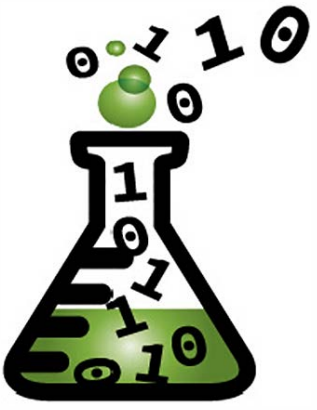
noble desktop

Noble Desktop
JavaScript Bootcamp



VOL. I

Lesson & Lab Workbook

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 "sees" web pages as a hierarchical collection of objects: 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...

- between a pair of 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. Open the 00-Introduction folder and from in there, 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.

4. 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.

```
alert('Hello World');
```

string

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

5. Inside the script tag, add an alert with the string: "Hello World"

```
```

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

6. Save the file and open it in Chrome. You should see a dialog that says *Hello World*.
7. 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 play with JS directly in the Console, as many developers do—no html page required. • To output content to the Console, use console.log(). In the Chrome browser, type Command-Option-I to open Developer Tools. Add this command:

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.

8. Use the console.log() method 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");
```

9. Save the file and reload the browser. You get the alert again, but there's more:
10. 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. Since the window object is top-level, it is assumed, and therefore can be omitted from the syntax. But we can also add them:

11. 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.

comment

Comments explain what the code is doing. Comments are for us humans and, as such, are ignored by JS when the program is run. Comments have no effect on how a program runs, but they do make a program easier to read and understand. We recommend you comment your code.

single-line comment

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

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

```
// alert() and console.log() output what's in parentheses
// window. is optional, since it is understood
window.alert('Hello World');
window.console.log("Hello from the Console");
```

in-line comment

If a comment is short enough, it can go on the same line as the code to which it refers.

13. Add an in-line comment:

```
// alert() and console.log() output what's in parentheses
window.alert('Hello World');
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.

14. We don't want to keep getting that pop-up, so comment-out the alert.

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

We could have just deleted the alert, but by commenting it out instead, we keep an example for study and review purposes.

multi-line comments

If a comment runs more than one line, you can wrap all in /* ... */.

15. Replace the single-line comment at the top of the script with a multi-line comment:

```
/*
2 ways to output JS:
1.) alert() for a pop-up in the browser
2.) console.log() for output to the console
*/
// window.alert('Hello World');
window.console.log("Hello from the Console");
```

---

anonymous function

A function without a name is called 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**.

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 avaialable 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;
```

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 decremneted 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.

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 and is called an *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 sscope, global memory

Global scope / memory refers to variables that are declared at the level of the *global object* and as such are avaialable 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();
```

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*.

index

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

```
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
}
```

methods

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:

```
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
```

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 conditon 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 Javascript Window object is the top-level "thing" in html: it is above the document (web page) itself.

Workbook Instructions

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 hmtl 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.

sumbitting lab work as HW

- 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

- 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**.
- 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

- Your big **PDF** file, **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 browsser.

---

var vs. let

string variables

number variables

typeof() method

boolean variables

undefined variables

---

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 tyes**. These "primitives" are the topic of this lesson.

variable data types

There are two major 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 only *one* value at a time. Primitives come in various **data types**:
    - **string** -- with value as text in quotes (e.g. "Hello World")
    - **number** -- with value as integer and decimal (e.g. 3, 3.5)
    - **boolean** -- with value of only **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, we 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**, altnough **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
```

One difference between **var** and **let** is that a **var** can be redeclared.

3. Redeclare **pet**:

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

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
```

To change the value of an existing variable, don't redeclare it; just set it equal to something else.

5. Comment out **let petSound = "Grrr!"**, and then set **petSound** to "Grrr!" *without redeclaring* the variable:

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

Multiple values can be outputted in the same **console.log**:

6. 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, setting them equal to 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
```

Naturally, numeric variables can be used in mathematical operations, the result of which may also be assigned to a variable.

9. 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 two booleans, each with a value of **true**:

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

Toggling or *flipping* a boolean can be done either by direct assignment or by putting an exclamation point (**!**) in front of it.

11. 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 four 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
```

Multiple variables, separated by commas, can be delared in *one* line of code.

14. Declare three variables with just one instance of the **let** keyword:

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

Such "one-liner" variable declarations are more common when the vars that are not to be assiged an initial value.

15. Declare four **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"
```

- **END Lesson 01.01**
- **NEXT: Lab 01.01**
- **Lesson 01.02**

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));
```

- **END Lab 01.01**
- **SEE Lab 01.01 Solution**

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));
```

END 01.01 LAB SOLUTION

Next: Lesson 01.02

"number-like strings"

NaN (Not a Number)

Number() method

null

not defined

string concatenation

"number-like strings"

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
```

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

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 the number 80, so attemptinh 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 get new 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 = '$' + food + bev + tip;
console.log(tot, typeof(tot)); // $25158 string
```

14. Remove the dollar sign, and run it again; this time the math works. *After* the math is done, put back the **$**:

```
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
In the above example, the number **tot** is changed into a string. Changing the datatype of a variable *is* permitted, but it should be done sparingly.

16. Change a three digit number into a four-digit PIN by concatenating a leading zero. This result is a string, but the reason for changing datatypes does make sense:

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

- **END Lesson 01.02**
- **NEXT: Lab 01.02**
- **Lesson 01.03**

Instructions:

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

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'
```

Use the Number() method, as needed, to get the expected values:

```
// 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***:

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

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

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

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

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

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

- **END Lab 01.02**
- **SEE Lab 01.02 Solution**

01.02 Lab Solution

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

On your own:

```
    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'
```

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***:

```
// 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);

// 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);

// 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);
```

- **END Lab Solution 01.02**

UNIT 01

LESSON 01.03

---

resoloving nested quotes

string interpolation

const

---

resolving conflicting nested quotes

Strings can go in either double or single quotes, but nested quotes can cause problems.

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 identified

An **unexpected identifier**, also known as an **unexpected token**, is a character that doesn't belong. The problem is, the apostrophe was mistaken for the end of the string, which makes 'Won' the entire string--and everything else '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:

```
    // let song2 = 'Don't Stop Believin''; // error
    let song2 = 'Don\'t Stop Believin\'';
    console.log(song2);
```

string interpolation

Long string concatenation can get hard to read, with all its quote and plus-signs. A cleaner alternative is **string interpolation**, which doesn't have any 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, preceded by a dollar sign: **${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 catGreeting = "Meow! 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 = `Meow! 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 = 'Shopping List:' +
        item1 +
        item2 +
        item3;
    console.log(shoppingList);
    // Shopping List:eggsbreadmilk
```

8. Try it with string interpolation. Now, hitting enter gives you a line break in the output. Tab works too:

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

The line breaks and tabs can be done with regular concatenation:

- **\n** in string concatenation gives a line break.
- **\t** in string concatenation gives a tabs.

9. Try the shopping list with regular string concatenation, including the line breaks and tabs. It can be done, but is a pain to write, so feel free to copy-paste this one:

```
    shoppingList = "Shopping List: \n\t" + item1 + "\n\t" + item2 + "\n\t"
+ 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* be changed.

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

10. Declare an uppercase variable with **let**:

```
    let MOON_DIAMETER = 2159.1;
    console.log(MOON_DIAMETER); // 2159.1
    MOON_DIAMETER *= 10; // multiply by 10
    console.log(MOON_DIAMETER); // 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 as a **var**:

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

14. It gives an error, so comment it out.

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

- **End Lesson 01.03**
- **Next: 01.03 Lab**
- **Lesson 01.04**

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!
*/
```

- **END Lab 01.03**
- **SEE Lab 01.03 Solution**

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! ***
*/
```

- **END Lab 01.03**
- **SEE Lab 01.03 Solution**

Math Object

math operators: +, -, *=, /, **, %

Math.random() .round() .floor() .ceil()

.max() .min() .abs() .pow() .PI

.toFixed()

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

Naturally, we can do math with numeric variables.

Mathematical operators include:

```
+   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
console.log(n1 % n2); // 1
```

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
- Operations inside () are done 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:

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(2.999);
console.log(y); // 2
```

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

7. Round up a number:

```
let z = Math.ceil(2.001);
console.log(z); // 3
```

getting a random number greater than 1

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
```

getting a random integer in a range

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

10. Round down 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
```

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
```

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

12. Find the minimum of a set of numbers:

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

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

13. 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, the ** operator does the same thing as Math.pow():

14. 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 just makes it positive:

15. Use **Math.abs()** to get the absolute value:

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

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

16. 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:

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

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

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

18. 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
```

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

```
console.log(pi2 + pi2); // 3.143.14
```

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

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

20. Try these math shorthand operators:

```
x = 20;

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

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

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

```
x *= 2; // multiply
console.log(x); // 420

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

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

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

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

- **END Lesson 01.04**
- **NEXT: Lab 01.04**
- **NEXT: Lesson 02.01**

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
*/
```

- **END Lab 01.04**
- **SEE Lab 01.04 Solution**

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 = π r². 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: a²+b²=c², 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

- **END Lab Solution 01.04**
- **NEXT Lesson 02.01**

UNIT 02

LESSON 02.01

---

logical operators

conditional logic: if-else

if - else if - else

variable scope: let vs var

A comparison operator *compares* two values and resolves to a boolean (**true** or **false**). The various operators are:

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

single vs. double vs. triple equal signs

- **=** assigns a value:
  - **x = 5**. Whatever **x** used to be, now it's 5.
- **==** *compares* two values; it does *not* assign a value.
  - returns **true** if the two values are equal
  - returns **false** if the two values are not equal.
  - does not consider datatype. **5 == "5"** is **true**.
- **===** compares two values; it does *not* assign a value.
  - returns **true** if they are equal in both value *and* datatype.
  - returns **false** if they are not equal in either value *or* datatype.
  - considers datatype: **5 === "5"** is **false**.
- **!=** checks for inequality of value, but not of datatype.
- **!==** checks for inequality of both value and data type.

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

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

2. Make some comparisons using the inequality operators:

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

3. Make some comparisons using the greater than less than equal to operators:

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

conditional logic with if else

An **if-statement** makes a comparison between two values, which resolves to a **boolen**. If the condition is **true**, the code inside the curly braces of the if statement will run. If the condition is **false**, the code will not run. There may be an "else part" that runs if the condition is false.

How to write an **if-statement()**:

- write *if* followed by a pair of parentheses: **if()**
- inside that put the condition to evaluate: **if(5 > 3)**
- after that, put a pair of curly braces:
  - **if(5 > 3) { }**
- inside the curly braces, write the code that you want to run if the condition is true:
  - **if(5 > 3) { console.log("It's true!"); }**

4. Do some comparisons inside if-statements:

```
if(7 == "7") {
    console.log('close enough');
}

if(7 === "7") {
    console.log('close but no cigar');
}
```

A boolean is already true or false, so we don't need to explicitly compare it true or false.

5. Do two versions of an if-statement with a boolean, with and without parentheses:

```
let raining = true;

if(raining == true) {
    console.log("It's raining, so bring an umbrella!"); // runs
}

if(raining) { // also works
    console.log("It's raining, so bring an umbrella!"); // runs
}
```

if else

What if we want go to the beach if it *not* raining. For that, we need an **else** part":

6. Make raining *false* and add an *else*:

```
raining = false;

if(raining) {
    console.log("It's raining, so bring an umbrella!");
} else {
    console.log("It's not raining! Let's go to the beach!"); // runs
}
```

Strings can also be evaluated with if-else logic.

7. Check if the weather is "sunny":

```
let weather = "sunny";

if(weather == "sunny") { // runs
    console.log("It's sunny! Don't forget your sunglasses!");
} else {
    console.log("It isn't sunny! Leave the sunglasses at home!");
}
```

With "number-like strings", the digits are evaluated like individual letters. "50" is greater than "100", because 5 is greater than 1.

8. Compare two "number-like strings":

```
let full = "100";
let half = "50";
```

```
if(full > half) {
    console.log('"100" > "50"');
} else {
    console.log('because 5 > 1'); // runs
}
```

9. Convert the "number-like strings" to actual numbers and try again:

```
full = Number("100");
half = Number("50");

if(full > half) {
    console.log('100 > 50'); // runs
} else {
    console.log('50 > 100');
}
```

if-else if-else logic

else if adds another condition to evaluate if **if** returns false:

10. Try this **else if**:

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

if (highScore > myScore) {
    console.log('You did not beat the high score!');
} else if (highScore < myScore) {
    console.log('You beat the high score!');
} else {
    console.log('You tied the high score!');
}
```

11. Try different values of **myScore** to get all three outcomes.

12. Check the weather with **else if** logic:

```
weather = "cloudy";

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

13. Try different values of **weather** to get all three outcomes.

There can be any number of **else if** blocks between the opening **if** and the closing **else**.

14. Run this code with its three **else if** clauses. The best approach is to go higher to lower by ranges (90+, 80-89, 70-79, 65-69, 64 and below):

```
let score = 77;
let grade = "";

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";
}
let reportCard = **Your score: ${score}.
Your grade: ${grade}.**
console.log(reportCard);
```

block scope

As you recall from a previous lesson:

let is **block-scoped**, meaning it is available only inside the the curly braces in which it was declared. Block-scoping prevents variables from "leaking out" into parts of the application where they don't belong.

The curly braces of if-statements enclose code blocks, so **let** variables declared therein are confined to that code block. This is not the case with **var**, which is global in scope even when it is declared within curly braces.

15. Write an "if-statement" with a **let** variable declared inside its curly braces. Declare the variable inside and outside the code block:

```
let meal = "lunch";

if (meal == "lunch") {
    let special = "Burritos";
}

console.log(meal); // lunch
console.log(special); // ERROR: special is not defined
```

We get **not defined**, because **special** is block-scoped to its if-statement; **special** does not exist in the global scope where we are attempting to access it.

16. Run the same test, but with **var**:

