, select.id, div.id, button.id, input.id
- C. Arrays are objects that store values in numeric order (index): fruits = ['apple', 'banana', 'cherry']
    - The first item is at index 0: fruits[0] // apple
    - Array items are not properties; they are "index-value" pairs
    - Arrays have many methods: push(), pop(), sort(), etc.

This fourth kind of object is what this lesson is all about:

- D. "User-Defined Objects" are made up by the programmer:
    - User-defined objects consist of properties in curly braces: const pet = {type: "Cat", name: "Fluffy", age: 3, cute: true}
    - As with DOM objects, properties are accessible via dot-syntax: console.log(pet.name); // Fluffy
    - In Web applications, datasets often exist as arrays of objects.

1. Open 04.02-Objects.html, and preview it in the browser.
2. Click the Get Car Info button to display car object property values.
3. Choose an animal from the menu; we get animal object property values.
4. In 04.02-Objects.html, switch from importing FINAL.js to PROG.js.
5. Switch to 04.02-Objects.js.

6. Declare a const called sportscar, and set it equal to a pair of curly braces.

```
const sportscar = {};
```

We use const to protect the data type from being mutated to, say, a string.

7. Try to change sportscar to a string. We get an error:

```
sportscar = "Porsche"; // ERROR: assignment to constant variable.
```

An object declared with const cannot be mutated, but its properties can be added, removed and changed as we see fit.

8. Declare another object, this time assigning it properties as key-value pairs. The key and its value are separated by a colon. Properties are separated one from the other by a comma. Values can be any data type: strings, numbers and booleans.

```
const car = {
    category: "sports car",
    make: 'Ford',
    model: 'Mustang GT',
    year: 1998,
    color: 'red',
    condition: 'excellent',
    miles: 123456,
    onRoad: true,
    forSale: false
};
```

9. Log the object and open it up in the Console to check the properties, which have been alphabetized:

```
console.log(car);
```

dot-syntax

Properties are available only to their object.vTo reference a property, use the dot-sytnax: **car.year** returns the value **1998**.

10. Get some car properties, which returns the values:

```
console.log(car.category); // sports car
console.log(`For Sale: ${car.year} ${car.make} ${car.model}`); // For
Sale: 1999 Ford Mustang GT
```

```
console.log(car.year); // 1998
console.log(car.miles); // 123456
console.log(car.onRoad); // true
```

setting property values

To set a value, get the property and set it to a new value: **car.year = 1999** changes the year to **1999**. Inside the curly braces, key-value pairs are separated by a colon, but when a value is changed later, an equal sign is used instead:

11. Set some car properties; this changes the value:

```
car.category = "muscle car";
car.condition = "very good";
car.miles = 123567;
car.year = 1999;
forSale = true;
```

12. Log the whole object to see the changed values:

```
console.log(car);
```

adding properties

Properties are added to an existing object the same way a property value is changed: object.key = value

13. Add three more properties to the car object: a string, a number and a boolean:

```
car.transmission = "manual";
car.doors = 2;
car.convertible = true;
console.log(car);
```

arrays and objects as properties

Properties can be of any data type, including arrays and even other objects.

14. Add two more properties: an array and an object:

```
car.options = ['sun roof', 'CD player', 'leather seats'];
car.mpg = {city: 18, hwy: 24};
console.log(car);
console.log(car.options); // ['sun roof', 'CD player', 'leather
seats']
console.log(car.mpg); // {city: 18, hwy: 24}
```

15. Access array items using square bracket syntax:

```
console.log(car.options); // ['sun roof', 'CD player', 'leather
seats']
console.log(car.options[2]); // leather seats
```

16. Access child objects using "dot-dot" syntax:

```
console.log(car.mpg.city); // 18
console.log(car.mpg.hwy); // 24
```

bundling related properties into one

17. Add three properties having to do with the engine:

```
car.cylinders = 8;
car.liters = 4.6;
car.horsepower = 260;
console.log(car);
```

It works, but since these properties all pertain to the engine, better might be to make an engine property and assign it an object having the three properties:

18. Bundle cylinders, liters and horsepower into an engine property. Since "engine" will be included in any reference to these properties, we can abbreviate the names without causing any confusion:

```
car.engine = {cyl: 8, ltr: 4.6, hp: 260};
```

object property with array as its value:

```
```js
car.options = ['sun roof', 'CD player', 'leather seats'];
console.log(car);
```
```

19. Access the engine properties using "dot-dot" syntax:

```
car(car.engine.cyl); // 8
car(car.engine.ltr); // 4.6
```

```
car(car.engine.hp); // 260
```

deleting properties

The delete keyword, preceding a property reference, removes that property. Now that car has an engine property, we can delete horsepower, cylinders and liters.

20. Delete the horsepower, cylinders and liters properties:

```
delete car.horsepower;
delete car.cylinders;
delete car.liters;
console.log(car);
```

square bracket syntax for keys

keys with spaces

Keys are essentially variable names, but unlike "regular" variables, keys can have spaces. This may be a good option for a two-word key that we prefer to keep readable as such, as opposed to using camelCase. The space means we cannot use dot-syntax. Instead, the keys go in quotes, inside square brackets:

21. Add a few properties of two words each:

```
car["top speed"] = 149;
car["consumer reviews"] = 234;
car["stars rating"] = 4.7;
console.log(car);
console.log(car["top speed"]);
```

22. Try to access "top speed" via dot syntax. With or without quotes, we get an error:

```
// car.top speed; // ERROR
// car."top speed"; // ERROR
```

dynamic variable keys

A key itself can be dynamic, that is a variable. For dynanmic keys, dot syntax won't work. Use square brackets, instead.

23. Declare a carKey variable, and set it equal to one of the keys, as a string:

```
let carKey = "model";
```

24. Using dot-syntax, try to get the car model as carKey. We get an error, because car has no carKey property:

```
console.log(car.carKey); // ERROR
```

25. To make car.carKey be understood as car.model, use square brackets:

```
console.log(car[carKey]); // Mustang GT
```

26. To set a multi-word key value, also use square brackets:

```
car["consumer reviews"] = 248;
console.log(car["consumer reviews"]); // 248
```

toLocaleString()

The toLocaleString() method is called on a number and returns the number with commas, converting it to a string in the process.

27. Get the six-digit mileage, and output it with a comma:

```
console.log(car.miles.toLocaleString()); // 123,567
```

object methods

- A method is a function that is scoped to -- belongs to -- an object
- A method is a property of an object, where the value is a function
- To make a method, set a key equal to an anonymous function
- Inside a method, all other properties are available via the 'this' keyword
- A method must return a value: call a method, gets the return value

28. Assign the car object a method, called listForSale. It returns a FOR SALE listing, concatenated from numerous properties referenced by 'this' syntax:

```
car.listForSale = function() {
return `FOR SALE! ${this.year} ${this.make} ${this.model},
only ${this.miles.toLocaleString('en-us')} miles, ${this.condition}
condition.
Loaded: ${this.options[0]}, ${this.options[1]}, ${this.options[2]},
much more.
MPG: ${this.mpg.hwy} highway, ${this.mpg.city} city. Best Offer!`;
}
```

29. Log a call to the method. We get the classified listing:

```
console.log(car.listForSale());
```

30. Declare a variable, set equal to a call to the method, and then log it:

```
let listing = car.listForSale();
console.log(listing);
```

outputting property values to the DOM

31. Get the elements where the car data is displayed:

```
const carTitle = document.getElementById('car-title');
const carMPG = document.getElementById('car-mpg');
const carEngine = document.getElementById('car-engine');
const carOptions = document.getElementById('car-options');
const carListing = document.getElementById('car-listing');
const carReviewsRating = document.getElementById('car-reviews-
ratings');
const carPic = document.getElementById('car-pic');
```

32. Get the button, and have it call the showCarInfo function when clicked:

```
const btn = document.querySelector('button');
btn.addEventListener("click", showCarInfo);
```

33. Define the showCarInfo function:

```
function showCarInfo() {

// 34. Hide the button, since it only gets clicked once:
this.style.display= "none";

// 35. Output the title as "year make model":
carTitle.textContent = `${car.year} ${car.make} ${car.model}`;

// 36. Output the properties of the mpg property:
carMPG.textContent = `MPG: ${car.mpg.hwy} hwy - ${car.mpg.city} city`;

// 37. Output the properties of the engine property:
carEngine.textContent = `Engine: V${car.engine.cyl} ${car.engine.ltr}
L ${car.engine.hp} hp - ${car.model}`;
```

```
// 38. Output the options, one array item at a time:
carOptions.textContent = `Options: ${car.options[0]},
${car.options[1]}, ${car.options[2]}`;

// 39. Output the two-word properties, using square bracket syntax:
carReviewsRating.textContent = `${car['consumer reviews']} consumer
reviews, ${car['star rating']} consumer reviews`;

// Output the FOR SALE listing, which is returned by the listForSale()
method:
carListing.textContent = `${car.listForSale()}`;

} // end function showCarInfo()
```

arrays of objects

In web applications, data is often stored as arrays of objects or objects with child objects for properties. For this animal menu part of the lesson, we will work with an extenal file, animals.js, which contains an array of animal objects.

40. Open animals.js. It contains two big variables, an object called animalsObj, and an array by the name of animalsArr.

- animalsObj consists of 20 properties:
    - Each key is an animal name, with 2-word names in quotes.
    - Each value is an object with 3 properties:
        - class (value is string)
        - herbivore (value is boolean)
        - continent (value is string)
- animalsArr has 20 items, each an object
    - The objects are not key-values
    - The animal name is a property--not a key
    - Each array item object has 4 properties:
        - name (value is string)
        - class (value is string)
        - herbivore (value is boolean)
        - continent (value is string)

We will use animalsObj for this exercise:

41. Get the DOM elements for the animals:

```
const animalsMenu = document.getElementById('animals-menu');
const animalName = document.getElementById('animal-name');
const animalClass = document.getElementById('animal-class');
const continent = document.getElementById('continent');
```

```
const herbivore = document.getElementById('herbivore');
const animalPic = document.getElementById('animal-pic');
```

42. Have the menu call a function on 'change' (menu choice):

```
animalsMenu.addEventListener('change', showAnimalInfo);
```

43. Define the function that runs when a menu choice is made:

```
function showAnimalInfo() {

    // 44. The menu value is an animal, such as "giraffe" or "panda". Save it to a variable, animal:
    let animal = animalsMenu.value; // e.g. giraffe, panda

    // 45. Look up the animal in the animals object, using the string variable as the dynamic key. For this, we use square brackets--not dot-syntax:
    let animalObj = animals[animal];

    // 46. Output the property values to their respective tags:
    animalName.textContent = animal;
    animalClass.textContent = 'Class:' + animalObj.class;
    continent.textContent = 'Continent:' + animalObj.continent;
    herbivore.textContent = 'Herbivore:' + animalObj.herbivore;

    // 47. Set the souce of the animal image:
    animalPic.src = `images/${animal}.jpg`;
}
```

CHALLENGE:

Make another function that produces the same DOM output. BUT rather than get data from the animals object, uses animalsArr, which is the array of objects.

```
```js
animalsMenu.addEventListener('change', showAnimalInfo2);

function showAnimalInfo2() {

    // 48. The menu animals are in the same order as the array animals, so get the index of the chosen animal:
    let indx = animalsMenu.selectedIndex;

    // 49. Look up the animal in the array. Subtract one from the index, since the second menu choice is the first animal:
```
```

```
    let animalObj = animalsArr[indx-1];
    console.log(animalObj); // {name: 'giraffe', class: 'mammal',
herbivore: true, continent: 'Africa'}

    // 50. Output the name of the animal:
    animalName.textContent = animalObj.name; // giraffe

    // 51. Output the class and continent:
    animalClass.textContent = 'Class: ' + animalObj.class; // mammal
    continent.textContent = 'Continent: ' + animalObj.continent;

    // 52. Output the animal image:
    animalPic.src = `images/${animalObj.name}.jpg`;

    // converting boolean to user-friendly output

    // For the Herbivore part, we don't want to literally output 'true' or
'false'. Better would be a more user-friendly "Yes" or "No". For this we
need conditional logic that sets a string to "Yes" or "No", depending on
the value of the boolean:

    // 53. Do a ternary that sets a variable to "Yes" or "No":
    let yesNo = animal.herbivore ? "Yes" : "No";

    // 54. Output Herbivore: Yes (for giraffe) or Herbivore: No (for lion:
    herbivore.textContent = 'Herbivore: ' + yesNo;

}
```

Lesson 04.03

Array Methods

Review: push(), pop(), sort(), reverse()

NEW: unshift(), shift(), concat(), splice(),

slice(), includes(), indexOf(), join(), flat()

Array methods are called on arrays and perform operations upon the array's items. We looked at several array methods in Lesson 04.01, namely: **push(), pop(), sort()** and **reverse()**. Let's recap these methods before moving on:

array redux: push(), pop(), sort(), reverse()

1. Declare an array with a few items, and use **push()** to add items to the end:

```
const fruits = ['blueberry', 'cherry', 'banana'];
fruits.push('kiwi');
fruits.push('orange', 'grape', 'lime');

console.log(fruits); // ['blueberry', 'cherry', 'banana', 'kiwi',
'orange', 'grape', 'lime']
```

2. Remove the last item using `pop()`. The method returns the popped item, so save it to a variable:

```
let popped = fruits.pop();
console.log(popped); // lime

console.log(fruits); // ['blueberry', 'cherry', 'banana', 'kiwi',
'orange', 'grape']
```

3. Arrange the items in alphabetical order with sort(). The method changes the existing array--it does not make a new one:

```
fruits.sort();

console.log(fruits); // ['banana', 'cherry', 'kiwi', 'orange']
```

4. Reverse the order of the array items. Since they are already sorted from A-Z, the result will be items from Z-A:

```
fruits.reverse();
console.log(fruits);  // ['orange', 'kiwi', 'grape', 'cherry',
'blueberry','banana']
```

Now, for some more array methods:

- **unshift()** : adds item to beginning
- **shift()** : removes item from beginning
- **concat()** : combines two or more arrays
- **splice()** : removes or swap out items
- **slice()** : makes a new array from a range of items
- **includes()** : looks for item and returns true or false
- **indexOf()** : returns index of first matching item
- **lastIndexOf()** : returns index of last matching item
- **join()** : turn an array into a string
- **flat()** : turn a matrix into a 1D array

unshift() and shift()

- **unshift()** adds an item to the beginning of an array.
- **shift()** removes the first item and returns it.

To help remember which is which:

- the longer word ("unshift") makes the array longer
- the shorter word ("shift") makes the array shorter
- "shift" reminds us that items added to the beginning of an array make the exising items shift to a higher index.

5. Use **unshift()** to add an item to the beginning of the fruits array:

```
fruits.unshift('strawberry');
console.log(fruits); // ['strawberry', 'orange', 'kiwi', 'grape',
'cherry', 'blueberry', 'banana']
```

6. Use **shift()** to remove and return the first item, saving that to a variable:

```
let shifted = fruits.shift();
console.log(shifted); // strawberry
console.log(fruits);
// ['orange', 'kiwi', 'grape', 'cherry', 'blueberry', 'banana']
```

concat()

- The **concat()** method concatentates (combines) two or more arrays into one. - Call **concat()** on one array, and pass the other array(s) in as argument(s).

7. Declare three arrays of fruits:

```
const tropicalFruits = ['mango', 'kiwi', 'banana', 'pineappple',
'papaya'];
const citrusFruits = ['grapefruit', 'lemon', 'lime', 'tangerine',
'orange'];
const blossomFruits = ['apple', 'peach', 'cherry', 'plum', 'pear'];
```

8. Use the **concat()** method to concatenate the three arrays into one:

```
const fruitCocktail = tropicalFruits.concat(citrusFruits,
blossomFruits);
console.log(fruitCocktailArr); // ['mango', 'kiwi', 'banana',
'pineappple', 'papaya', 'grapefruit', 'lemon', 'lime', 'tangerine',
'orange', 'apple', 'peach', 'cherry', 'plum', 'pear']
```

9. Put the fruit cocktail in alphabetical order:

```
fruitCocktail.sort();
console.log(fruitCocktail); // ['apple', 'banana', 'cherry',
'grapefruit', 'kiwi', 'lemon', 'lime', 'mango', 'orange', 'papaya',
'peach',  'pear', 'pineappple', 'plum', 'tangerine']
```

splice()

The **splice()** method removes (swaps out) items at a specified index or range of indices. Spliced out items may optionally be replaced with new items.