```
    if (meal == "lunch") {
        var special = "Burritos";
    }
    console.log(special); // Burritos
```

The "var variable" still exists after the if-statement, because, **var** is in the **global scope**, even though it was declared inside an "if-statement".

let instead of var for better scope control

This is why **let** is preferred over **var**. With **let**, you have better control over variable scope. With **var**, variables "leak out" from their blocks into the rest of the application.

global variables with let

With **let**, you can still have **global variables**. Just declare them outside of any curly braces, and they will be available throughout the script.

- **END Lesson 02.01**
- **NEXT Lab 02.02**
- **Lesson 02.02**

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

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:

```
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);
```

END Lab 02.01 SEE Lab 02.01 Solution

UNIT 02

LESSON 02.02

---

nested if-else logic

truthy-falsey values

&& (AND) operator

|| (AND) operator

---

nested if-else logic

In the previous lesson we saw if-else logic involving a planned activity:

- if it's rainy, go to the museum.
- else if it's sunny, go to the beach
- else (it's cloudy) so go to the park.

But what if there are also wind conditions that will determine our activities at the beach and park:

```
- If it is sunny, go to the beach.
    - If it is windy, go windsurfing.
    - If it is not windy, play frisbee.
- If it is cloudy, go to the park.
    - If it is windy, fly a kite.
    - If it is not windy, have a picnic.
- If it is rainy, go to the museum.
```

This is nested decision making, and it requires nested if-else logic.

1. Set up the basic, "non-nested" if-else logic, and add a new variable, windy, that we can use for "nested logic":

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

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

```
        console.log('Go to the park!');
    }
```

2. Wind conditions do not apply if we are at the museum. So, move on to the "else if", and add this nested logic that specifies what to do at the beach:

```
    if (weather == "rainy") {
        console.log('Go to the museum!');
    } else if (weather == "sunny") {
        console.log('Go to the beach!');
        if (windy == true) {
            console.log('Go windsurfing!');
        } else {
            console.log('Play frisbee!');
        }
    } else {
        console.log('Go to the park!');
    }
```

3. Moving on to the last part, specify what to do at the park:

```
    if (weather == "rainy") {
        console.log('Go to the museum!');
    } else if (weather == "sunny") {
        console.log('Go to the beach!');
        if (windy == true) {
            console.log('Go windsurfing!');
        } else {
            console.log('Play frisbee!');
        }
    } else {
        console.log('Go to the park!');
        if (windy) {
            console.log('Fly a kite!');
        } else {
            console.log('Have a picnic!');
        }
    }
```

4. Change weather and windy to get different output.

truthy and falsey values

Truthy and falsey values are not literally true or false, but they return true or false in a boolean context, i.e. in an "if-statement". Here are the falsey values:

falsey values

```
- NaN
- undefined
- null
- 0
- "", ''
```

All other values, including all strings and non-zero numbers, both positive and negative, are truthy.

A falsey resolves to false in an if-statement:

5. Do a test to confirm that **NaN** is falsey:

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

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

6. Do a test to confirm that **undefined** is falsey:

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

if(player1) {
    console.log('undefined is truthy');
} else {
    console.log('undefined is falsey'); // runs
}
```

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

8. Do a test to confirm that **0** is falsey:

```
let x = 0;
console.log('x', x, typeof(x));
// x 0 number

if(x) {
    console.log('0 is truthy');
} else {
    console.log('0 is falsey'); // runs
}
```

9. Do a test to confirm that empty strings are falsey:

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

if(zip) {
    console.log('empty strings are truthy');
} else {
    console.log('empty strings are is falsey'); // runs
}
```

- **END Lesson 02.02**
- **NEXT Lab 02.02**
- **Lesson 02.03**

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.
2. Write a nested if-else block, that satisfies the following conditions:
    - If the pet is a cat, increase the pet price by 20%, unless it is a 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 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.

- **END Lab 02.02**
- **SEE Lab 02.02 Solution**

02.02 Lab Solution

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.

```
let studentScore = 94;
let isHighSchool = true;
let grade;

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

console.log(grade);
```

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

    - If the pet is a cat, increase the pet price by 20%, unless it is a 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 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 = false;

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

console.log(price);
```

- **END Lab 02.02**
- **SEE Lab 02.02 Solution**

UNIT 02

LESSON 02.03

---

ternary expressions

&& (AND) operator

|| (OR) operator

ternary expression

A ternary expression is a concise alternative to an if-else statement. What takes an if-else five lines of code to accomplish, a ternary expression gets done in one.

Let's start with an if-else statement.

1. Declare three number variables and do some conditional math. If x is less than y, add them; othewise (else) multiply them:

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

if(x < y) {
    z = x * y
} else {
    z = x + y;
}

console.log(z);
```

console.log(z);

Now to convert the if-else to a ternary:

2. Add a question mark right inside the if-block:

```
if(x < y) {
    ? z = x * y;
} else {
    z = x + y;
}
```

3. Add a colon right inside the else-block:

```
if(x < y) {
    ? z = x * y;
} else {
    : z = x + y;
}
```

4. Delete the if() and else, including parentheses:

```
x < y {
    ? z = x * y;
} {
    : z = x + y;
}
```

5. Delete all the curly braces:

```
x < y
  ? z = x * y;
  : z = x + y;
```

6. This is all one line now, so get rid of the first semi-colon and back everything up onto the same line:

```
x < y ? z = x * y : z = x + y;

console.log(z);
```

7. You can even get rid of the second "z=" since it is assumed by the first one:

```
x < y ? z = x * y : x + y;

console.log(z);
```

CHALLENGE:

8. Convert this if-else into a ternary:

```
let n = 5;

if(n == 7) {
    n = 0;
} else {
    n++;
```

```
}

console.log(n);
```

9. Convert this if-else into a ternary:

```
let num = 20;

if(num == 20) {
  num++;
} else {
  num--;
}

console.log(num);
```

multiple conditions

Multiple conditions can be evaluated with the && (AND) and || (OR) operators:

The **&&** (AND) operator requires at least two conditions to be true.

10. Do an if statement with && where two conditions must be true.

```
    let city = 'Texarkana';
    let state = 'Texas';
    let msg = '';

    // && (AND) operator
    if(city == 'Texarkana' && state == 'Texas') {
        msg = `Welcome to ${city}, ${state}!`;
    } else {
        msg = 'This is not Texarkana, Texas--but it could be Texarkana,
Arkansas or Houston, Texas';
    }

    console.log(msg); // Welcome to Texarkana, Texas
```

11. Do an if statement with || where only one condition must be true.

```
    // || (OR) opertor
    if (city == 'Texarkana' || city == 'Houston') {
        msg = `Welcome to ${city}`; // runs
    } else {
        msg = 'This is neither Texarkana nor Houston';
    }
```

```
    console.log(msg); // Welcome to Texarkana
```

12. Change city to 'Dallas'`, and run it again. The else part will run both times.

There can be more than two **&&** conditions.

13. Try one with three conditons; if *any* of them are false, the **else** part runs:

```
    let R = 123;
    let G = 155;
    let B = 202;

    if(R > 100 && G > 100 && B > 100) { // true
        msg = 'All RBG values are greater than 100';
    } else {
        msg = 'At least one RBG value is 100 or less';
    }

    console.log(msg); // All RBG values greater than 100
```

14. Set any R, G, B values to below 100, and run it again. Now you get the else part.

15. Try using more than two || operators:

```
    let car = 'blue';

    if (car == 'black' || car == 'silver' || car == 'blue') {
        msg = 'car is black, sliver or blue.';
    } else {
        msg = 'The car is not black, sliver, or blue. It is actually ' +
car;
    }

    console.log(msg);
```

16. Change the car color to red so that the else part runs.

switch case break

An alternative to if-else if-else logic is a **switch-case-break** statement. Unlike a ternary, a switch-case-break is used for evaluating multiple conditions, which would otherwise require "else if" logic.

Convert if-else if-else to switch-case-break by following these guidelines:

- the evaluated variable is written just once
- there is only one set of curly braces
- there is only one set of parentheses

- there are no equality operators (==, ===)

Other differences: switch, case and break replace if-else keywords

- switch instead of if to start
- case instead of else if
- default instead of else

17. Write out this if-else if-else:

```
let moneySymbol = "JPY";
let currency = "";

if (moneySymbol === "USD") {
    currency = "US Dollar";
} else if (moneySymbol === "JPY") {
    currency = "Japanese Yen";
} else if (moneySymbol === "GBP") {
    currency = "British Pound";
} else {
    currency = "Unknown";
}
```

18. Convert the if-else if-else to a switch-case-break:

```
switch (moneySymbol) {
    case "USD":
    currency = "US dollar";
    break;
    case "JPY":
    currency = "Japanese Yen";
    break;
    case "GBP":
    currency = "British Pounds";
    break;
    default:
    currency = "Unknown";
    break;
}
```

- **END Lesson 02.03**
- **NEXT Lab 02.03**
- **NEXT Lesson 02.04**

1. Convert the following if-else into a ternary:

```
let msg = "";

if ('plums' < 'pears') {
    msg = 'plums b4 pears';
} else {
    msg = 'pears b4 plums';
}

console.log(msg);
```

2. Convert the following if-else into a ternary:

```
let x = 12;
let y = 6;
let z = 0;

if (x <= y) {
    z = x / y;
} else {
    z = y / x;
}

console.log(z);
```

3. Convert the following ternary into if-else:

```
let n = 8;

n == 8 ? console.log("equal") : console.log("not equal");
```

4. Convert the following if-else to switch-case-break:

```
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 = "Australia";
    }
```

- **END Lab 02.03**
- **SEE Lab 02.03 Solution**

02.03 Lab Solution

1. Convert the following if-else into a ternary:

```
let msg = "";

if ('plums' < 'pears') {
    msg = 'plums b4 pears';
} else {
    msg = 'pears b4 plums';
}
console.log('if else:', msg);

'plums' < 'pears' ? msg = 'plums b4 pears' : msg = 'pears b4 plums';
console.log('ternary:', msg);
```

2. Convert the following if-else into a ternary:

```
let x = 12;
let y = 6;
let z = 0;

if (x <= y) {
    z = x / y;
} else {
    z = y / x;
}

console.log(z);

x <= y ? z = x / y : z = y / x;
console.log(z);
```

3. Convert the following ternary into if-else:

```
let n = 8;

n == 8 ? console.log("equal") : console.log("not equal");

if(n == 8) {
    console.log("equal")
} else {
    console.log("not equal");
}
```

switch case break

4. Convert the following if-else to switch-case-break:

```
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 = "Australia";
}

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 "France":
    continent = "Europe";
    break;
    default:
    continent = "Australia";
    break;
}
```

- **END Lab 02.03 Solution**

Date Object

Outputting date and time to the web page

The Date Object returns the full date and time from the user's computer. It is instantiated (declared) using the **new** keyword.

1. Instantiate an instance of the Date object.

```
let dateTime = new Date();
console.log(dateTime);
```

The individual time units are available by calling the Date object's "get methods".

2. Get the current hour, minute and second:

```
let hour = dateTime.getHours();
console.log(hour); // 0-23

let minute = dateTime.getMinutes();
console.log(minute); // 0-59

let second = dateTime.getSeconds();
console.log(second); // 0-59
```

3. Express the time in 00:00:00 format:

```
let timeIs = `${hour}:${minute}:${second}`;
console.log(timeIs);
```

leading zeroes for minute and second

If minute or second is less than 10, you get wonky output, such as 1:2:3, instead of 1:02:03. To fix this, add leading zeros to minute and second, as needed. This is done with conditional logic.

4. Add a leading 0 if minute or second is less than 10:

```
if(minute < 10) {
  minute = '0' + minute;
}

if(second < 10) {
  second = '0' + second;
}

timeIs = `${hour}:${minute}:${second}`;
console.log(timeIs);
```

If an if-statement has only one line of code inside its curly braces, you can omit the curly braces altogether and put everything on the same line.

5. Make these short if-statements even more concise by eliminating the curly braces:

```
if(minute < 10) minute = '0' + minute;
if(second < 10) second = '0' + second;

timeIs = `${hour}:${minute}:${second}`;

console.log('timeIs w leading 0', timeIs); // 00:00:00
```

converting military time to AM/PM

The hour is from 0-23 ("military time"), so 3pm is 15:00 and 10pm is 22:00.

To convert to AM/PM time, we need:

- a variable to store the string "AM" or "PM".
- two if-statements, done in this order:
  - if hour > 11, use "PM".
  - if hour > than 12, subtract 12.

6. Declare a variable **amOrPm** with an initial value of 'AM', and follow that with the if-statements:

```
let amOrPm = 'AM';

if(hour > 11) {
    amOrPm = 'PM';
}

if(hour > 12) {
    hour -= 12;
}
```

```
    timeIs = `${hour}:${minute}:${second} ${amOrPm}`;
    console.log('time is: ', timeIs);
```

timely greeting

Now let's make a "timely greeting" that is appropriate for the current hour:

- **if** the hour is less than 12 (noon), say "Good morning!".
- **else if** the hour is less than 18 (6:00pm), say "Good afternoon!"
- **else**, say "Good Evening!"

Start the greeting with "Good" and then use += to concatenate the "timely" part:

7. Get a fresh hour, since our original hour may have already had 12 subtracted from it:

```
    let hr = dateTime.getHours();
```

8. Declare greeting with an initial value of 'Good':

```
    let greeting = "Good ";
```

9. Do the logic for **hr < 12** (noon):

```
    if(hr < 12) {
        greeting = "morning";
    }
```

10. Add an **else if** for **hr < 18** (6pm). Follow that with an **else** that runs when **hr** is 18 and up:

```
    if (hr < 12) {
      greeting += "Morning!";
    } else if (hr < 18) {
      greeting += "Afternoon!";
    } else {
      greeting += "Evening!";
    }

    console.log(greeting);
```

"Timely Greeting"

11. Pair the greeting with the time of day in AM-PM format:

```
let timelyGreeting = `${greeting} The time is: ${timeAMPM}`;
console.log(timelyGreeting);
```

12. Output the "timely greeting" to the web page. Start by getting the tag that will display the greeting:

```
let greetingTag = document.getElementById('greeting');
```

13. Set the **textContent** property of the tag object to **timelyGreeting**:

```
greetingTag.textContent = timelyGreeting;
```

The Date object's other time units can be used to concatenate and output today's date:

14. Get today's date:

```
let date = dateTime.getDate();
console.log('date', date);
```

15. Get the month, which is returned as a number from 0-11 (Jan = 0, Dec = 11):

```
let month = dateTime.getMonth();
console.log('month', month);
```

16. Get the month as a string (January, February, etc). This gives us the flexibility to use the month as either a number or a day:

```
let fullMonth = dateTime.toLocaleString('default', {month:'long'})
console.log('fullMonth', fullMonth);
```

17. Get the day of the week, which is a number, with Sunday=0 and Saturday=6:

```
let day = dateTime.getDay();
console.log('day', day);
```

18. Make an array of the days of the week.