- **splice()** requires two arguments: the index of the first item to remove, and the number of items to remove.
- **splice()** returns the removed item(s). For multiple items, it returns an array.
- **splice()** optionally takes additional arguments, one for each replacement item.

10. Remove the item at index 7, which is 'mango':

```
fruitCocktail.splice(7,1);
console.log(fruitCocktail); // ['apple', 'banana', 'cherry',
'grapefruit', 'kiwi', 'lemon', 'lime', 'orange', 'papaya', 'peach',
'pear', 'pineappple', 'plum', 'tangerine']
```

11. Starting at index 2, splice out four consecutive items, saving the result to a new array:

```
let splicedItems = fruitCocktail.splice(2,4);
console.log(splicedItems; // ['cherry', 'grapefruit', 'kiwi', 'lemon']
console.log(fruitCocktail); // ['apple', 'banana', 'lime', 'mango',
'orange', 'papaya', 'peach',  'pear', 'pineappple', 'plum',
'tangerine']
```

To replace spliced item(s), pass in the new item(s) as additional argument(s).

12. Using splice(), swap 'lime' for 'lemon':

```
fruitCocktail.splice(2, 1, 'lemon');
console.log(fruitCocktail);
// ['apple', 'banana', 'lemon', 'mango', 'orange', 'papaya', 'peach',
'pear', 'pineappple', 'plum', 'tangerine']
```

To swap more than one item with `splice()`, just add more arguments.

13. Replace 'apple' and 'banana' with 'apricot' and 'blueberry':

```
fruitCocktail.splice(0, 2, 'apricot', 'blueberry');
console.log(fruitCocktail); // ['apricot', 'blueberry', 'lemon',
'mango', 'orange', 'papaya', 'peach',  'pear', 'pineappple', 'plum',
'tangerine']
```

adding item(s) at index

To add one or more items -- without removing any -- specify the index where you want the new item(s) to go, and then put 0 as the second argument of the splice() method. This means you are splicing out zero items.

14. Insert 'grape' and 'grapefruit' in its correct alphabetic position, that is, right before 'lemon':

```
fruitCocktail.splice(0, 2, 'grape', 'grapefruit');
console.log(fruitCocktail); // ['apricot', 'blueberry', 'grape',
'grapefruit', 'lemon', 'mango', 'orange', 'papaya', 'peach',  'pear',
'pineappple', 'plum', 'tangerine']
```

splicing items at random

If we choose items, again and again, at random, from an array, we will sooner or later get repeats. To avoid repeats, remove each item as it is chosen.

15. Pick three random fruits, one at a time, from fruitCocktail. We can get repeats:

```
let r = Math.floor(Math.random() * fruitCocktail.length);
console.log('random fruit 1:', fruitCocktail[r]); // random fruit

r = Math.floor(Math.random() * fruitCocktail.length);
console.log('random fruit 2:', fruitCocktail[r]); // random fruit

r = Math.floor(Math.random() * fruitCocktail.length);
console.log('random fruit 3:', fruitCocktail[r]); // random fruit
```

16. Check the console. and refresh it enough times to get a repeat.

17. Once again, pick three random fruits, one at a time, from fruitCocktail. Only this time, splice out the chosen fruit as you go. We get no repeats:

```
r = Math.floor(Math.random() * fruitCocktail.length);
console.log('spliced random fruit 1:', fruitCocktail[r]); // random
fruit
fruitCocktail.splice(r, 1);

r = Math.floor(Math.random() * fruitCocktail.length);
console.log('spliced random fruit 2:', fruitCocktail[r]); // random
fruit
fruitCocktail.splice(r, 1);

r = Math.floor(Math.random() * fruitCocktail.length);
console.log('spliced random fruit 3:', fruitCocktail[r]); // random
fruit
fruitCocktail.splice(r, 1);
```

18. Rerun the console.log until you get a repeat fruit. To rerun the console command, hit the Up Arrow and then hit Enter.

slice()

- **slice()** is called on an array and takes two arguments: a start and end index. These items get sliced, which means copied--not cut out.
- **slice()** returns a new array of sliced items, without removing the sliced items from the original array.
- **slice()** end index is exclusive, so NOT included in the new array.

19. Make a 2-fruit tropical smoothie from the 3rd and 4th fruits of the tropicalFruits array:

```
console.log("tropicalFruits:", tropicalFruits); // ['mango', 'kiwi',
'banana', 'pineappple', 'papaya']
const tropicalSmoothie = tropicalFruits.slice(2,4);
console.log("tropicalSmoothie:", tropicalSmoothie); // 'banana',
'pineappple']
```

If the second argument is omitted, it slices from the start index all the way to the end:

20. Take a slice of tropicalFruits from the 3rd item to the to last:

```
const slicedToEnd = tropicalFruits.slice(2);
console.log("3rd to last tropicalFruits:", slicedToEnd); // ['banana',
'pineappple', 'papaya']
```

includes()

The **includes()** method is called on an array and returns true if the argument is found in the array, and false if it is not:

21. Call the includes() method twice, so that we get one true and one false:

```
console.log("tropicalFruits includes 'kiwi':");
console.log(tropicalFruits.includes('kiwi')); // true
console.log("tropicalFruits includes 'plum':");
console.log(tropicalFruits.includes('plum')); // false
```

indexOf()

The **indexOf()** method is called on an array and returns the index of the first instance of the argument. If it is not found, it returns -1.

22. Call the indexOf() method twice, so that we get an index and a -1:

```
console.log("tropicalFruits index of 'banana':");
console.log(tropicalFruits.indexOf('banana')); // 2
console.log("tropicalFruits index of 'apple':");
console.log(tropicalFruits.indexOf('apple')); // -1
```

join()

- The join() method is called on an array and returns a string made from the array items.
- In no argument is passed in, the resulting string includes the array commas but no spaces.
- The join method can take an argument called the delimiter.
- The commas can be removed by passing in an empty string: join(" ").
- The join method does not change the array.

The join method can take a *delimiter* argument--character(s) that will appear between the items in the resulting string.

23. Use the join() method to make a string from fruitCocktail. For this first try, don't have any argument:

```
let fruitStr1 = fruitCocktail.join();
console.log("\nfruitStr1:\n", fruitStr1); //
lemon,orange,papaya,peach,pineappple,plum,tangerine
```

24. Try again, this time replacing the comma with a space:

```
let fruitStr2 = fruitCocktail.join(" ");
console.log("\nfruitStr2:\n", fruitStr2); // lemon orange papaya peach
pineappple plum tangerine
```

Given this array of strings, make a headline and a file name:

```
```js
const newsArr = ['Mets', 'Win', 'World', 'Series'];
```
```

25. Make the headline: 'Mets Win World Series'

```
let headline = newsArr.join(" ");
console.log("headline:", headline); // 'Mets Win World Series'
```

26. Make the file name. Use '-' as the delimiter to connect the words with hyphens. Also make everything lowercase and add '.jpg' at the end:

```
let filename = newsArr.join('-').toLowerCase() + '.jpg';
console.log('filename:', filename); // mets-win-world-series.jpg
```

flat()

The flat() method takes a nested, 2D array as its argument and returns a flat, one-dimensional array. It does not change the original array.

27. Declare a 3x3 / 2D / matrix array:

```
const ticTacToe = [
    ['X', 'O', null],
    [null, 'X', 'O'],
    ['O', null, 'X'] ];
console.log('ticTacToe:', ticTacToe);
```

28. Flatten the array:

```
const flatArr = ticTacToe.flat();
console.log('flatArr', flatArr);
// ['X', 'O', null, null, 'X', 'O', 'O', null, 'X']
```

use case for flat()

Some methods for finding values in an array only work if the array is flat.

29. Given a 3x3 array, flatten it and get the max value using Math.max() and the spread operator...:

```
let nums3x3 = [ [5,6,7], [15,16,17], [2,9,8] ];
console.log('nums3x3: ', nums3x3);
let flatNums = nums3x3.flat();
console.log('flatNums: ', flatNums);
let maxValue = Math.max(...flatNums);
console.log('maxValue: ', maxValue); // maxValue: 17
```

30. Given this nested array of animals, check if there ia a panda in it:

```
let animals = [ ['giraffe', 'zebra', 'elephant'], ['panda', 'koala',
'kangaroo'] ];
console.log('animals includes panda:', animals.includes('panda')); //
false
console.log('index of panda:', animals.indexOf('panda')); // -1
```

We get false for includes() and -1 for indexOf(), because these methods cannot see the nested items. Therefore, we first need to flatten the array.

31. Flatten the animals array, and try to find 'panda' again:

```
let animalsFlat = animals.flat();
console.log('flat animals includes panda:',
animalsFlat.includes('panda')); // true
console.log('flat index of panda:', animalsFlat.indexOf('panda')); //
3
```

This time it successfully finds the panda, because the array it is searching is flat.

Lab 04.04

Type your code directly in **04.03-Lab.html**, directly in the **script** tags:

1. Make a pets array, containing 'cat', 'dog' and 'bunny'.
2. Use an array method to add 'iguana' and 'parrot' to the end of the array.
3. Use an array method to insert 'hamster' between 'dog' and 'bunny'.
4. Use an array method to make 'snake' the first pet.
5. Use an array method to put the pets in alphabetical order (A-Z).
6. Use array methods to move the next to last pet to the beginning of pets. The next to last pet happens to be 'parrot'. Don't remove any pets or write any pet names:
7. The dog and cat don't like being next to each other, but they both like the bunny, so put the bunny between them. Don't remove any pets or write any pet names:
8. Given pets2, combine both arrays into a new array called petShopArr, and then alphabetize petShopArr:
9. Make a string called petsShopStr, consisting of all the pets in petShopArr, with the pets separated by ' & ':
10. Given this nested array, use two array methods to get the index of the dog. HINT: You need to make a new array.

```
const nestedPets = [
    ['bunny', 'parrot', 'canary'],
    ['cat', 'dog', 'gerbil'],
    ['goldfish', 'hamster', 'iguana'],
    ['snake', 'tarantula', 'turtle']
];
```

Lab 04.04 - SOLUTION

Solution code in **04.03-Lab-Solution.html**, in the **script** tags:

1. Make a pets array, containing 'cat', 'dog' and 'bunny'.

```
const pets = ['cat', 'dog', 'bunny'];

console.log(pets); // ['cat', 'dog', 'bunny']
```

2. Use an array method to add 'iguana' and 'parrot' to the end of the array.

```
pets.push('iguana', 'parrot');

console.log(pets);
// ['cat', 'dog', 'bunny', 'iguana', 'parrot']
```

3. Use an array method to insert 'hamster' between 'dog' and 'bunny'.

```
pets.splice(2, 0, 'iguana');

console.log(pets);
// ['cat', 'dog', 'hamster', 'bunny', 'iguana', 'parrot']
```

4. Use an array method to make 'snake' the first pet.

```
pets.unshift('snake');

console.log(pets);
// ['snake', 'cat', 'dog', 'hamster', 'bunny', 'iguana', 'parrot']
```

5. Use an array method to put the pets in alphabetical order (A-Z).

```
pets.sort();

console.log(pets);
// ['bunny', 'cat', 'dog', 'hamster', 'iguana', 'parrot', 'snake']
```

6. Use array methods to move the next to last pet to the beginning of pets. The next to last pet happens to be 'parrot'. Don't remove any pets or write any pet names:

```
let lastIndex = pets.length - 1;
let nextToLastItem = pets.splice(lastIndex - 1, 1);
pets.unshift(nextToLastItem);

console.log(pets);
// ['parrot', 'bunny', 'cat', 'dog', 'hamster', 'iguana', 'snake']
```

7. The dog and cat don't like being next to each other, but they both like the bunny, so put the bunny between them. Don't remove any pets or write any pet names:

```
let secondPet = pets.splice(1,1);
pets.splice(2, 0, secondPet);

console.log(pets);
// ['parrot', 'cat', 'bunny', 'dog', 'hamster', 'iguana', 'snake']
```

8. Given pets2, combine both arrays into a new array called petShopArr, and then alphabetize petShopArr:

```
const pets2 = ['canary', 'turtle', 'gerbil', 'tarantula', 'goldfish'];
const petShopArr = pets.concat(pets2);
petShopArr.sort();

console.log('petShopArr:', petShopArr);
// petShop: ['bunny', 'parrot', 'canary', 'cat', 'dog', 'gerbil',
'goldfish', 'hamster', 'iguana', 'snake', 'tarantula', 'turtle']
```

9. Make a string called petsShopStr, consisting of all the pets in petShopArr, with the pets separated by ' & ':

```
let petsShopStr = petShopArr.join(' & ');

console.log(petsShopStr);
// canary & cat & dog & gerbil .. etc.
```

10. Given this nested array, two array methods to get the index of the dog. HINT: You need to make a new array.

```
const nestedPets = [
    ['bunny', 'parrot', 'canary'],
    ['cat', 'dog', 'gerbil'],
    ['goldfish', 'hamster', 'iguana'],
    ['snake', 'tarantula', 'turtle']
];
```

```
const flatPets = nestedPets.flat();
let dogIndex = flatPets.indexOf('dog');
consolelog("index of dog:", dogIndex);
```

Lesson 04.04 - PROG

string methods

includes(), indexOf(), lastIndexOf(), toUpperCase(), toLowerCase()

slice(), replace(), replaceAll(), charAt(), split()

String are kind of like arrays of characters; both structures have:

- index position, with the first item (or character) at index 0
- length property, which returns the number of items (or characters)
- includes(X) method, which checks if X exists in the array or string
- slice(A,B) method, which copies from index A to B (not including B)
- indexOf(X) method, which gets the index of the first instance of X
- lastIndexOf(X) method gets the index of the last instance of X

Naturally, there are also many differences between strings and arrays:

- strings can only store one value at a time (pet = 'cat'), whereas arrays can store many values at a time (pets = ['cat', 'dog'])
- const strings cannot be changed, whereas the items of const arrays can be changed; the const part being "once an array, always an array"
- strings have methods not found in arrays, and vice-versa

string[index]

1. Declare a string, and then get the first and fourth characters:

```
let car = 'Corvette C3';
console.log(car[0]); // C
console.log(car[3]); // v
```

string.length

2. Get the number of characters in car; spaces count:

```
console.log(car.length); // 11
```

3. Get the last character of car, the same way you would get the last item of an array: length-1

```
console.log(car[car.length-1]); // 3
```

4. Check if this 'Vette is a "C2" or "C3" model:

```
console.log(car.includes('C2')); // false
console.log(car.includes('C3')); // true
```

5. Get the index of the first and last "C":

```
console.log(car.indexOf('C')); // 0
console.log(car.lastIndexOf('C')); // 9
```

slice(start_index, end_index)

Copy the make ("Corvette") and model ("C3") to their own variables. We will use the slice method for this.

- slice takes a start and end index as its arguments
- slice has negative indices: -1 is the last character, -2 is next to last
- slice copies and returns a substring from the start to end range
- the end index is exclusive, that is, not included in the substring
- the slice operation does not modify the original string in any way

6. To get the make, "Corvette", slice from index 0 to the index of the space:

```
let spaceIndex = car.indexOf(" ");
let make = car.slice(0, spaceIndex);
console.log(make); // Corvette
```

7. To get the model, "C3", slice from the next index after the space to the end. We are slicing to the end, so omit the second argument (the end index):

```
let model = car.slice(spaceIndex+1);
console.log(model); // C3
```

8. If we are sure that the model is two characters, we can slice backwards:

```
model = car.slice(-2);
console.log(model); // C3
```

Given this string promoting the car:

```
```js
let carPromo = "The 2023 Chevrolet Corvette furthers its reputation as a
high-value everyday supercar. The Corvette provides blistering
acceleration, phenomenal handling, a comfortable and well-trimmed cabin
```
```

```
and usable cargo space. Chevrolet reintroduces the high-performance
Corvette Z06 for the 2023 model year, features a new V8 engine with 670
horsepower. Chevrolet also adds a special appearance package that
celebrates the Corvette's 70th anniversary.";
```

9. Get the first half of the promo, so from index 0 to 1/2 the length:

```
let firstHalf = carPromo.slice(0, carPromo.length/2);
console.log('firstHalf:', firstHalf);
```

10. Sample a middle slice. Starting some distance before the middle index, and end an equal distance past the half-way point. The math is easier to read if we do this in steps, assigning each number to a variable:

```
let halfLen = carPromo.length / 2;
let distance = 18;
let middleText = carPromo.slice(halfLen - distance, halfLen +
distance);
console.log(middleText); // usable cargo space. Chevrolet reintroduces
```

getting the second sentence of the passage

Let's get the second sentence. This ivolves a little fancy slicing:

11. Get the start index of the second sentence. This is the index of the first period, plus one:

```
let sentence2StartIndex = carPromo.indexOf('.') + 1;
console.log('sentence2StartIndex:', sentence2StartIndex);
// indexOf() method for getting the second instance of a character
```

indexOf('.') gets the index of the first '.', but we want the second period. Pass in startIndex as a second argument. This tells it to start looking for the '.from the startIndex, not from the beginning of the string.