```
let daysArr = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday"];
```

We'll explore arrays thoroughly in upcoming lessons, but basically, an array is a variable that stores multiple values as a list, inside square brackets.

19. Look up the first item in the array (Sunday) by its index (0):

```
let Sun = daysArr[0];
console.log('Sun', Sun); // Sun Sunday
```

20. Get the day of the week by looking it up index (number) in the array:

```
let dayOfWeek = daysArr[day];
console.log('dayOfWeek', dayOfWeek); // string
```

21. Get the full 4-digit year:

```
let fullYear = dateTime.getFullYear();
console.log(fullYear);
```

22. Concatenate today's date as **Day, Month Date, Year**, using the non-numeric day and month, e.g. **Tuesday, May 17, 2022**:

```
let todaysDate = `${dayOfWeek}, ${fullMonth} ${date}, ${fullYear}`;
console.log('todaysDate', todaysDate); // string
```

23. Output today's date to its place on the web page:

```
let todaysDateTag = document.getElementById('todays-date');

todaysDateTag.innerHTML = todaysDate;
```

updating the time every second

You may be wondering: Why doesn't the time on the web page updating every 1 second? To auto-update every second would require more code. In a later lesson, we'll get into using **setInterval** to call a function X times per second to do just this sort of thing.

END Lesson 02.04

NO LAB EXERCISES

PROCEED DIRECTLY TO LESSON 03.01

1. Continuing where we left off with the "timely greeting", add to the logic so that:

- if the hour is between midnight and 2 AM, the greeting is "Good Evening"
- if the hour is between 2-4 AM, greeting is "Hey, Night Owl!"
- all other greeting times remain the same

```
let dateTime = new Date();
let hr = dateTime.getHours();
let greeting = 'Good ';

if(hr < 12) {
    greeting += "Morning!";
} else if(hr < 18) {
    greeting += " Afternoon!";
} else {
    greeting += 'Evening!';
}

console.log(greeting);
```

2. Test the output by hard-coding **hr**. Sample hr values and their expected output:

- When **hr = 1**, output should be "Good Evening!";
- When **hr = 3**, output should be "Hey, Night Owl!";
- When **hr = 6**, output should be "Good Morning!";
- When **hr = 15**, output should be "Good Afternoon!";
- When **hr = 20**, output should be "Good Evening!";
- **END Lab 02.04**
- **SEE Lab 02.04 Solution**

02.04 Lab Solution

1. Continuing where we left off with the "timely greeting", add to the logic so that:

- if the hour is between midnight and 2 AM, the greeting is "Good Evening"
- if the hour is between 2-4 AM, greeting is "Hey, Night Owl!"
- all other greeting times remain the same

```
let dateTime = new Date();
let hr = dateTime.getHours();
let greeting = 'Good ';

if(hr < 2) {
    greeting += "Evening!";
} else if(hr < 4) {
    greeting = "Hey, Night Owl!";
} else if(hr < 12) {
    greeting += " Morning!";
} else if(hr < 18) {
    greeting += " Afternoon!";
} else {
    greeting += 'Evening!';
}

console.log(greeting);
```

2. Test the output by hard-coding **hr**. Sample hr values and their expected output:

- When **hr = 1**, output should be "Good Evening!";
- When **hr = 3**, output should be "Hey, Night Owl!";
- When **hr = 6**, output should be "Good Morning!";
- When **hr = 15**, output should be "Good Afternoon!";
- When **hr = 20**, output should be "Good Evening!";

```
let dateTime = new Date();
let hr = dateTime.getHours();
let greeting = 'Good ';

// HARD-CODED EXAMPLE:
hr = 2;

if(hr < 2) {
    greeting += "Evening!";
} else if(hr < 4) {
    greeting = "Hey, Night Owl!";
} else if(hr < 12) {
    greeting += " Morning!";
} else if(hr < 18) {
```

```
        greeting += " Afternoon!";
    } else {
        greeting += 'Evening!';
    }

    console.log(greeting); // Hey, Night Owl!
```

- **END Lab 02.04 Solution**

---

functions

parameters and arguments

return values

function-scope vs global scope

default parameter values

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 typically (but not always) has a name by which to call it.

Mozilla mdn web docs defines functions in this way:

> Functions are one of the fundamental building blocks in JavaScript. A function in JavaScript is similar to a procedure—a set of statements that performs a task or calculates a value, but for a procedure to qualify as a function, it should take some input and return an output where there is some obvious relationship between the input and the output. To use a function, you must define it somewhere in the scope from which you wish to call it.

A function *should take some input and return an output*, so it is useful to think of a function as an input-output machine. Think of real world analogies:

- coffee grinder as real world function (input-output machine):
  - input: you put coffee beans in
  - procedure: blades turn inside the machine
  - output: out comes ground coffee

All that said, it is possible to have a function without inputs (parameters). This is where we will begin:

how to define (declare) a function

Follow these steps:

1. Begin by writing the keyword: `function`

2. Give the function a name. Variable naming rules apply. Since a function *does* something, it is customary for the name to begin with a verb: `function sayHello`

3. Next put a pair of parentheses: `function sayHello()`

The parentheses are for passing data -- *inputs* -- to the function, but some functions don't need inputs, and so the parentheses are left empty.

4. Next add a pair of curly braces: `function sayHello() {}`

5. Inside the curly braces, put the code that will run when the function is called:

```
    function sayHello() {
        console.log("Hello!");
    }
```

6. Run this example and check the Console. Nothing happens. Where's "Hello"?

Remember, a function must be *called*, but all we have done is *define* it.

7. We do not have a button to click to call the function, so's call it directly in the code:

```
    sayHello();
```

8. Run the code in the Console again; this time you get the "Hello!".

variable scope

global variables are available everywhere in a script: - **var** declared outside of a function is global. - **let** declared outside of a code block is global. - **const** (constants) follow the same scoping rule as let.

function scoped (local) variables - **var**, **let** declared inside a function is local to that function. - **let** declared inside a code block is available only to that code block.

// 9. Declare a local variables inside the function:

```
    function welcomePlayer() {
        var username = "Pikachubaca";
        let highScore = '6234';
        console.log(`Welcome back, ${username}! Your high score is
${highScore}`);
    }

    welcomePlayer();
```

10. Try to access the function variables in the global scope, outside the function:

```
    // console.log('username', username); // Error: username is not
defined
    // console.log('highScore', highScore); // Error: username is not
defined
```

parameters-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 call parentheses.

11. Write a function with two parameters:

```
    function greetPlayer(username, highScore) {
        console.log(`Welcome, ${username}! Your high score is
${highScore}!`);
    }
```

12. Call the function twice with different arguments each time:

```
    greetPlayer("Brian88", 12345);
    // Welcome, Brian88! Your high score is 12345!

    greetPlayer("Bob", 78967);
    // Welcome, Bob! Your high score is 78967!
```

If a function expects an argument, but none is provided, the missing value will be 'undefined'.

13. Call the function again, but this time omit the second argument:

```
    greetPlayer('Sally123');
    // Welcome, Sally123 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.

14. Write a new function, greetPlayerScore(). It is the same as greetPlayer, but with a default high score of 1000:

```
    function greetPlayerScore(username, highScore=1000) {
        console.log(`Welcome, ${username}! Your high score is
```

```
${highScore}`);
    }
```

15. Call the function twice, but the second time, only pass in the username. The high score will default to 1000:

```
greetPlayerScore("Xyz1", 13240);
// Welcome, Xyz1! Your high score is 13240!
greetPlayerScore("Abc2");
// Welcome, Abc2! Your high score is undefined!
```

Global variables are available everywhere, including to all functions:

16. Declare some global vars and use them in a function;

```
let num1 = 10;
let num2 = 14;
let sum = 0;

function addNums() {
    sum = num1 + num2;
    console.log("sum", sum);
}

addNums(); // sum 24
```

A more self-contained approach would be to num1 and num2 to the funcition as parameters (unless num1 and num2 are needed elsewhere in the script).

refactoring Refactoring code means changing it for the better.

17. Refactor addNums() as addNumbers() and give it parameters. The values of num1 and num2 will be passed in as the arguments when the function is called:

```
function addNumbers(num1, num2) {
    let sum = num1 + num2;
    console.log(sum, sum);
}

addNumbers(12, 18); // sum 30
addNumbers(123, 184); // sum 307
```

return value

So far, our output has consisted only of console.log. This is useful for tesing and debugging, but once the output is logged, that data is gone. The answer is for the output to be a "return value" which we can save in

the script.

return value

To return a value from a function, put the keyword 'return' in front of a value.

Save the return value by setting the function call equal to a variable.

A return not only returns a value, it exits the function.

18. Define a new function, that is similar to the previous one, but with one big exception: this one has a return value:

```
let numeroUno = 7;
let numeroDos = 8;

function multiplyNumbers(n1, n2) {
    return n1 * n2;
}
```

19. Call the function, setting the function call itself equal to a variable to save the return value:

```
let prod1 = addNums3(12, 18);
console.log('prod1', prod1);

let prod2 = addNums3(3412, 5618);
console.log('prod2', prod2); // 9030
```

20. Refactor the greet function to return a value. Use a new name: getGreeting.

- "get" in function names means it returns a value
- replace the `console.log` with `return`
- set the function calls equal to variables
- log the variables to output the return values

```
function getGreeting(username, highScore) {
    return `Welcome, ${username}! Your high score is ${highScore}`;
}

let welcome1 = getGreeting("Viper", 73489);
console.log('welcome1:', welcome1);
// Welcome, Viper! Your high score is 73489!

let welcome2 = getGreeting("Ktrav26", 12345);
console.log('welcome2:', welcome2);
// Welcome, Ktrav26! Your high score is 12345!
```

21. Declare a function **calcCafeBill()**.

- Give it three parameters: **subTotal**, **tipPct** and **taxRate**, the latter two with default values: **tipPct=15** and **taxRate=4**. This assigns a default 15% tip and default tax rate of 4% if no value(s) are provided for these.

```
function calcCafeBill(subTotal, tipPct=15, taxRate=4) {

}
```

The function calculates the tax and tip and adds those amounts to the subtotal

- calculate tip:
  - **let tip = subTotal * tipPct / 100;**
- calculate tax:
  - **let tax = subTotal * taxRate / 100;**
- add tax and tip to subtotal to get total:
  - **let total = subTotal + tax + tip;**

22. Write the calculations. The tax and tip percents are divided by 100, because 15% is not 15, but rather 0.15:

```
function calcCafeBill(subTotal, tipPct=15, taxRate=4) {
    let tax = subTotal * taxRate / 100;
    let tip = subTotal * tipPct / 100;
    let total = subTotal + tax + tip;
}
```

23. Concatenate the bill and return it:

```
function calcCafeBill(subTotal, tipPct=15, taxRate=4) {
    let tax = subTotal * taxRate / 100;
    let tip = subTotal * tipPct / 100;
    let total = subTotal + tax + tip;
    let bill = `Sub-total:  $ ${subTotal}
    Tax Rate:   ${taxRate} %
    Tip Pct:    ${tipPct} %
    Tip:        ${tip}%
    Tax:        $ ${tax}
    TOTAL:      $ ${total}
    Thank you!`;
    return bill;
}
```

24. Add toFixed(2) to round the cents to 2 decimal places:

```
    Tip:        ${tip.toFixed(2)} %
    Tax:        $ ${tax.toFixed(2)}
    TOTAL:      $ ${total.toFixed(2)}
```

25. Call the function, and supply all three parameters. Make the tip 20 percent and the tax 6 percent. This overrides the default tipPct and taxRate.

```
// $80 subtotal, 20% tip, 6% tax
let bill1 = calcCafeBill(80, 20, 6);
console.log(bill1);
```

26. Call the function again, but this time omit the third argument (tax). The tax rate will therefore default to 4%:

```
// $90 subtotal, 18% tip, 4% tax
let bill2 = calcCafeBill(90, 18);
console.log(bill2);
```

27. Call the function, passing in only one parameter, understood to be subTotal. tipPct and taxRate will use the defaults:

```
// $65 subtotal,  15% default tip,  4% default tax
let bill3 = calcCafeBill(65);
console.log(bill3);
```

END Lesson 03.01

NEXT: Lab 03.01

1. The weight of anything on the moon is approximately one-sixth its weight on earth. So, a person who weighs 60 kg on earth would weigh 10 kg on the moon. Write a **function calcMoonWeight** that takes an earth weight as its parameter, converts it to moon weight, and returns the moon weight

2. Complete this function to convert the inputted **feet** value to meters and **return** the result. Conversion units: **39.37 inches = 1 meter**. Call the function, saving the return value to a variable. Log the variable.

```
function convertFeetToMeters(feet) {
    // convert feet to meters
}
```

3. . Make a function called **squareNum** that:

    - takes in one number as its input
    - squares that number and returns the result So, if you input 4, it logs 16. Run the function three times with different inputs.