12. Get the index of the second sentence's period.

```
let period2Index = carPromo.indexOf('.', sentence2StartIndex);
console.log('period2Index:', period2Index);
```

13. Get the second sentence slice. Add one to the end to include the period:

```
let sentence2 = carPromo.slice(period2Index+1);
console.log('sentence2:', sentence2); // The Corvette provides
blistering acceleration, etc.
```

getting the last sentence of the passage

We need the index of the next-to-last period, as right after this is the starting point of the last sentence. Now, lastIndexOf('.') returns the index of the last period, so we want to tell the method to sample a carPromo that does not end in a period.

14. Copy carPromo to itself, minus the last character, that last '.':

```
carPromo = carPromo.slice(0,-1);
// The last period now belongs to the next-to-last sentence.
```

15. Get the index of the last period:

```
let lastPeriod = carPromo.lastIndexOf('.');
console.log('lastPeriod:', lastPeriod);
```

16. The last sentence starts two characters after the last period, so slice from there to the end. Since we are going to the end, there is no second argument, but we do need to put back the period at the end:

```
let lastSentence = carPromo.slice(lastPeriod + 2) + '.';
console.log('lastSentence:', lastSentence);
```

17. Take a slice of the middle 20 characters. The index of the mid-point is one-half the string length. So, start 10 characters before the mid-point and end 10 charcters after the mid-point:

```
let mid20Chars = carPromo2.slice(carPromo.length/2-10,
carPromo.length/2+10);
console.log('mid20Chars:', mid20Chars); // Chevy reintroduces
```

Now that we've seen how arrays and strings both have index, length, includes(), indexOf() and lastIndexOf(), let's get into the methods that are unique to strings.

toUpperCase(), toLowerCase()

- toUpperCase() returns an uppercase version of the string
- toLowerCase() returns a lowercase version of the string
- the original string is unchanged

18. Make uppercase and lowercase versions of car:

```
let carUC = car.toUpperCase();
console.log(carUC); // CORVETTE C3
let carLC = car.toLowerCase();
console.log(carLC); // corvette c3
```

replace(a,b), replaceAll(a,b)

- replace() is called on a string and takes two arguments: a substring to replace and its replacement.
- replace() operates on the first instance of the target substring.
- replaceAll() works like replace() but hits all instances of the target

19. Replace "Corv" with "'V". This gives us the car's nickname:

```
let nickname = make.replace("Corv", "'V");
console.log(nickname); // 'Vette
```

20. Use replaceAll() to globally change "Covette" to "'Vette". The method returns a new string, so save it to a variable, either itself or a new var:

```
carPromo = carPromo.replaceAll("Corvette", "'Vette");
console.log(carPromo); // 'Vette has replaced Corvette
```

Saving the string to itself vs. to a new variable:

- To modify the original string, save it to itself.
- To also keep the original string, save it to a new var, such as carPromo2.

21. Use replaceAll() again to globally change "Chevrolet" to "Chevy". Save carPromo to itself, so that both replacements accumulate in the same string:

```
carPromo = carPromo.replaceAll("Chevrolet", "Chevy");
console.log(carPromo); // 'Vette & Chevy have replaced Corvette &
Chevrolet
```

22. On second thought, we should have one full "Chevrolet Corvette" before reverting to nicknames. Use replace() to restore the first instance of each:

```
carPromo = carPromo.replace("'Vette'", "Corvette");
carPromo = carPromo.replace("Chevy", "Chevrolet");
```

charAt()

The charAt() method is called on a string:

- charAt(index) takes an index and returns the character found there.
- if the index argument exceeds the max index, it returns undefined.
- charAt() is another way of getting characters by index with []

23. Get the 100th character of the car promo by index and with charAt():

```
console.log(carPromo[99]); // v
console.log(carPromo.charAt(99)); // v
```

split()

- The split() method is called on a string and returns an array.
- It takes an argument (the delimiter) that specifies how the split is done.
- the split() method is often used in conjunction with join() to split a string into an array, perform some operations, and then join the array back into a string.

24. Declare a string (a movie quote about a Corvette C3):

```
let movieQuote = 'Start down low with a 350 cube, three and a quarter
horsepower, 4-speed, 4:10 gears, ten coats of competition orange,
hand-rubbed lacquer with a dual-plane manifold';
```

25. Call the split() method on the string, saving the result to an array:

```
const movieQuoteArr = movieQuote.split();
console.log(movieQuoteArr); // ['Start down low with a 350 cube, three
and a quarter horsepower, etc.]
```

Notice there's only one item in the array--the entire string.

26. Make each word an array item by passing in a space as delimiter:

```
const quoteWords = movieQuote.split(" ");
console.log(quoteWords); // ['Start', 'down', 'low', 'with' 'a',
'350', etc.]
```

27. Make each individual letter an array item with an empty string delimiter:

```
const quoteLetters = movieQuote.split("");
console.log(quoteLetters); // ['S', 't', 'a', 'r', 't', ' ', 'd', 'o',
etc.]
```

28. Pass in a comma-space (", ") as the delimiter. This splits the string every few words, into phrases:

```
const quotePhrases = movieQuote.split(", ");
console.log(quotePhrases); // ['Start down low with a 350 cube',
'three and a quarter horsepower', etc.]
```

That's nice, but that last phrase should be two. Let's pop the last item, make two strings from it, and then push the two strings back into the array.

29. Pop the last item, and save it:

```
let lastPart = quotePhrases.pop();
```

30. Get part A by slicing from the beginning to "with":

```
let partA = lastPart.slice(0, lastPart.indexOf("with"));
console.log(partA); // hand-rubbed lacquer
```

31. Get part B by starting at indexOf("with") and slicing to the end. Add "with a".length, because we want to skip "with a ", which has a length of 6:

```
let partB = lastPart.slice(lastPart.indexOf("with") + "with
a".length+1);
console.log(partB); // dual-plane manifold
```

32. Push the two parts into the quotePhrases array:

```
quotePhrases.push(partA, partB);
console.log(quotePhrases); // ['Start down low with a 350 cube', 'three
and a quarter horsepower', '4-speed', '4:10 gears', 'ten coats of
competition orange', 'hand-rubbed lacquer', 'dual-plane manifold']
```

04.04 Lab

1. Write a function that takes in a word and returns a message indicating whether the word starts with a vowel or a consonant. The function should also capitalize the first letter of the inputted word:

- if input is "apple", output is 'Apple starts with "A", which is a vowel.'
- if input is "cat", output is 'Cat starts with "C", which is a consonant.'

```
function vowelOrConsonant(word) {
    // A. replace the first letter with the same letter to uppercase
    // B. concatenate most of the message
    // C. check if the set of vowels contains the first letter, to
lowercase
        // D. finish concatenation based on the true-false from
includes()
}
```

E. try out the function

```
console.log(vowelOrConsonant("elephant"));
// Elephant starts with "E" which is a vowel.
console.log(vowelOrConsonant("giraffe"));
// Giraffe starts with "G" which is a consonant.
```

2. Make a news headline from this file name. Expected result: **Mets Lead Off Game By Hitting Three Straight Homers**

```
let fileName = "Mets-Lead-Off-Game-By-Hitting-Three-Straight-
Homers.html";

// A. replace all hyphens with spaces
let headline;
console.log(headline); // Mets Lead Off Game By Hitting Three Straight
Homers.html
// get rid of the file extension without assuming ".html" (could be
".jpg")
// B. find the index of the last dot, which is where the file
extension starts
let lastDotIndex;
// C. set the string equal to itself up to, but not including, the dot
console.log(headline); // Mets Lead Off Game By Hitting Three Straight
Homers
```

3. Write a function that takes in a singular noun and returns the plural of the word, according to these rules:

- if the word ends in "y", drop the "y" and add "ies" // examples: "city" ==> "cities" "cherry" ==> "cherries"
- if the word ends in "o" or "h" add "es" // examples: "mango" ==> "mangoes" "echo" ==> "echoes"
- if the word is in the animals array, the singular and plural are the same
- test the function on the seven words provided, below.

```
const animals = ["deer", "fish", "moose", "sheep", "swine", "bison",
"cod", "salmon", "shrimp", "trout"];

function makePlural(word) {
    // A. get the last letter of the inputted word
    let lastChar;
    // B. check if the word is in the animals array
        // C. if it is, so return the word, unchanged
    // D. check if the word ends in "h" or "o"
        // E. if it does, form the plural by adding "es"
        // F. check if the word ends in "y"
        // G. if it does, concatenate "ies" onto all but the last "y",
which we get as slice(0,-1)
        // H. if none of the above are true, just add "s"
}

// animals array words are the same in singular and plural
console.log(makePlural("moose")); // moose
// word ends in "o", so add "es" to form plural:
console.log(makePlural("mango")); // mangoes
// word ends in "h", so add "es":
console.log(makePlural("peach")); // peaches
// word ends in "y", so drop "y", so add "ies":
console.log(makePlural("cherry")); // cherries
// word ends in "y"--but that's not the only "y"
console.log(makePlural("boysenberry")); // boysenberries
console.log(makePlural("cherry")); // cherries
// none of the above, so just add "s" to form the plural:
console.log(makePlural("apple")); // apples
```

04.04 Lab

1. Write a function that takes in a word and returns a message indicating whether the word starts with a vowel or a consonant. The function should also capitalize the first letter of the inputted word:

- if input is "apple", output is 'Apple starts with "A", which is a vowel.'
- if input is "cat", output is 'Cat starts with "C", which is a consonant.'

```
function vowelOrConsonant(word) {
    // A. replace the first letter with the same letter to uppercase
    word = word.replace(word[0], word[0].toUpperCase());
    // B. concatenate most of the message
    let msg = `${word} starts with "${word[0].toUpperCase()}", which
is a `;
    // C. check if the set of vowels contains the first letter, to
lowercase
    if("aeiou".includes(word[0].toLowerCase())) {
        // D. finish concatenation based on the true-false from
includes()
        return msg += "vowel.";
    } else {
        return msg += "consonant.";
    }
}
```

E. try out the function

```
console.log(vowelOrConsonant("elephant"));
// Elephant starts with "E" which is a vowel.
console.log(vowelOrConsonant("giraffe"));
// Giraffe starts with "G" which is a consonant.
```

2. Make a news headline from this file name. Expected result: **Mets Lead Off Game By Hitting Three Straight Homers**

```
let fileName = "Mets-Lead-Off-Game-By-Hitting-Three-Straight-
Homers.html";

// A. replace all hyphens with spaces
let headline = fileName.replaceAll("-", " ");
console.log(headline); // Mets Lead Off Game By Hitting Three Straight
Homers.html

// get rid of the file extension without assuming ".html" (could be
".jpg")
```

```
// B. find the index of the last dot, which is where the file
extension starts
let lastDotIndex = headline.lastIndexOf(".");

// C. set the string equal to itself up to, but not including, the dot
headline = headline.slice(0, lastDotIndex);
console.log(headline); // Mets Lead Off Game By Hitting Three Straight
Homers
```

3. Write a function that takes in a singular noun and returns the plural of the word, according to these rules:

- if the word ends in "y", drop the "y" and add "ies" // examples: "city" ==> "cities" "cherry" ==> "cherries"
- if the word ends in "o" or "h" add "es" // examples: "mango" ==> "mangoes" "echo" ==> "echoes"
- if the word is in the animals array, the singular and plural are the same
- test the function on the seven words provided, below.

```
const animals = ["deer", "fish", "moose", "sheep", "swine", "bison",
"cod", "salmon", "shrimp", "trout"];

function makePlural(word) {
    // A. get the last letter of the inputted word
    let lastChar = word[word.length-1]
    // B. check if the word is in the animals array
    if(animals.includes(word)) {
        // C. if it is, so return the word, unchanged
        return word;
    // D. check if the word ends in "h" or "o"
    } else if(lastChar == "h" || lastChar == "o") {
        // E. if it does, form the plural by adding "es"
        return word + "es";
    // F. check if the word ends in "y"
    } else if(lastChar == "y") {
        // G. if it does, concatenate "ies" onto all but the last "y",
which we get as slice(0,-1)
        return word = word.slice(0,-1) + "ies";
    // H. if none of the above are true, just add "s"
    } else {
        return word + "s";
    }
}

// animals array words are the same in singular and plural
console.log(makePlural("moose")); // moose
// word ends in "o", so add "es" to form plural:
console.log(makePlural("mango")); // mangoes
// word ends in "h", so add "es":
console.log(makePlural("peach")); // peaches
```

```
// word ends in "y", so drop "y", so add "ies":
console.log(makePlural("cherry")); // cherries
// word ends in "y"--but that's not the only "y"
console.log(makePlural("boysenberry")); // boysenberries
console.log(makePlural("cherry")); // cherries
// none of the above, so just add "s" to form the plural:
console.log(makePlural("apple")); // apples
```

Lesson 05.01

Loops

Interating (Looping) Arrays

A loop executes a block of code, repeatedly, for as long as a condition remains true. There are a few kinds of loops, including "for loops" and "while loops". In this lesson, we will focus on "for loops".

for loops

A for loop considers three pieces of information in parentheses to decide if it will run the code inside its curly braces: a **counter**, a **condition** and an **incrementer**:

```
```js
for (counter, condition, incrementer) {
    // do stuff
}
```
```

counter

The counter is a number variable that goes up or down each time the loop is run. The counter is usually **i**, but can be any name. The counter is assigned an initial value, typically 0, although that can be any number.

condition

The condition compares the counter to another value. If the condition is true, the code inside the curly braces will run. If the condition is false, the loop ends.

incrementer

The incrementer (or decrementer), specifies the value by which the counter increases or decreases with each iteration of the loop. Typically, a counter will go up by one each time though the loop, as assigned by the shorthand, **i++**.

1. Write a for loop with a counter, **i**, that starts at 0, goes up by one each time, and stops when **i** gets to 10:

```js
for (let i = 0; i < 10; i++) {
    console.log(i); // 0, 1, 2...9
}
```

let variables are block scoped

As discussed in previous lessons, a `let` variable is `block scoped`, meaning that it is available only inside the code block in which it is declared.

2. Try to access `i` outside the loop. We get an error:

```
for (let i = 0; i < 10; i++) {
    console.log(i); // 0, 1, 2...9
}

console.log(i); // ERROR: i is not defined
```

The error indicates that **i** does not exist in the global scope, which is fine because **i** was only needed in the loop. Once the loop ended, **i** was deleted from memory ("garbage collected", as they say).

var variables are not block scoped

A **var** is not block scoped, so if you use **var i** for a loop counter, **i** will be instantiated in the global scope--and will still be there after the loop ends. It is said that **var** counters "leak out" of their loop. We don't want variables persisting in memory with nothing to do, so use **let** instead.

3. Do another loop with **var** instead of **let**, and verify that **i** still exists after the loop has ended:

```
for (var i = 0; i < 10; i++) {
    console.log(i); // 0, 1, 2...9
}

console.log('after loop', i); // after loop 10
```

Notice that the value of **i** after the loop ends is 10, which matches the loop condition. Once the counter reached 10, the condition **i < 10** became false, so the loop ended.

4. Change the condition to **i <= 10** to get to 10 inside the loop and 11 outside the loop.

Let's try counting down. Start **i** at 10 and *decrement* by 1 each time with **i--**. When **i** reaches 0, the condition **i > 0** is false, so the loop ends.

5. Write a loop with a counter that decrements (counts backwards):

```
for (let i = 10; i > 0; i--) {
    console.log(i) // runs 10 times
}

console.log(i) // 11
```

The 11 reminds us that we still have a lingering **var i** which "leaked out" of its loop into the global scope.