4. Make a function that:

    - takes TWO positive integers as its inputs (arguments)
    - raises the first number to the power of the second number
    - returns the answer So, if you input (5, 3), you get back 125. Run the function three times with different inputs.

5. Make another version of the previous function that:

    - takes one number as its input (argument)
    - if the number is even, it squares the number
    - but if the number is odd, it cubes the number
    - returns the answer. So, if you input 3, you get back 27. So, if you input 4, you get back 16. Run the function three times with different values.

6. Declare a function called **introducePet**, that:

    - has four parameters: **pet**, **name**, **age** and **sound**
    - returns a message, such that if the arguments are **cat**, **Fluffy**, **3** and **Meow**, the returned message is: **Meow! My name is Fluffy! I am a 3-year-old cat!**. Run the function three times, with different pet inputs each time.

7. Declare a function with two parameters, **num1** and **num2**. The function call passes in two arguments, both numbers.
   The function does the following math:
   - If the **num1** is greater than **num2**, subtract **num2** from **num1**
   - If **num1** is less than or equal to **num2**, add the numbers together.
     Return the answer. Run the function twice, once with the numbers being subtracted, the other with the numbers being added.

8. Given: two sides of a right triangle as global variables **sideA** and **sideB**

- Write a function with parameters **a** and **b**
  - Function uses the Pythagorean Theorem (**$a^2 + b^2 = c^2$**) to find the hypotenuse, **c**, of **a** and **b**.
  - Function returns **c**, the hyotenuse.
  - Call the function, passing in **sideA** and **sideB** as its two arguments.

9. Write a function that:

- takes in numbers of pennies, nickels, dimes and quarters
- calculates the total value of all coins
- returns the total as dollars and cents, to two decimal places and with dollar-sign

10. Define a function that:

- takes the radius of a circle as its input
- calculates the area of the circle using the formula `A = πr2`
- returns the area

Call the function, passing in `radius` as its argument. Set the function call equal to a variable to save the return value.

END Lab 03.01
SEE Lab 03.01 Solution

Lab 03.01 - Solution:

1. **calcMoonWeight** takes an earth weight as its parameter, converts that to moon weight by dividing by 6 and returns that value:

```
function calcMoonWeight(earthWt) {
    // convert earth wt on moon wt
    return earthWt / 6;
}

let earthWt = 300;
let moonWt = calcMoonWeight(earthWt);
console.log(moonWt); // 50
```

2. Complete this function to convert the inputted **feet** value to meters and **return** the result. Conversion units: **39.37 inches = 1 meter**.

```
function convertFeetToMeters(feet) {
    // convert feet to meters
    return feet * 12 / 39.37;
}
```

3. Make a function called **squareIt** that:
    - takes in one number as its input
    - squares that number and returns the result So, if you input 4, it logs 16. Run the function three times with different inputs.

```
function squareNum(x) {
    return x ** 2; // or: x * x;
}

console.log(squareNum(4)); // 16
console.log(squareNum(5)); // 25
console.log(squareNum(6)); // 36
```

4. Make a function called **powerUp** that:
    - takes TWO positive integers as its inputs (arguments)
    - raises the first number to the power of the second number
    - returns the answer So, if you input (5, 3), you get back 125. Run the function three times with different inputs.

```
function powerUp(x,y) {
    return x ** y;
}
```

```
let pow44 = powerUp(4,4);
console.log(pow44); // 256

let pow53 = powerUp(5,3);
console.log(pow53); // 125

let pow62 = powerUp(6,2);
console.log(pow62); // 36
```

5. Make another version of the previous function that:
    - takes one number as its input (argument)
    - if the number is even, it squares the number
    - but if the number is odd, it cubes the number
    - returns the answer. So, if you input 3, you get back 27. So, if you input 4, you get back 16. Run the function three times with different values.

```
function squareOrCubeIt(x) {
    let answer = 0;
    // if x is even, dividing by 2 yields remainder of 0
    if(x % 2 == 0) { // if x is even, square it
        answer = x * x;
    } else { // x is odd, so cube it
        answer = x ** 3;
    }
    return answer;
}

let a = squareOrCubeIt(5); // cube it
console.log(a); // 125

let b = squareOrCubeIt(6); // square it
console.log(b); // 36

let c = squareOrCubeIt(7); // cube it
console.log(c); // 343
```

6. Declare a function called **introducePet**, that:
    - has four parameters: **pet**, **name**, **age** and **sound**
    - return a message, such that if the arguments are **cat**, **Fluffy**, **3** and **Meow**, the returned message is: **Meow! My name is Fluffy! I am a 3-year-old cat!**. Run the function three times, with different pet inputs each time.

```
function introducePet(pet, name, age, sound) {
    return `${sound}! My name is ${name}! I am a ${age}-year-old
${pet}`;
}
```

```
let petIntro1 = introducePet('cat', 'Fluffy', 3, 'Meow');
console.log(petIntro1);
// Meow! My name is Fluffy! I am a 3-year-old cat!

let petIntro2 = introducePet('dog', 'King', 2, 'Woof');
console.log(petIntro2);
// Woof! My name is King! I am a 2-year-old cat!

let petIntro3 = introducePet('canary', 'Tweety', 4, 'Chirp');
console.log(petIntro3);
// Chirp! My name is Tweety! I am a 4-year-old canary!
```

7. Declare a function with two parameters, **num1** and **num2**. The function call passes in two arguments, both numbers.
   The function does the following math:
   - If the **num1** is greater than **num2**, subtract **num2** from **num1**
   - If **num1** is less than or equal to **num2**, add the numbers together.
     Return the answer. Run the function twice, once with the numbers being subtracted, the other with the numbers being added.

```
function addOrSubtractNums(num1, num2) {
    let answer = 0;
    if(num1 > num2) {
        answer = num1 - num2;
    } else {
        answer = num1 + num2;
    }
    return answer;
}

let answr1 = addOrSubtractNums(30, 20); // 10
let answr2 = addOrSubtractNums(20, 30); // 50
```

8. Given: two sides of a right triangle as global variables **sideA** and **sideB**

- Write a function with parameters **a** and **b**
  - Function uses the Pythagorean Theorem (**$a^2 + b^2 = c^2$**) to find the hypotenuse, **c**, of **a** and **b**.
  - Function returns **c**, the hyotenuse.
  - Call the function, passing in **sideA** and **sideB** as its two arguments.

```
let sideA = 3;
let sideB = 4;

function findHypotenuse(a, b) {
    let cSquared = a**2 + b**2;
    let c = Math.sqrt(cSquared);
    return c;
}
```

```
    let c1 = findHypotenuse(sideA, sideB);
    console.log(c1); // 5

    let c2 = findHypotenuse(6, 8);
    console.log(c2); // 10

    let c3 = findHypotenuse(9, 12);
    console.log(c3); // 15
```

9. Write a function that:

- takes in numbers of pennies, nickels, dimes and quarters
- calculates the total value of all coins
- returns the total as dollars and cents, to two decimal places and with dollar-sign

```
    function countCoins(pennies, nickels, dimes, quarters)  {
        let cents = (pennies) + (nickels * 5) + (dimes * 10) + (quarters *
25);
        return '$' + ((cents / 100).toFixed(2));
    }

    let money1 = countCoins(250, 50, 25, 10); // 2.50 + 2.50 + 2.50 + 2.50
= $10.00
    console.log(money1);

    let money2 = countCoins(100, 100, 100, 100); // 1 + 5 + 10 + 25 =
$41.00
    console.log(money2);
```

10. Define a function that:

- takes the radius of a circle as its input
- calculates the area of the circle using the formula `A = πr2`
- returns (outputs) the area

```
    function findAreaOfCircle(r) {
        let area = Math.PI * r**2;
        return area;
    }

    let area1 = findAreaOfCircle(6);
    console.log(area1); // 113.09733552923255
```

END Lab 03.01 Solution

NEXT: Lesson 03.02

DOM (Document Object Model) functions

events calling functions (click, change)

Document Object Model (DOM)

JS "sees" the web page as a hierarchical collection of objects, known as the DOM (Document Object Model).

DOM manipultion

JS can make stuff happen and things change on the web page. This is known as "DOM manipultion".

DOM event

A DOM event is something that happens on the web page that can call a function. Functions that manipulate the DOM are typically triggered by a button click, menu change, or other **DOM event**.

HTML elements as JS Objects

Any HTML element can be brought into JS where it becomes an object. The commands for getting elements are:

- **document.getElementById()** gets the element of specified id and stores it in an object
- **document.querySelector()** gets the first element of specified class or tag name and stores it in an object
- **document.querySelectorAll()** gets all elements of specified class or tag name and stores them in an array called a Node List

button.addEventListener()

To enable a button or other element to call a function, attach the **addEventListener** method to the object. The method requires two arguments: an event to 'listen' for and a function to call when the event takes place.

DOM function exploration page

This lesson has a page already set up with the HTML and CSS for several buttons, some input text fields and a select menu.

Open the HTML file in the browser and have a look: lots of buttons and things, but nothing works yet, because it is our job to write the code for that.

return value vs. no return value

Before we get started, it is important to note that these functions do not **return** a value. The function output is being displayed on the web page, rather than being stored in a variable in the script. That said, it is possible to combine return values with DOM output, as we will see.

function #1: greetWorld()

We want the **greet world** button to call a **greetWorld()** function. The button is in the html as: `greet world`

1. In the JS START file, add this function, and call it from the script (no button yet):

```
function greetWorld() {
  alert('Hello World');
}
greetWorld();
```

2. Comment out the function call since a button click is taking over.
3. Get the element `greet world` by id so that we can tell it to call the function. Save it to a variable so that we can tell it to call a function:

```
const btn1 = document.getElementById('btn-1');
```

4. Also get the

   output

   tag to hold the output. Save it to a variable so that we can output our function results there:

```
const output = document.getElementById('output');
```

const for objects

As we learned in a previous lesson, **const** for primitive data types (string, number, boolean) cannot be changed in any way.

But **const** works differently for objects:

- **const** objects cannot be *mutated*, which means the *data type* cannot be changed.
- **const** properties, however, *can** be changed.
- **const** protects objects from being accidentally changed to, say, a string, but otherwise you can change/add/remove properties at will.

For these reasons, we typically use **const** to save DOM objects that we bring into JS.

element attributes become object properties

Once in JS, the elements' attributes are the object's properties.

5. Log **btn1** and **object** as well as and some properties:

```
console.log(btn1);
console.log(btn1.id);
console.log(btn1.textContent);
console.log(output);
console.log(output.id);
console.log(output.textContent);
```

addEventListener()

The **addEventListener()** is called on an object and takes two arguments: an event to "listen" for and a function to call when the event takes place.

6. Tell the **btn1** object to "listen" for a click upon itself and to call **greetWorld** when the click occurs.

```
btn1.addEventListener('click', greetWorld);
```

Notice that it is **greetWorld** *not* **greetWorld()**. This is because the parentheses would instantly call the function. We want the funtion to wait for the click.

7. Run the page and click the button. You should get the Hello World alert.

outputting to the DOM

The function works, so next let's have the output appear on the web page, rather than in a pop up.

8. In the function, comment out the alert and switch to displaying the message as the **textContent** property of the **output** tag:

```
function greetWorld() {
  // alert('Hello World');
  output.textContent = 'Hello World';
}
```

.textContent

The **textContent** property is everything between the tags, but cannot include any html. It is therefore best for simple strings, such as 'Hello World'.

.innerHTML

innerHTML refers to everything between the tags, including html, so tags within tags.

9. Change the output to be a link, which necessitates a switch from *textContent* to *innerHTML*:

```
function greetWorld() {
  // alert('Hello World');
  output.innerHTML = '<a href="#">Hello World</a>';
}
```

function #2: greetByName()

This next function also runs on button click. It gets the user-inputted names from the input fields, concatenates a personalized greeting, and outputs it to the page.

The html is already written:

```
<div>
  <input type="text" id="firstName" placeholder="First Name">
  <input type="text" id="lastName" placeholder="Last Name">
  <button id="btn-2">greet by name</button>
</div>
```

10. Get the button `greet by name`:

```
const btn2 = document.getElementById('btn-2');
```

11. Have the button "listen" for a click and call the function when the click occurs:

```
btn2.addEventListener('click', greetByName);
```

12. Write the function. Get the **values** of the text input fields, which have id's of **firstName** and **lastName**. The values are whatever the user typed:

```
function greetByName() {
    let fName = document.getElementByIc('firstName').value;
    let lName = document.getElementByIc('lastName').value;
    output.textContent = **Hey, ${fname} $lName}**;
}
```

13. Run the page and enter a first and last name.

14. Click the **greet by name** button. The personalized output should appear.

function #3: pickFruit()

This next function runs when the user chooses a fruit from the menu. The html is already there:

```
<select id="fruit-menu">
  <option value="choose">Pick a Fruit:</option>
  <option value="apple">Apple</option>
  <option value="banana">Banana</option>
  <option value="cherry">Cherry</option>
  // etc. more options
</select>
```

The way it works is:

- user chooses a fruit from the **select** menu.
- the choice is a **change** event, which calls the function, which:
    - gets the menu **value**, which equals the value of the selected **option**
    - concatenates and outputs a message to the **output** tag.