6. Change **i** to **j** and see that **j** ceases to exist once the loop ends:

```
for (let j = 10; j > 0; j--) {
    console.log(j) // runs 10 times
}
console.log(j) // ERROR: j is not defined
```

infinite loop

An infinite loop is one that never ends, because the condition is always true. The condition is supposed to eventually flip from true to false, but in an infinite loop that never happens due to a flaw in the logic.

This next loop runs forever because **i** starts at 10 and goes up from there. The condition **i > 0** is therefore always true.

7. Write but then comment out and do not run these infinite loops, as doing so may freeze your browser. Just study it before commenting it out.

```
/*
for(let i = 0; i < 10; i--) {
    console.log(i);
}

for (let i = 10; i > 0; i++) {
    console.log(i);
}
*/
```

the += and -= operators

The counter can be incremented / decremented by any value. To increment by 5, it's **i+=5**. To decrement by 2, it's **i-=2**.

8. Run a loop where the counter starts at 0, goes up by 5 each time until it reaches 100 (inclusive):

```
for (let i = 0; i <= 100; i+=5) {
    console.log(i); // 0, 5, 10...95, 100
}
```

Challenge: use for loops to produce the following output:

- 0, -3, -6, -9, -12
- 1900, 1920, 1940, 1960, 1980, 2000, 2020

continue

The **continue** keyword skips an iteration of a loop. It is used with conditional logic to specify when to skip:

9. Starting with 1900, output once each decade (1900, 1910, 1910, 1920, etc.) stopping at 2020, but skipgin 1960:

```
for(let i = 1900; i <= 2020; i=10) {
    if(i == 1960) {
        continue;
    }
    console.log(i); // 1900, 1920, 1940, 1980, 2000, 2020
}
```

iterating (looping) arrays

Loops are commonly used to iterate arrays, which means to go through them, item by time. The counter is used to get the current item by index.

In these next steps, we will loop through an array while also working in some review of array methods:

10. Declare an array:

```
const fruits = ['apple', 'blueberry', 'cherry', 'kiwi', 'lime',
'orange', 'plum'];
```

11. Push a few items into the end of the array:

```
fruits.push('apricot', 'papaya', 'grape');
```

12. Add a few items to the beginning of the array:

```
fruits.unshift('grapefruit', 'watermelon', 'tangerine');
```

13. Output the array and its length:

```
console.log(fruits, fruits.length); // ['grapefruit', 'watermelon',
'tangerine', 'apple', 'blueberry', 'cherry', 'kiwi', 'lime', 'orange',
'plum', 'apricot', 'papaya', 'grape'] 13
```

14. Starting with 'cherry', and assuming you don't know the index of any of the items, replace 'cherry' along with the next two items with 'lemon' and 'pear':

```
fruits.splice(fruits.indexOf('cherry'), 3, 'lemon', 'pear');
console.log(fruits, fruits.length); //  ['grapefruit', 'watermelon',
```

```
'tangerine', 'apple', 'blueberry', 'pear', 'lemon', 'orange', 'plum',
'apricot', 'papaya', 'grape'] 12
```

15. Without removing any items, add 'raspberry' and 'mango' right before 'apricot':

```
fruits.splice(fruits.indexOf('apricot'), 0, 'raspberry', 'mango');

console.log(fruits, fruits.length);
```

// ['grapefruit', 'watermelon', 'tangerine', 'apple', 'blueberry', 'pear', 'lemon', 'orange', 'plum', 'raspberry', 'mango', 'apricot', 'papaya', 'grape'] 14 ```

16. Iterate array with a for loop.

- Each time through the loop, make a jellybean.
- Number the output from 1-14

```
for(let i = 0; i < 14; i++) {
    let bean = `${i+1}. ${fruits[i]} jellybean`;
    console.log(bean);
}
// grapefruit jellybean
// .. etc. ..
// tangerine jellybean, etc.
```

17. Push in three more fruits; then sort and output the array:

```
fruits.push('banana', 'pineapple', 'peach');
console.log(fruits, fruits.length);
// ['apple', 'apricot', 'banana', 'blueberry'
// ..etc. .. 'plum', 'raspberry', 'tangerine', 'watermelon'] 17
```

18. Run the loop again:

```
for(let i = 0; i < 14; i++) {
   let bean = `${i+1}. ${fruits[i]} jellybean`;
    console.log(bean);
}
// apple jellybean
// ... etc. ...
// plum jellybean
```

The last three fruits ('raspberry', 'tangerine', 'watermelon') weren't outputted, because we are running the loop only 14 times for 17 array items.

array.length for loop condition

To make a loop dynamically respond to changes in the size of the array, don't hard-code the number of iterations. Use array.length, instead.

19. Change the loop condition so that it will always run as many times as there are items in the array. Dispense with the bean variable and the item numbering, and just log the jellybean directly:

```
    for(let i = 0; i < fruits.length; i++) {
    console.log(`${fruits[i]} jellybean`);
    /* apple jellybean
    ... etc. ...
    watermelon jellybean */
}
```

Now, we get all 17 jellybeans.

loops with conditional logic

Loops that iterate arrays often include conditional logic to evaluate the individual items. Let's give this a try.

Let's make jellybeans again, but this time only if the fruit has five or fewer characters. To check how many letters the fruit is, use the string.length property.

20. Make jellybeans, but only if the fruit is five letters or less:

```
for(let i = 0; i < fruits.length; i++) {
    if(fruits[i].length <= 5) {
        console.log(`${fruits[i]} jellybean`);
    }
} // jellybeans: 'apple', 'grape', 'lemon', 'mango', 'peach', 'pear',
'plum'
```

Challenge:

21. Make different fruit treats, depending on the number of letters

- if the fruit is 5 or fewer letters, make jellybeans
- if the fruit has 6-8 letters, make popsicles: "orange popsicle"
- if the fruit is 9 or more letters, make lollipops: "tangerine lollipop"

```
console.log("\nMake jellybeans, popsicles or lollipops:\n");
```

```
for(let i = 0; i < fruits.length; i++) {
    if(fruits[i].length <= 5) {
        console.log(`${fruits[i]} jellybean`);
    } else if(fruits[i].length <= 8) {
        console.log(`${fruits[i]} popsicle`);
    } else {
        console.log(`${fruits[i]} lollipop`);
    }
}
// apple jellybean
// apricot popsicle
// ... etc. ...
// watermelon lollipop
```

making a new array while looping

The loop is not saving the treats anywhere; they're just being dumped out onto the console.

22. Refactor the loop so that treats are saved to a new array, which we declared before the above the loop:

```
console.log("\nSave jellybeans, popsices and lollipops to the treats
array:\n");

const treats = [];
for(let i = 0; i < fruits.length; i++) {
    if(fruits[i].length <= 5) {
        treats.push(`${fruits[i]} jellybean`);
    } else if(fruits[i].length <= 8) {
        treats.push(`${fruits[i]} popsicle`);
    } else {
        treats.push(`${fruits[i]} lollipop`);
    }
}
console.log(treats); // ['apple jellybean', 'apricot popsicle' .. etc.
.. 'watermelon lollipop']
```

23. Push in three more berries and then sort the array:

```
fruits.push('strawberry', 'blackberry', 'boysenberry');
fruits.sort();
```

Challenge:

24. Refactor the fruit loop, so that we still make treats, based on the length of the word. But if the fruit is a 'berry', make jam instead of lollipops. Also this time, since we are using fruits[i] so often, simplify fruits[i] by saving it to a variable:

```
const treats2 = [];
for(let i = 0; i < fruits.length; i++) {
    let fruit = fruits[i];
    if(fruit.includes("berry")) {
        treats2.push(`${fruit} jam`);
    } else if(fruit.length < 6) {
        treats2.push(`${fruit} jellybean`);
    } else if(fruit.length < 9) {
        treats2.push(`${fruit} popsicle`);
    } else { // 9+ letters, but not a berry
        treats2.push(`${fruit} lollipop`);
    }
}
console.log('treats2:', treats2);
```

making arrays of objects with a loop

It can be very useful to make arrays of objects on a loop. As objects, the resulting array items can have as many properties as you like:

25. Make a loop that goes through this array of full names, and makes objects consisting of three properites each: firstName, lastName, pin, the last being a random 4-digit PIN, with leading 0's, as needed.

```
const basketballStarsArr = ["LeBron James", "Micheal Jordan", "Larry
Bird", "Julius Erving", "Wilt Chamberlain"];
// Push each object into this given array:
const basketballStarObjArr = [];

for(let i = 0; i < basketballStarsArr.length; i++) {

    // 26. Split the current array item, the full name, into an array
of two names
    let namesArr = basketballStarsArr[i].split(" ");

    // 27. Save the first name, which is the first array item, to a
variable
    let fName = namesArr[0];

    // 28. Save the last name, which is the second array item, to a
variable
    let lName = namesArr[1];

    // 29. Generate a random integer from 0-9999
    let pin = Math.floor(Math.random()*10000);

    // 30. Precede the number with leading 0's, as needed
    if(pin == 0) {
        pin = '0000';
    } else if(pin < 10) {
```

```
            pin = '000' + pin;
        } else if(pin < 100) {
            pin = '00' + pin;
        } else if(pin < 1000) {
            pin = '0' + pin;
        // it's already 4 digits, but stringify it
        } else {
            pin = '' + pin;
        }

        // 31. Make an object containing the three properties
        let obj = {
            firstName: fName,
            lastName: lName,
            pin: pin
        };

        // 32. Push the object into the array
        basketballStarObjArr.push(obj);

    }

    // 33. Log the newly made array of objects
    console.log(basketballStarObjArr);
```

26. Given this array of names, make passwords consisting or: the first name, backwards, all lowercase

- a special character: "#" if the first name has 4 letters or less "&" if the first name has exactly 5 letters "%" if the first name has more that 5 letters
- the first three letters of the last name
- a piece of punctuation: an exclamation point ("!") if the first name and last name have the same number of letters a question mark ("?") if the first name is longer than the last name an colon ("😊 if the last name is longer than the first name
- the number of letters in the full name
- if the first and last name start with the same letter: add an equal sign "=" to the end otherwise, add an asterisk "*"
- finally, if the last name starts with a vowel: add "V" for vowel otherwise add "C" for consonant--unless the last name starts with "Y", in which case, add "X"

Save the passwords to an array of objects that include the names, divided into first and last name

```
```js
const ballplayersArr = ["Hank Aaron", "Ernie Banks", "Carl Yastrzemski",
"Mickey Mantle", "Tony Oliva", "Babe Ruth", "Willie Mays", "Nolan Ryan"];

const ballplayersObjArr = [];

for(let i = 0; i < ballplayersArr.length; i++) {
```
```

```
    // 35. Split the first and last name into array
    let names = ballplayersArr[i].split(" ");

    // 36. Assign the first and last names to variables
    let fName = names[0];
    let lName = names[1];

    // 37. Split the first name string into an array
    let fNameCharsArr = fName.split("");

    // 38. Reverse the array
    let fNameCharsArrRev = fNameCharsArr.reverse();
    // make a backwards string from the reversed array
    let fNameBackwards = fNameCharsArrRev.join("");

    // 39. O, chain split(), reverse() and join() together:
    let fNameRev = fName.split("").reverse().join("");

    // 40 Start concatenating the password, beginning with the first name
backwards:
    let pswd = fNameRev;

    // 41. Add a special character, based on the length of the first name:
    // add "#", if the first name has 4 letters or less
    if(fName.lenght <= 4) {
        pswd += "#";
    // add "&", if the first name has exactly 5 letters
    } else if(fName.lenght == 5) {
        pswd += "&";
    } else { // first name has 5+ letters, so add "%""
        pswd += "%";
    }

    // 42. Add the first three letters of the last name:
    pswd += lName.slice(0,3);

    // 43. If first name and last name have the same number of letters,
add "!"
    if(fName.length == lName.length) {
        pswd += "!";
    // if first name is longer than last name, add "?"
    } else if(fName.length > lName.length) {
        pswd += "?";
    } else { // last name is longer than first; add ":"
        pswd += ":";
    }

    // 44. Check if the first and last names start with the same letter
    if(fName[0] == lName[0]) {
        pswd += "=";
    } else {
```

```
        pswd += "*";
    }

    // 45. Check if the first name starts with a vowel, consonant or "y"
    if("aeiou".includes(lName[0])) {
        pswd += "V";
    } else if(lName[0] == "y") {
        pswd += "X";
    } else {
        pswd += "C";
    }

    // 46. Make an object of 3 properties:
    let obj = {
        firstName: fName,
        lastName: lName,
        password: pswd
    };

    // 47. Add the object to the new array
    ballplayersObjArr.push(obj);
}

// 48. Output the new array
console.log(ballplayersObjArr);

// Expected output:
/* namesAndPasswords = [
    {firstName: "Hank", lastName: "Aaron", password: "knah#Aar:9*V"},
    {firstName: "Ernie", lastName: "Banks", password: "einre&Ban!10C="},
    {firstName: "Carl", lastName: "Yastrzemski", password: "lrac"},
    -- etc. --
]; */
```

Lab 05.01

1. Write a for loop that makes the following array: [3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
2. Write a for loop that makes the following array: [100, 80, 60, 40, 20, 0, -20, -40, -60, -80, -100]
3. Given this array of numbers, use a for loop to add up all the numbers. Save the total to a variable, sum.

```
const nums = [45, 54, 63, 72, 89, 91, 106];

// 4. Given this array of mixed numbers, 'number-like strings' and
fruits, find the sum of the numbers and 'number-like strings'. This
requires you to ignore the fruits and to convert the 'number-like
strings' to actual numbers. Hint: Think Falsey!
const mix = ["4", 5, "6", "apple", 7, "8", "kiwi", 9, 10, "plum"];

// if it's convertible into a number to do math with
// Number("apple") is NaN, falsey, which returns false
```

4. Given this empty array, numsObjArr, and starter loop, populate the array with objects, each having four properties:

- **num** : a number from 1-10
- **sq** : the square of the number
- **sqRt** : the square root of the number
- **even** : true if the numer is even, else false */ *const numsObjArr = []; /* {num: 2, sq: 4, sqRt: 1.4142135623730951, even: true} {num: 3, sq: 9, sqRt: 1.7320508075688772, even: false} {num: 4, square: 16, sqRt: 2, isEven: true} */

Lab 05.01 - Solution

1. Write a for loop that makes the following array: [3, 5, 7, 9, 11, 13, 15, 17, 19, 21]

```
let nums1 = [];

for(let i = 3; i <= 21; i+=2) {
    nums1.push(i);
}
console.log(nums1);
```

2. Write a for loop that makes the following array: [100, 80, 60, 40, 20, 0, -20, -40, -60, -80, -100]

```
let nums2 = [];
for(let i = 100; i >= -100; i-=20) {
    nums2.push(i);
}
console.log(nums2);
```

3. Given this array of numbers, use a for loop to add up all the numbers. Save the total to a variable, sum.

```
const nums = [45, 54, 63, 72, 89, 91, 106];
let sum = 0;

for(let i = 0; i < nums.length; i++) {
    sum += nums[i];
}
console.log('sum:', sum);
```

4. Given this array of mixed numbers, 'number-like strings' and fruits, find the sum of the numbers and 'number-like strings'. This requires you to ignore the fruits and to convert the 'number-like strings' to actual numbers. Hint: Think Falsey!

```
let tot = 0;

const mix = ["4", 5, "6", "apple", 7, "8", "kiwi", 9, 10, "plum"];

for(let i = 0; i < mix.length; i++) {
    // if it's convertible into a number to do math with
    let x = Number(mix[i]); // Number("apple") is NaN, falsey, which
returns false
    if(x) tot += x;
}
console.log('tot:', tot);
```

5. Given this empty array, numsObjArr, and starter loop, populate the array with objects, each having four properties:

- num : a number from 1-10
- sq : the square of the number
- sqRt : the square root of the number
- even : true if the numer is even, else false

```
const numsObjArr = [];

for(let i = 0; i < 10; i++) {
    let even = i % 2 == 0 ? true : false;
    let obj = {num: i, sq: i**2, sqRt: Math.sqrt(i), even: even};
    numsObjArr.push(obj);
}

console.log(numsObjArr);
/*
{num: 2, sq: 4, sqRt: 1.4142135623730951, even: true}
{num: 3, sq: 9, sqRt: 1.7320508075688772, even: false}
{num: 4, square: 16, sqRt: 2, isEven: true}
*/
```