15. Get the select menu:

```
let fruitMenu = document.getElementById('fruit-menu');
```

16. Tell it to listen for a **change** event. When it hears the change (to itself), it calls a **pickFruit** function:

```
fruitMenu.addEventListener('change', pickFruit);
```

17. Write the function:

- Get the value of **fruitMenu** and save it to a local variable, **fruit**.
- Append **toUpperCase()** to make the fruit UPPERCASE.
- Concatenate and output a message about the chosen fruit.

```
function pickFruit() {
    let fruit = fruitMenu.value.toUpperCase();
    output.textContent = **Your chosen fruit is: ${fruit}!**;
}
```

function #4: addNumbers()

Next, we will do a function that adds numbers inputted by the user and outputs the sum.

We already have the html for this:

```
<div>
  <input type="number" id="num1-box" placeholder="Enter a Number">
  <input type="number" id="num2-box" placeholder="Enter a Number">
```

```
  <button id="btn-3">Add Numbers</button>
</div>
```

The **input** elements are set to **type="number"** to prevent non-numeric characters from being entered.

19. Get the **add numbers** button and tell it to call a function when clicked:

```
let btn3 = document.getElementById('btn-3');
btn3.addEventListener('click', addNumbers);
```

20. Write the function. Get the values from the input boxes. Then add the numbers and concatenate the output:

```
function addNumbers() {
  let num1 = document.getElementById('num1-box').value;
  let num2 = document.getElementById('num2-box').value;
  let sum = num1 + num2;
  output.textContent = sum;
}
```

21. Run the page. Enter two numbers and click the **add numbers** button.

There is a bug. The sum is no sum at all, but rather a concatenation of the two numbers. This happened because even though the input boxes are of type number, the "numbers" still came into JS as strings.

22. Convert the numeric input to actual numbers so that the + adds them:

```
  function addNumbers() {
    let num1 = document.getElementById('num1-box').value;
    let num2 = document.getElementById('num2-box').value;
    num1 = Number(num1);
    num2 = Number(num2);
    let sum = num1 + num2;
    output.textContent = sum;
}
```

23. Run the page again. The math should work now.

data-attribute

You can attach additional data to any tag by adding a "data-name" attribute. The data is available in JS as: object.dataset.name

24. Comment out the select menu and "uncomment out" the other select menu--the one where each option has a data-food attribute:

```
    <select id="fruit-menu">
        <option value="choose">Pick a Fruit:</option>
        <option value="apple" data-food="apple sauce">Apple</option>
        <option value="banana" data-food="banana bread">Banana</option>
        - ETC -
    </select>
```

options array

The select object has an options property, which is an array. We will get into arrays in Unit 4, but for now, just know that an array is a variable that holds more than one item at a time. The items are stored by number, called index. We can look up an array item by index. When a menu choice is made, the object selectedIndex property is set with a number that corresponds to the option chosen. We can use that number to look up the selected option from the options array. Once we have that option, can access any **data-** attribute via the **dataset** property. We have an attribute data-food, the value of which is **dataset.food**

Refactor the pickFruit function to include the food:

25. In the function, save the we numeric index of the selected item to a variable. This simplifies the next line.

```
    let indx = fruitMenu.selectedIndex;
```

26. Get the selected option by its index in the options array. Save that to a variable **optn**:

```
    let optn = fruitMenu.options[indx];
```

27. Get the value of the **data-food** attribute for the option:

```
    let food = optn.dataset.food;
```

28. Add the food to the output:

```
      output.textContent = `Your chosen fruit is: ${fruit}! Do you like
  ${food}?`;

    } // end function pickFruit()
```

END LESSON 03.02

NEXT LESSON 03.03

Login Form

event object

event.preventDefault()

hiding DOM elements with display='none'

_____-

We have a Login form that we will make work in an emulation scenario. The html and css for the form are all ready.

Open the html page in the browser to have a look at the form. It doesn't work yet--that's our job to write the code to make it work.

How the log in works:

- The user will enter a username and password
- The user clicks the LOGIN button to call the login() function
- The function gets the values inputted in the username and password fields
- The entered values are compared to the "real" username and password
- If they both match, we get a welcome message and the form elements disappear
- If either the username or password are wrong. we get a 'login failed' message

There are several DOM elements that we need in our login function:

1. Get the login button and save it as an object:

```
let loginBtn = document.getElementById('login-btn');
```

2. Make the button clickable to call the login function:

```
loginBtn.addEventListener('click', login);
```

3. Get the 'Please Log In' heading, since we will hide it on login success:

```
let plsLogin = document.getElementById('pls-login');
```

4. Get the username and password input boxes:

```
let userBox = document.getElementById('username');
let pswdBox = document.getElementById('password');
```

This login form doesn't have a database connection to look up a real username and password, so we will hard-code our username and password:

5. Hard-code the "correct" username and password:

```
let correctUser = "joe";
let correctPswd = "abc";
```

6. Get the response element, which is where the output goes:

```
let response = document.getElementById('response');
```

event.preventDefault() When a button inside a form is clicked, it reloads the page by default. This is to export the form variables to a server-side processing script. But since we don't have server connectivity, are just hard-coding the username and password. We don't want to reload, because doing so will clear the input boxes. So to prevent that default behavior, we call the `event.preventDefault()`.

event object

The `event` object is default parameter that is there even if you don't add it as a parameter. The default name is `event`, but you can rename the event object anything you like; `evt` and `e` are the most commonly used aliases.

7. Write the function, passing in the **event** argument and calling the **event.preventDefault()** method:

```
function login(event) {
    event.preventDefault();
}
```

8. Get the inputted `username` and `password`. These are the `value` properties of their respective input objects. We just need the values--not the whole objects.

```
function login(event) {
    event.preventDefault();
    let inputtedUser = userBox.value;
    let inputtedPswd = pswdBox.value;
}
```

9. Compare the entered username and password to the "correct" values. Since there are two conditions to evaluate, use the `&&` operator:

```
    function login(event) {
        event.preventDefault();
        let inputUser = userBox.value;
        let inputPswd = pswdBox.value;

        if(inputUser === correctUser && inputPswd === correctPswd) {
        }

    } // end login function
```

10. If login was successful, output the welcome message and hide the form elements by setting their `display` property to `none`:

```
    function login(event) {

        event.preventDefault();

        let inputUser = userBox.value;
        let inputPswd = pswdBox.value;

        if(inputUser === correctUser && inputPswd === correctPswd) {
            response.style.fontSize = '1.5em';
            response.textContent = 'Welcome, ' + correctUser;
            userBox.style.display = "none"; // hide username field
            pswdBox.style.display = "none"; // hide password field
            loginBtn.style.display = "none"; // hide Log In button
            plsLogin.style.display = "none"; // hide Please Log In
        }

    } // end login function
```

11. Add an `else` part to run if either the username or password are incorrect:

```
        } else {
            response.textContent = "Log in failed. Try again.";
        }

    } // end login function
```

leaving out event.preventDefault()

If you omit `event.preventDefault()`, the form won't work because the page will reload, which will reset the username and password fields.

12. Comment out the `event.preventDefault()`, reload the page and log in again. You will see the page flash as it reloads and the form fields will be reset.

```
    function login(event) {

        // event.preventDefault();
```

END Lesson 03.03

NEXT: Lesson 03.04

UNIT 03

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. Define the function. First thing it does is prevent the form from doing a default page reload:

```
function calcBill(event) {
    event.preventDefault();
}
```

4. Next, get the values from the form.

- Use the Number() method to convert the "number-like strings" to real numbers:

```
function calcBill(event) {
    event.preventDefault();

    let foodTot = Number(food.value);
    let bevTot = Number(bev.value);
    let tipPct = Number(tipMenu.value);
```

```
    let taxPct = Number(taxMenu.value);
    // calc a sub total by adding food and bev
    let subTot = foodTot + bevTot;
}
```

5. Calculate the tip and tax from their percents. To do this, multiply `subTot` by the tip and tax rates, respectively. Divide by 100, because 15% has a numeric value of 0.15.

```
function calcBill(event) {
    event.preventDefault();

    let foodTot = Number(food.value);
    let bevTot = Number(bev.value);
    let tipPct = Number(tipMenu.value);
    let taxPct = Number(taxMenu.value);

    // get a sub total by adding food and bev
    let subTot = foodTot + bevTot;

    // calc the tax and tip from their pcts
    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)`;

```
function calcBill(event) {
    event.preventDefault();

    let foodTot = Number(food.value);
    let bevTot = Number(bev.value);
    let tipPct = Number(tipMenu.value);
    let taxPct = Number(taxMenu.value);

    // get a sub total by adding food and bev
    let subTot = foodTot + bevTot;

    // calc the tax and tip from their pcts
    let taxTot = subTot * taxPct / 100;
    let tipTot = subTot * tipPct / 100;

    // add up the total:
    let total = subTot + taxTot + tipTot;

    // round off tip and total to 2 decimal places
    // do this AFTER all math is done, because toFixed()
    // converts the number to a string
    tipTot = tipTot.toFixed(2); // becomes a string
```

```
        taxTot = taxTot.toFixed(2); // becomes a string
        total = total.toFixed(2);
    }
```

7. Concatenate a bill message and output it. The output includes `<br>` tags, so use `innerHTML` instead of `textContent`:

```
        // round off tip and total to 2 decimal places
        // do this AFTER all math is done, because toFixed()
        // converts the number to a string
        tipTot = tipTot.toFixed(2); // becomes a string
        taxTot = taxTot.toFixed(2); // becomes a string
        total = total.toFixed(2);

        // concat bill
        let bill = `
        ***JS BISTRO***
        <br>***GUEST CHECK***
        <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:
        response.innerHTML = bill;
    }
```

END Lesson 03.04 NEXT: Lesson 03.05

Form Submission

evaluating checkboxes

Preview the lesson html file in the browser. As we see, the form's html and css are all done, but the form itself does not work. This is because we will be writing the JS for the form in this lesson.

The form works as follows:

- The user enters their first name and email into text fields.
- Optionally, the user chooses from the select menu.
- The user can check or uncheck the checkbox to subscribe to the newsletter.
- The user clicks the 'Sock It To Me' button to submit the form.
- The button click calls a function which:
    - gets the values from the name and email from the text input fields.
    - checks to see if the checkbox is checked.
    - ignores the select menu
    - concatenates and outputs an appropriate message.
    - to make room for the message, the submt ubtton is hidden

1. Get the form input elements and have the button listen for a function:

```
let firstName = document.getElementById('firstName');
let email = document.getElementById('email');
let subscribeCheckbox = document.getElementById('subscribe');
let feedback = document.getElementById('feedback');

let btn = document.querySelector('button');
btn.addEventListener('click', processEbookForm);
```

2. Write the function, starting with the prevent default:

```
function processEbookForm(event) {
    event.preventDefault();
}
```

3. Get the inputted text values and concatenate the first part of the response message. Rather than pass the values to local variables, we can just concatenate directly:

```
    function processEbookForm(event) {
        event.preventDefault();
        let msg = `Thank you ${firstName.value}. Check ${email.value} for
your Free eBook.`;
    }
```

4. Evaluate the checkbox by seeing if its checked property returns true. If it is checked, thank the user for subscribing. This gets added to the existing message with += :

```
    function processEbookForm(event) {
        event.preventDefault();
        let msg = `Thank you ${firstName.value}. Check ${email.value} for
your Free eBook.`;
        if(subscribeCheckbox.checked) {
            msg += ' The latest issue of the newsletter will accompany the
email--thanks for subscribing!';
        }
    }
```

5. Hide the btn to make room for the output, and output the message:

```
    function processEbookForm(event) {
        event.preventDefault();
        let msg = `Thank you ${firstName.value}. Check ${email.value} for
your Free eBook.`;
        if(subscribeCheckbox.checked) {
            msg += ' The latest issue of the newsletter will accompany the
email--thanks for subscribing!';
        }
        btn.style.display = 'none';
        feedback.textContent = msg;
    }
```

END Lesson 03.05 NEXT: Lesson 03.06

Hoisting

Function Expressions

Anonymous Functions

hoisting

Hoisting refers to functions being lifted to the top of their scope, so that they are available everywhere in their scope. Since they are hoisted, functions can be called before their declaration appears in the code.

1. Declare a function and call it before and after:

```
console.log(fruit);
let fruit = 'kiwi';
console.log(fruit);

console.log(1, sayHello('Joe')); // Hello, Joe!

function sayHello(name) {
    return `Hello, ${name}!`;
}

console.log(2, sayHello('Joe')); // Hello, Joe!
```

Hoisting does not work for variables, however. To access a variable, it must have already been declared:

2. Try to use a variable before it is declared. You get an error:

```
console.log(`Do you like ${froot}?`);
// ReferenceError: froot is not defined
let froot = 'apples';
console.log(`Do you like ${froot}?`);
// Do you like apples?
```

function experession

A function expression is a variable with a function for a value. As a variable, it doesn't get hoisted. A function expression must be defined *before* it can be called.

3. Write a function expression:

```
const welcomeUser = function(user) {
    return `Welcome ${user}!`;
}
```

4. Call the function above itself. We get an error:

```
console.log(welcomeUser('Jane1'));
// ReferenceError: welcomeUser is not defined

const welcomeUser = function(user) {
    return `*Welcome ${user}!`;
}

console.log(welcomeUser('Jane1'));
```

5. Comment out the error line and call the function down below. It works:

```
// console.log(welcomeUser('Jane123'));
// ReferenceError: welcomeUser is not defined

const welcomeUser = function(user) {
    return **Welcome ${user}**;
}

console.log(welcomeUser('Janet456'));
// Welcome Janet456
```

anonymous functions

An anonymous function is a function that does not have a name. Anytime you see the word function followed by parentheses with no name in between, it's an anonymous function:

```
// named function:
function addNumbers(x, y) {
    return x y;
}

// anonymous function:
function(x, y) {
    return x, y;
}
```

The reason functions have names at all is so they can be called. It stands to reason then that anonymous functions do not need to be called. But if that's true, how do they run?

function set equal to an event property

An anonymous function may be set equal to the event property of an object. When that event happens, the function runs. First let's just look at a function that is called by a listener:

6. Get the BTN 1 button and have it call a function when clicked:

```
    const btn1 = document.getElementById('btn-1');
    btn1.addEventListener('click', doSomething);
```

7. Define the function:

```
    function doSomething() {
        output.textContent = 'You clicked the button! That did
something!';
    }
```

8. Get the element where the function output goes:

```
    const output = document.getElementById('output');
```

9. Reload the page and click the button. We should get the message: *You clicked the button! That did something!*

This review of addEventListener is just to set up the contrast between calling function with a name and running an anonymous function on click.

10. Get BTN 2:

```
    const btn2 = document.getElementById('btn-2');
```

onclick property

Events that can be called on objects co-exist as properties. The 'click' event has its counterpart in the **onclick** property, which is called on the button and set equal to a function. When the click takes place, the function runs:

11. Access the button's onclick property and set it equal to an anonymous function:

```
    btn2.onclick = function() {
        output.textContent = 'Hello from the anonymous function!';
```

```
}
```

For that matter, an anonymous function can be used in a listener, rather than callng a function.

inline anonymous functions

Anonymous functions do not have to come to the right of an equal sign. An anonymous function can be written inline, such as inside the addEventListener method.

13. Get BTN 3 and have it listen for a click. Instead of calling a function by name, as we usually do, just run an inline anonymous function, right there on the spot:

```
const btn3 = document.getElementById('btn-3');

btn3.addEventListener('click', function() {
    output.textContent = 'Hello from inline anonymous function!';
})
```

Let's try one more example of a listener that calls a function by name vs. a listener that runs an anonymous function, inline:

14. Have the body listen for the 'load' event, which fires when the body, is fully loaded, and to call a function when the loading of the page is complete:

```
document.body.addEventListener('load', onBodyLoaded);

function onBodyLoaded() {
    output4.textContent = 'BODY LOADED says onBodyLoaded function';
}
```

15. Refactor the listener to run an inline anonymous function instead of calling a function by name:

```
document.body.addEventListener('load', function() {
    output5.textContent = 'BODY LOADED says inline anonymous
function';
});
```

16. Do away with the listener altogether, and instead use the onload event property to run the anonymous function:

```
document.body.onload = function() {
    output6.textContent = 'BODY LOADED says document.body.onload =
anonymous function';
};
```

keyboard object and events

The pressing of any key is also an event. These are typically also used in conjunction with anonymous functions.

The **document** can listen for a keyboard event and call a function when any key is pressed (**keydown**) or released (**keyup**).

The event properties for the keyboard are **event.key** which is the key that was pressed and **event.keyCode** which is the key's numeric code. The function takes **event** as its argument.

17. Have the document listen for a **keydown** event and call a function when any key is pressed. The function reports the **key** and its **keyCode**.

```
    document.addEventListener('keydown', reportKeyAndCode);

    function reportKeyAndCode(event) {
        output4.textContent = `listener keydown calls reportKeyAndCode
function:\nKey: ${event.key} Code: ${event.keyCode}`;
    }
```

18. Refactor the above to use an inline anonymous function:

```
    document.addEventListener('keydown', function(event) {
        output5.textContent = `listener keydown runs anonymous inline
function:\nKey: ${event.key} Code: ${event.keyCode}`;
    });
```

19. And finally, use the onkeydown property to call an anonymous function:

```
    document.onkeyup = function(event) {
        output6.textContent = `onkeyup = anonymous function \nKey:
${event.key} Code: ${event.keyCode}`;
    };
```

- **END: Lesson 03.06**
- **NEXT: Lesson 03.07**

Apartment Rent Estimator

Apartment Rent Estimator Form

In this **form**, we present the user with two select menus and four "amenities checkboxes" from which they configure their desired apartment. 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:

```
<select name="bdrms" id="bdrms">
  <option value="1000">Studio ($1,000)</option>
  <option value="1400" selected>1 Bedroom ($1,400)</option>
  <option value="1800">2 Bedrooms ($1,800)</option>
  <option value="2300">3 Bedrooms ($2,300)</option>
</select>

<select name="baths" id="baths">
  <option value="0">1 Bath</option>
  <option value="450">1.5 Baths (+$450)</option>
  <option value="850">2 Baths (+$850)</option>
  <option value="1250">2.5 Baths (+$1,250)</option>
</select>
```

checkboxes

The fees associated with the checkboxes will 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 value = 350, for the $350 parking fee
- Fitness Center (gym) value = 100, for the $100 gym fee
- Skyline View value = 0.25, for the 25% surcharge for a skyline view.
- The Doorman value = 0.1, for the 10% surcharge for a doorman building.

```
<input type="checkbox" id="doorman" value="0.1">
<label for="doorman">Doorman Building (+10%)</label><br>

<input type="checkbox" id="parking" value="350">
<label for="parking">Indoor Parking (+$350)</label><br>

<input type="checkbox" id="riverview" value=".25">
<label for="riverview">Overlooking River (+25%)</label><br>

<input type="checkbox" id="gym" value="350"'>
<label for='gym'>Fitness Center (+$100)</label>
```

The html also includes a `calculate rent()` button and an `h3` for posting the answer.

```
<button>calculate rent</button>
<h3></h3>
```

1. Get the button and have it call the function when clicked:

```
const calcBtn = document.querySelector('button');
calcBtn.addEventListener('click', calculateRent);
```

2. Get the "feedback" tag:

```
const feedback = 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) {
  event.preventDefault();
}
```

4. Get the values of the select menus. We don't need the whole objects--just their values:

```
function calculateRent(event) {
  event.preventDefault();

  let bdrms = document.getElementById('bdrms').value;
  let baths = document.getElementById('baths').value;
}
```

5. The `bdrms` and `baths` values come in as strings, so convert them to numbers. Then add them together, saving them to a var called `rent`:

```
  bdrms = Number(bdrms);
  baths = Number(baths);
  let rent = bdrms + baths;
```

6. Get the checkboxes:

```
  let doorman = document.getElementById('doorman'); // +10%
  let parking = document.getElementById('parking'); // +$350
  let view = document.getElementById('view'); // +25%
  let gym = document.getElementById('gym'); // +$100
```

See if each checkbox has its `checked` property set to true. If it returns true, increase the rent accordingly:

- Charge the flat fees first (parking, gym)
- For percent surcharges (doorman and view), multiply the value by the rent and then add that fee to the rent.
- Round off the final rent and output it to the feedback tag.

7. Write if-statements to see if the checkboxex are checked, and if so, increase the rent, accordingly:

```
  if(parking.checked) {
    rent += Number(parking.value);
  }

  if(gym.checked) {
    rent += Number(gym.value);
  }

  if(doorman.checked) {
    rent += rent * Number(doorman.value);
  }

  if(view.checked) {
    rent += rent * Number(view.value);
  }
```

8. Round off rent to an integer, and output the estimated rent:

```
    rent = Math.round(rent);

    feedback.textContent = 'Your estimated rent is: $' + rent;

  } // calculateRent(event)
```

9. Run the page in the browser.

- Choose the most expensive configuration: 3 bedrooms, 2.5 baths, and check all four boxes.
- Follow that by choosing the cheapest configuration (no checkboxes).
- Try various other configurations.

END: Lesson 03.07

NEXT: Lesson 03.08

keydown and keyup events

event.key and event.keyCode properties

set CSS

random hex color

keyboard events

The pressing of a key is a **keydown** event The release of a key is a **keyup** event

Like any event, a keyboard event can call a function. The event target (thing that fires the event) is the document, so the syntax is: **document.addEventListener('keydown', function)** An if-statement can check the **event.keyCode** to know which key was pressed.

The key **event** has **key** and **keyCode** properties:

The q **event.key** is **q**. The q **event.keyCode** is **81**.

The left arrow **event.key** is **ArrowLeft**. The left arrow **event.keyCode** is **37**.

In this activity, we will set up several keys to do different things, most of which involve setting CSS:

random body background color on 'c' keydown

When the **c** or **Enter** key is pressed, our function will set the body background to a random color.

toggle dark mode on 'd' key press

When the **d** key is pressed, our function will toggle the container between dark mode and light mode.

toggle font on 'f' key press

When the **f** key is pressed, our function will toggle the font between *serif* and *sans-serif*.

generate random 4-digit PIN on 'p' keydown

When the **p** or **n** key is pressed, our function will generate and output a random 4-digit PIN number.

move space ship to the left and right

When the left or right arrow is pressed, our function will move the space ship 10 pixels in that direction. The numeric codes for the left and right arrows are **37** and **39**, respectively.

There are four DOM elements we need:

- **container** div that holds instructions and some output divs:
- **rand-pin-box** div for outputting the **key** and **keyCode** of the pressed key
- **key-box** div for outputting hold a random number
- an image of a space ship

1. Get the DOM elments:

```
const container = document.querySelector('.container');
const keyBox = document.getElementById('key-box');
const spaceShip = document.getElementById('space-ship');
const randPinBox = document.getElementById('rand-pin-box');
```

2. Set the left position of the space ship to be approximately in the middle of the screen. Also set a var for the speed of the ship:

```
let leftPos = window.innerWidth / 2 - 150;
spaceShip.style.left = leftPos + 'px';
let shipSpeed = 20;
```

3. Set booleans to keep track of font (serif or not) and dark mode:

```
let serif = true;
let dark = false;
```

4. Have the document listen for **keyup** and call an inline anonymous function which outputs the **key** and its **keyCode**:

```
document.addEventListener('keyup', function(event) {
    keyBox.textContent = `Key: ${event.key}
    Code: ${event.keyCode}`;
});
```

5. Have document listen for **keydown** event and call the **respondToKeyPress**. The function checks if the key is **n**, **p**, **d**, **l**, **c**, **Enter** or the left or right arrow. If one of these keys was pressed, the specified action takes place.

```
document.addEventListener('keydown', respondToKeyPress);
```

6. Define the function **respondToKeyPress**, which runs on **keydown** (press). All functions take **event** object as its default argument, but pass in **event** explicitly, so that we can access its properties:

```
function respondToKeyPress(event) {
}
```

7. Inside the function, set up a scaffoling of a big if-else if block to check for each key: c, d, f, p, Enter (13), Left (37) and Right (39) arrows:

```
function respondToKeyPress(event) {

    if(event.key === 'c' || event.key === 'Enter') {

    } else if(event.key === 'd' ) {

    } else if(event.key === 'f') {

    } else if(event.key === 'p' || event.key === 'n') {

    } else if(event.keyCode === 37) { // ArrowLeft

    } else if(event.keyCode === 39) { // ArrowRight

    }

}
```

Next, we will use conditinal logic to handle what to do (if anything) when a key is pressed:

8. If the **c** key is pressed, call the **makeRandHexColor()** function that generates and returns a random hexadecimal color. Save the color to a variable, and use that to set the background color:

```
function respondToKeyPress(event) {

    if(event.key === 'c'  || event.key === 'Enter') {
        let randHex = makeRandHexColor();
        document.body.style.backgroundColor = randHex;
    }

} // end function
```

9. If the **d** key is pressed, toggle between dark and light mode by adding-removing their styles. Toggle the boolean each time, so that the behavior also toggles.

```
    } else if (event.key === 'd') {
        if(!dark) {
            container.classList.remove('light-mode');
            container.classList.add('dark-mode');
            dark = true;
```

```
            } else {
                container.classList.remove('dark-mode');
                container.classList.add('light-mode');
                dark = false;
            }
        }

    } // end function
```

10. If the **f** key is pressed, toggle between serif and sans-serif font. Toggle the boolean each time, so that the behavior also toggles.

```
        } else if (event.key === 'f') {
            if(serif) {
                document.body.style.fontFamily = "sans-serif";
                serif = false;
            } else {
                document.body.style.fontFamily = "serif";
                serif = true;
            }
        }

    } // end function
```

11. If the **p** or **n** key is pressed, generate a random 4-digit number from 0-9999. If the number is less than 4 digits, add leading zero(es) by means of if-statements with ranges; utput the random pin:

```
        } else if (event.key === 'p' || event.key === 'n') {

            let r = Math.floor(Math.random() * 10000);

            // add leading zero(es), as needed
            if(r < 1000 && r > 99) r = '0' + r; // 3 digits
            if(r < 100 && r > 9) r = '00' + r; // 2 digits
            if(r < 10) r = '000' + r; // 1 digit
            if(r === 0) r = '0000';

            randPinBox.textContent = 'PIN:\n' + r;
        }

    } // end function
```

12. If the **ArrowLeft (37)** key is pressed, and if the space ship is not all the way to the left, move the ship **shipSpeed** pixels to the left:

```
        } else if (event.keyCode === 37) {
            if(leftPos > 0) {
```

```
                leftPos -= shipSpeed;
                spaceShip.style.left = leftPos + 'px';
            }
        }

    } // end function
```

13. If the **ArrowRight (39)** key is pressed, and if the space ship is not all the way to the right, move the ship **shipSpeed** pixels to the right:

```
        } else if (event.keyCode === 39) {
            if(leftPos < window.innerWidth - 300) {
                leftPos += shipSpeed;
                spaceShip.style.left = leftPos + 'px';
            }
        }

    } // end function
```

14. If the spaceship doesn't move, try switching from event.keyCode to event.key:

```
        } else if (event.key === 'ArrowLeft') {
            if(leftPos > 0) {
                leftPos -= shipSpeed;
                spaceShip.style.left = leftPos + 'px';
        } else if (event.key === 'ArrowRight') {
            if(leftPos < window.innerWidth - 300) {
                leftPos += shipSpeed;
                spaceShip.style.left = leftPos + 'px';
            }
        }

    } // end function
```

Random Hex Color

It is possible to generate a random hexadecimal color. Here's the concept and procedure:

- A base-10 number consists of the digits 0-9
- A base-16 string uses the characters 0-9, A-F
- A hexadecimal color is a 6-character, base-16 string
- 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 (256 ** 3) colors in the RGB spectrum
- Calling **toString(16)** on an integer in the RGB range from 0-16777215 returns a hexadecimal color value
- Putting '#' before the hex value comletes the hex color

This function makes and returns a random hex color. It is called from within the **respondToKeyPress** function when the **c** key is pressed.

```
    function makeRandHexColor() {

        // A. Generate a random 16-digit float from 0-1
        let rando = Math.random();

        // B. Multiply the random float by 16,777,216 (RGB range)
        rando = rando * 16777216;

        // C. round down the random number:
        rando = Math.floor(rando);

        // D. convert the random number to a base-16 hexadecimal string
        rando = rando.toString(16);

        // E. add the # to complete the hexadecimal color
        rando = '#' + hex;

        // F. return the hexadecimal color value
        return hex;

    }
```

END: Lesson 03.08

NEXT: Lesson 03.09

Guess Mystery Number

Random Numbers

if-else if-else logic

show-hide buttons

buttons calling functions

Open 03.09-Number-Guessing-Game.html

In this lesson we will write a number guessing program:

- user clicks PLAY button
- button click calls function playGame
- function generates a random integer from 1-100
- input box and GUESS button replace PLAY button
- player enters number from 1-100
- player clicks GUESS button
- button calls function evalGuess
- function gets value from input box and converts it to a real number
- compares user guess number to the random number
- h2 tag of id "feedback" displays all messages
- when user gets number right, a PLAY AGAIN button appears.

First we need a few global variables. They need to be global, because they must be available to more than one function:

Declare randNum to hold the random number; playerGuess to store the player's number; and guessesLeft to count down from 7.

```
let randNum = 0;
let playerGuess = 0;
let guessesLeft = 7;
```

Next, get the DOM elements used for the game. There are two buttons, but the GUESS button is hidden on page load, so all we see is the PLAY button.

The input box for entering the guess number is also hidden on page load. Both hidden elements will appear when the user clicks PLAY button:

```
let playBtn = document.getElementById('play-btn');

// hide on page load, show on PLAY click
let guessBtn = document.getElementById('guess-btn');
```

Have each button listen for its respective function to call when clicked.

```
playBtn.addEventListener('click', playGame);

guessBtn.addEventListener('click', evalGuess);
```

Also get the "guess box" (number input box), which is also hidden on page load. The hidden states are handled in the CSS, with the display:none property.

```
// hide on page load, show on PLAY click
let guessBox = document.querySelector('input');
```

Get the h2 that displays the feedback:

```
let feedback = document.getElementById('feedback');
```

Now for the functions. Declare the playGame function, which runs when the player clicks the PLAY button. The function does the following:

- generates the random number from 1-100
- hides "this", which is itself, the PLAY button
- shows (display = "block") the guess box (number input field) and the GUESS button.
- sets the input box to an initial value of 0, so the user knows to click there and enter a number.
- gives the user some feedback: 'Guess the number!'

```
function playGame() {
    randNum = Math.ceil(Math.random() * 100);
    this.style.display = 'none';
    guessBox.style.display = 'block';
    guessBtn.style.display = 'block';
    guessBox.value = 0;
    feedback.textContent = 'Guess the mystery number from 1-100!';
}
```

The evalGuess function runs when the user clicks the GUESS button. This function gets the value from the input box and converts it to a real number using the `Number()` method.

```
    function evalGuess() {
        playerGuess = Number(guessBox.value);
    }
```

Now for the big if-else if-else block:

- if the guess is too low, user gets feedback (too LOW) and guesses again
- else if the guess is too high, user gets feedback (too HIGH) and guesses again
- else guess matches mystery number, so user is congratulated
- GUESS button is replaced by PLAY AGAIN button
- user gets 7 guesses, which count down and are displayed with each guess
- when the user gets it right, the resetGame() function is called
- there is no enforcement of guess limit. Game just keeps going after 7 guesses.

```
    function evalGuess() {
        playerGuess = Number(guessBox.value);

        if (playerGuess < randNum) {
            guessesLeft--;
            feedback.innerHTML = '<p>Your guess is too low! <br>You have '
+ guessesLeft + ' guesses left!</p>';
        } else if (playerGuess > randNum) {
            guessesLeft--;
            feedback.innerHTML = '<p>Your guess is too high! <br>You have
' + guessesLeft + ' guesses left!</p>';
        } else {
            guessesLeft--;
            feedback.innerHTML = '<p>Congrats!<br>The mystery number is
indeed ' + randNum + '! <br>You got it in ' + (7 - guessesLeft) + '
guesses!</p>';
            resetGame();
        }
    }
```

The resetGame function is called when the user guesses correctly. The function restores all the settings as they are on page load. The only difference is that the PLAY button now says PLAY AGAIN (it is the same button, with different text).

```
    function resetGame() {
        guessesLeft = 7;
        guessBtn.style.display = 'none';
        guessBox.style.display = 'none';
        playBtn.style.display = 'block';
        playBtn.textContent = 'PLAY AGAIN';
        guessBox.value = 0;
    }
```

END: Lesson 03.09

NEXT: Lesson 04.01

arrays

const

index, array.length

methods: push(), sort(), pop(),

nested / 2D / matrix 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: `[ 'hola', 38, true, ['apple, 'banana'] ]`

using const to declare an array

As detailed in a previous lesson, primitive variables (strings, numbers, booleans) declared with **const** cannot be modified (changed) in any way.

1. As a recap, declare a constant with `const`, and try to change it. Cannot be done:

```
const LITERS_PER_GALLON = 3.78;
LITERS_PER_GALLON = 4; // ERROR: Assigment to constant variable
```

const for objects

However, arrays declared with `const` *can* be modified. Items can be changed, added or deleted. But the array object itself cannot be *mutated* to a different datatype. So, with *const*, once an array, always an array.

how to declare an array

To declare an array, start with the `let` or `const` keyword, followed by a name. We will start with **let** in order to later prove our point about mutability. The array is wrapped in square brackets, with items separated by a comma.

It is a good practice to pluralize array names so: *fruits* not *fruit*. You can also add *Arr* to make it crystal clear 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, so if there are 3 items in an array, the last item is at index 2.
- Use square bracket syntax to access 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 a value after the last item:

6. Add 'mango' 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

One 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 an accident from occurring.

7. Change *fruitsArr* to a string. Just like that, no more array:

```
fruitsArr = "fresh";
console.log(fruitsArr); // fresh
```

8. Declare another fruits array, this time with `const`. We need a new name, as `const` cannot redeclare an existing variable:

```
const fruits = ['apricot', 'pear', 'kiwi', 'grape'];
```

9. Try to mutate this `const` array into a string. You get an error:

```
fruits = "ripe";
// ERROR: Assignment to constant variable
```

array.length property

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[arrray.length-1]

- The first item in an array is at: **array[0]**.
- The last item is at: **array[array.length-1]**.

11. Use `length-1` to access the last item in the array:

```
let lastFruit = fruits[fruits.length - 1];
console.log(lastFruit); // grape
```

random array items

To get a random item from an array, generate a random integer within the range of the array length, and pass that number to the array square brackets:

12. Get a random fruit from the *fruits* array.

```
let randInt = Math.floor(Math.random() * fruits.length);
let randFru = fruits[randInt];
```

```
console.log(randFru);
```

13. 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)

We used `fruits[3]` to add to the end of the array. But what if the length of the array is unknown? Better is to use the `push()` method, which adds an item to the end without needing to know how many items are in the array.

14. Using the **push()** method, add 'lime' to the end of the array.

```
fruits.push('lime');
console.log(fruits);
// ['apricot', 'pear', 'kiwi', 'grape', 'lime']
```

declaring a new empty array

15. Use the **sort()** method to put the string items of an array in alphabetical order:

```
fruits.sort();
console.log(fruits);
// ['apricot', 'grape', 'kiwi', 'lime', 'pear']
```

array.pop()

The **pop()** method removes the last item from the array and returns it, so the item can be saved by setting `pop()` equal to a variable.

16. Using the **pop()** method, remove the last item and save it to a variable, `popped`:

```
let popped = fruits.pop();
console.log(popped); // pear
console.log(fruits);
// ['apricot', 'grape', 'kiwi', 'lime']
```

An array can be declared with just a pair of empty curly braces--no items.

17. Declare a new, empty array as a pair of square brackets:

```
const veggies = [];
```

18. Use the push() method to populate the array:

```
veggies.push('carrot');
veggies.push('celery');
console.log(veggies);
// ['carrot', 'celery']
```

2D maxtrix arrays

An array can have arrays for its items. The terms to describe such an array include: **matrix, 2D array** and **nested array**

19. Make a 3x3 matrix of three items, each an array of three items.

```
const nestedArr = [[1,2,3], [4,5,6], [7,8,9]];
console.log(nestedArr.length);
console.log(nestedArr);
console.log(nestedArr[0]); // [1,2,3]
```

20. Use double square brackets: the first to get the inner array, the second to get the inner array item:

```
console.log(nestedArr[0][0]); // 1
console.log(nestedArr[1][1]); // 5
console.log(nestedArr[2][2]); // 9
```

21. Represent a tic-tac-toe board, where all 9 squares start out with a value of null:

```
let ticTacToe = [
    [null, null, null],
    [null, null, null],
    [null, null, null]
];
```

22. Then the game starts; "X" chooses the middle square, and "O" chooses the lower left square:

```
ticTacToe[1][1] = "X";
ticTacToe[2][0] = "O";
```

23. Log the result: it's a tic-tac-toe game in 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]
*/
```

END: Lesson 04.01

NEXT: 04.01 Lab Lesson 04.02

1. Declare an array called gems and give it three items: 'diamond', 'sapphire' and 'ruby'.
2. Log the entire array and also the first item.
3. Add 'emerald' to the end of the array, using the index of the new item
4. Use an array method to add 'amethyst' to the end of the array.
5. Use an array method to sort the array.
6. Log the length (number of items of the array). Should be 5.
7. Use length-1 to get the last item in the array. Save it to a variable and log it.
8. Use an array method to remove the last item from the array. Save it to a new variable. HINT: the popped item should be 'sapphire'.
9. Given these 12 months: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec. Make a 2D array called months, containing the 12 items, divided into 4 sub-arrays or 3 items each, so: Jan, Feb, Mar would be the first array. Log the months that start with 'J' and 'M'.
10. Make four arrays called winter, spring, summer and autumn, each of which contains three items, assigned as one item of the months array. Log random month from each array.

SEE 04.01 Lab Solution

04.01 Lab Solutions

1. Declare an array called gems and give it three items: 'diamond', 'sapphire' and 'ruby'.

```
const gems = ['diamond', 'sapphire', 'ruby'];
```

2. Log the entire array and also the first item.

```
console.log(gems);
// ['diamond', 'sapphire', 'ruby']
console.log(gems[0]); // diamond
```

3. Add 'emerald' to the end of the array, using the index of the new item.

```
gems[3] = 'emerald';
console.log(gems);
// ['diamond', 'sapphire', 'ruby', 'emerald']
```

4. Use an array method to add 'amethyst' to the end of the array.

```
gems.push('amethyst');
console.log(gems);
// ['diamond', 'sapphire', 'ruby', 'emerald', 'amethyst']
```

5. Use an array method to sort the array. gems.sort();

```
console.log(gems);
// ['amethyst', 'diamond', 'emerald', 'ruby', 'sapphire']
```

6. Log the length (number of items of the array). Should be 5.

```
console.log(gems.length); // 5
```

7. Use length-1 to get the last item in the array. Save it to a variable and log it.

```
let lastGem = gems[gems.length-1];
console.log(lastGem); // sapphire
```

8. Use an array method to remove the last item from the array. Save it to a new variable. HINT: the popped item should be 'sapphire'.

```
let poppedGem = gems.pop();
console.log(poppedGem); // sapphire
console.log(gems);
// ['amethyst', 'diamond', 'emerald', 'ruby']
console.log(gems.length); // 4
```

9. Given these 12 months: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec. Make a 2D array called months, containing the 12 items, divided into 4 sub-arrays or 3 items each, so: Jan, Feb, Mar would be the first array. Log the 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. Make four arrays called winter, spring, summer and autumn, each of which contains three items, assigned as one item of the months array. Log random month from each array.

```
const winter = months[0];
let randWinterMo = Math.floor(Math.random() * winter.length);
console.log(randWinterMo);

const spring = months[1];
let randSpringMo = Math.floor(Math.random() * spring.length);
console.log(randSpringMo);

const summer = months[2];
let randSummerMo = Math.floor(Math.random() * summer.length);
console.log(randSummerMo);

const autumn = months[3];
let randAutumnMo = Math.floor(Math.random() * autumn.length);
console.log(randAutumnMo);
```

NEXT: Lesson 04.02

Objects

object

An object is a variable that stores a collection of properties bundled in curly braces {}.

properties

- Properties are variables belonging to an object; as such, they have a name and a value.
- Properties are child variables of the parent object variable.

key-value pairs

- Properties are name-value pairs, called **key-value** pairs.
- A property name is called a **key**. Unlike "regular" variable names, keys can have spaces.
- The value can be of any data type: string, number, boolean--even object and function.

method

- An object property whose value is a function is called a **method**.

const for objects

As with arrays, we will usually declare with `const`. This gives us the freedom to modify the properties at will, while keeping us in our swim lane, so to speak, by preventing us from changing the data type.

difference between object and array

Arrays have a data type of object, but array items have no name-value pairs, but are rather numbered by position (index), and are bundled in square brackets.

DOM objects

When a web page (DOM) element is brought into JS, it exists in the script as an **object**. The properties of the object are the attributes of the elements. We have already been working with the DOM by getting elements and setting their properties.

1. Open **04.02-Objects.html** and preview it in the browser. This page is just a springboard for checking our console.log output. Remember to switch the file being used from FINAL .js to START .js.

2. In the JS file, declare a new object variable, and assign it some properties in between its curly braces. The key-values are separated by colons. Each property ends with a comma to separate it from the next property. Optionally, a so-called **trailing comma** may come after the last property.

```
const car = {
    make: 'Ford',
    model: 'Mustang GT',
    year: 1999,
    color: 'red',
    condition: 'excellent',
    miles: 123456,
    onRoad: true,
    forSale: false
};
```

Properties of an object are available only to the object, and are referenced by dot-sytnax: `object.property`.

3. Log the whole object, as well as a few properties. In the console, open the arrow to see the properties. Notice that, in contrast to array items, properties are not numbered:

```
console.log(car);
console.log(car.year, car.make, car.model);
```

Properties are added to an existing object by dot syntax: `object.property = value`. Adding a property amounts to declaring a variable *scoped* to the object.

nested objects: objects as object properties

Object properties can be other objects, including arrays.

4. Add two more properties to the car object, one an array, the other a child object: and then access them using "dot.dot" and square bracket syntax

```
car.mpg = {city: 18, hwy: 25};
car.options = ['sun roof', 'CD player', 'leather seats'];

console.log(car);
```

5. Access the new properties, using "dot.dot" for the child object and square brackets for array items:

```
console.log('MPG City:', car.mpg.city);
console.log('Audio:', car.options[1]);
```

Set (change) a property value with dot-syntax in the same way you would add a property:

6. Update some of the property values of the car object.

```
car.miles = 135790;
car.condition = "very good";
car.mpg.city = 17;
car.mpg.hwy = 24;
car.options[0] = "moon roof";
car.forSale = true;

console.log(car);
```

consolidating related properties into child objects

7. Add three properties having to do with the engine:

```
car.cylinders = 8;
car.liters = 4.6;
car.horsepower = 260;
```

Since `cyl` (cylinders), `hp` (horsepower) and `ltr` (liters) all have to do with the engine, we can bundle these into an `engine` property.

8. Retool the object by "bundling" `cyl`, `hp` and `ltr` into an `engine` property. This also lets us abbreviate propeterties without obscuring the meaning:

```
car.engine = { cyl: 8, ltr: 4.5, hp: 270 };
```

9. Log both versions of the duplicate engine properties:

```
console.log(car.horsepower, car.engine.hp); // 260 260
console.log(car.cylinders, car.engine.cyl); // 8 8
console.log(car.liters, car.engine.L); // 4.6 4.6
```

delete keyword

The `delete` keyword is used to remove object properties, as `detete object.propery`.

10. Delete `horsepower`, `cylinders` and `liters` and then log `car` to make sure they've been deleted:

```
delete car.horsepower;
delete car.cylinders;
delete car.liters;

console.log(car);
```

keys can have spaces

While there is generally no upside to having spaces in keys, it is allowed. The key goes in quotes and is accessed with square brackets: `object[key]`.

11. Add two properties with spaces in the keys. Log the `car` object to confirm that they got added:

```
car["consumer reviews"] = 234;
car["star rating"] = 4.7;

console.log(car);
```

toLocaleString()

The `number.toLocaleString()` method is called on a number and returns the number with commas, which converts it to a string.

12. Convert a number with no commas to a "number-like string" with commas:

```
let price = 21500;
let priceStr = price.toLocaleString();
console.log(priceStr, typeof(priceStr));
```

object methods

- When a property value is a function, that property is known as a **method**.
- A method must `return` a value
- A method is called using dot syntax: `object.method()`
- A method can make use of the object's properties, accessing them by referring to the object itself as `this`.

this keyword

The `this` keyword refers to different objects, depending on the context:

- in the global scope, `this` is the Global Window Object
- inside a function, `this` is the object that "owns" the event that calls the function; in the case of a function called by a button click, `this` is the button.
- inside a method, `this` is the object itself.

13. Define a method called `listForSale`. Have it return a "FOR SALE" listing. Refer to various properties using the `this` keyword:

```
car.listForSale = function() {
    return `<strong>FOR SALE!</strong>
    ${this.year} ${this.make} ${this.model},
    only ${car.miles.toLocaleString('en-us')} miles,
    ${this.condition} condition.
```

```
        Loaded with options:<br>${this.options[0]}, ${this.options[1]},
${this.options[2]} and much more.
        MPG: ${this.mpg.hwy} highway, ${this.mpg.city} city.
        Price negotiable.`;
    }
```

13. Call the method in two ways; directly log it and also save it to a variable, and then log the variable:

```
    console.log(car.listForSale());
    let listing = car.listForSale();
    console.log(listing);
```

DOM output of object properties as "Details Page"

In a search application, that page that displays the search results is the "results page". When you click on an individual result, you go to the "details page" where you see more info about that item. Let's make a "details page" for the car, as if it were a search result.

The html and css for this are already done:

```
    <div>
        <img src="images/1999-Ford-Mustang-GT.jpg" id="car-pic">
        <h2 id="car-title">year make model</h2>
        <h3 id="car-mpg">hwy city</h3>
        <h3 id="car-engine">engine L cyl hp</h2>
        <h3 id="car-options">options</h2>
        <hr>
        <h3 id="car-reviews-ratings">reviews ratings</h2>
        <hr>
        <p id="car-listing">listing</p>
    </div>
```

14. Get the elements for displaying the car data:

```
    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 carReviewsRatings = document.getElementById('car-reviews-
ratings');
    const carPic = document.getElementById('car-pic');
```

There's no button or menu for displaying the car data; it just displays automatically on page load. But still we can have the document call a function when it loads.

15. Have the body `onload` event call an anonymous function when the document is fully loaded:

```
    document.body.onload = function() {
    }
```

16. Output the object properties to the webpage:

```
    document.body.onload = function() {

        carTitle.textContent = `${car.year} ${car.make} ${car.model}`;

        carMPG.textContent = `MPG: ${car.mpg.hwy} hwy - ${car.mpg.city}
city`;

        carEngine.textContent = `Engine: V${car.engine.cyl}
${car.engine.L} L - ${car.engine.hp} horsepower`;

        carOptions.textContent = `Options: ${car.options[0]},
${car.options[1]}, ${car.options[2]}`;

        carReviewsRatings.textContent = `${car['consumer reviews']}
customer reviews - ${car['stars rating']} stars`;

        carListing.innerHTML = car.listForSale();
    }
```

looking up object property by dynamic key

A dynamic key is a variable serving as an object key. Looking up items by dynamic key requires square bracket -- not dot -- syntax.

Next we'll work with an `animals` object with numerous properties, each an individual animal.

17. Open `animals.js` and have a look:

```
    const animals = {

        'American bison': { class: 'mammal', herbivore: true, continent:
'North America' },

        anaconda: { class: 'reptile', herbivore: false, continent: 'South
America' },

        // -- ETC. --
    }
```

- It is an object called `animals` with 18 properties, each an individual animal.

- Each `key` is an animal name, with the two-word keys in quotes.
- Each `value` is an object with four properties:
    - class (string)
    - herbivore (boolean)
    - continent (string)
    - legs (number)

We will write a program where the user chooses an animal from the menu to load its info:

- The user chooses an animal from a menu
- The change event calls a function
- The function gets the choice and saves it to a variable, `chosenAnimal`. The choice is the name of an animal (e.g. 'panda', 'lion')
- The function looks up that animal by dynamic key: `animals[chosenAnimal]`
- The animal data is outputted to the page

The html and css for this are already done, with zebra displayed by default. Both JS files: `animals.js` and `04.02-Objects-FINAL.js` are already imported into the html file.

```
<select id="animals-menu">
    <option id="choose">Choose an animal..</option>
    <option id="American bison">American bison</option>
    -- ETC. --
</select>

<div id="animal">
    <img src="images/zebra.jpg" id="animal-pic">
    <div>
        <p id="animal-name">zebra</p>
        <p id="animal-class">class: mammal</p>
        <p id="continent">continent: Africa</p>
        <p id="herbivore">herbivore: yes</p>
        <p>
    </div>
</div>
```

18. Get the elements for displaying the animal data:

```
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');
```

19. Have the menu call a function on 'change' (menu choice):

```
animalsMenu.addEventListener('change', displayAnimalInfo);
```

20. Write the function, starting with saving the menu choice to a variable:

```
function displayAnimalInfo() {

    // get the menu choice, which is an animal name
    let chosenAnimal = animalsMenu.value; // e.g. giraffe, panda
}
```

21. Look up the animal in the `animals` object, using the variable as the key. For this dynamic property accessor, use square brackets, not dot-syntax:

```
let animalObj = animals[chosenAnimal];
```

22. Output the animal properties to their respective tags:

```
animalName.textContent = chosenAnimal;
animalClass.textContent = 'Class: ' + animalObj.class;
continent.textContent = 'Continent: ' + animalObj.continent;
herbivore.textContent = 'Herbivore: ' + animalObj.herbivore;
```

23. Set the souce of the animal image. That completes the function:

```
    animalPic.src = `images/${chosenAnimal}.jpg`;

} // end function
```

END: Lesson 04.02 NEXT: 04.02 LAB

Making an Object with Properties

Adding a Method (function) to an Object

Updating / Setting Object Properties

---

1. Make an object variable called product. - In the object variable declaration, include the following properties of indicated key and datatype. - For the values, refer to the method's return value

- product object's properties: key (datatype)
  - prodName (string)
  - patentNum (number)
  - assemblyRequired (boolean)
  - buzzwords (array with 3 items)
  - company (object with 2 properties: name, country
  - promote (method) the returns a string concatenated from ALL the other properties and array items: promote method returns: "Announcing Doo-Dad Incorporated's newest high-tech, labor-saving, water-resistant Wonder Widget! Made in U.S.A. Patent number: 7654321. Some assembly required." HINT: Use if-else logic in the method to concatnate or not based on the boolean

```
const product = {
    // your code
}
```

2. Log the following properties of the object:

   - the whole object
   - the boolean, including with it the label "Assembly Required:"
   - the first item in the buzzwords array
   - the last item in buzzwords, accessed via the array length
   - the company name and country as: "All Doo-Dad, Inc. products are made in U.S.A."
   - the promote method: Get Doo-Dad Inc.'s new high-tech, labor-saving, water-resistant Wonder Widget! Made in U.S.A. Patent number: 7654321. Some assembly required.

```
console.log(product); // {} (contains all properties)
console.log(product); // Assembly Required: true
console.log(product); // high-tech
```

```
    console.log(product); // water-resistant
    console.log(product); // All Doo-Dad, Inc. products are made in U.S.A.
    console.log(product); // Get Doo-Dad Inc.'s new high-tech, labor-
saving, water-resistant Wonder Widget!
    // Made in U.S.A. Patent number: 7654321. Some assembly required.
```

3. Add four new properties outside of the object. In other words, do not add these new properites directly into the object declaration:
    - msrp (number) -- MSRP (Manufacturer's Suggested Retail Price)
    - discount (number) -- as decimal: .2 means 20%
    - salePrice (result of math done with mspr and discount)
    - caution (string)
4. Update the promote method to include the new properties.
    - do not edit the original object method HINT: product.promote = function()
    - The promote method's updated return value should be: "Get Doo-Dad Inc.'s new high-tech, labor-saving, water-resistant Wonder Widget! Made in U.S.A. Made in U.S.A. Patent number: 7654321. Some assembly required. MSRP: $40 * * * ON SALE! 20% OFF * * * NOW ONLY: $32 *** (CAUTION: Water-resistant does NOT mean waterproof. Do not submerge.)
    - log the promote method to see the changes.

```
    product.promote = function() {
    }

    console.log(product.promote());
    // Get Doo-Dad Inc.'s new high-tech, labor-saving, water-resistant
Wonder Widget!
    // Made in U.S.A. Patent number: 7654321. Some assembly required.
    // *** MSRP: $40 *** ON SALE! $20% OFF *** NOW ONLY: $32;
```

5. Update / change properties and log the whole object and the method again:
    - flip the boolean to false
    - increase the MSRP to $75
    - increase the discount to 50%
    - push another buzzword into the array: "shock-resistant"
    - update the promote method to include the new array item
    - log the promote method to see the changes.

```
    product.promote = function() {
    }

    console.log(product.promote());
    // Get Doo-Dad Inc.'s new high-tech, labor-saving, water-resistant,
shock-resistant Wonder Widget!
```

```
    // Made in U.S.A. Patent number: 7654321. Some assembly required.
    // *** MSRP: $75 *** ON SALE! $50% OFF *** NOW ONLY: $37.50;
```

SEE Lab 04.02 Solution

04.02 Lab Solution

Making an Object with Properties

Adding a Method (function) to an Object

Updating / Setting Object Properties

---

1. Make an object variable called product.

- In the object variable declaration, include the following properties of indicated key and datatype.
- For the values, refer to the method's return value
- product object's properties: key (datatype)
    - prodName (string)
    - patentNum (number)
    - assemblyRequired (boolean)
    - buzzwords (array with 3 items)
    - company (object with 2 properties: name, country
    - promote (method) the returns a string concatenated from ALL the other properties and array items: promote method returns: "Announcing Doo-Dad Incorporated's newest high-tech, labor-saving, water-resistant Wonder Widget! Made in U.S.A. Patent number: 7654321. Some assembly required." HINT: Use if-else logic in the method to concatnate or not based on the boolean

```
    const product = {
        prodName: 'Wonder Widget',
        patentNum: 7654321,
        assemblyRequired: true,
        buzzwords: ['high-tech', 'labor-saving', 'water-resistant'],
        company: {name: "Doo-Dad, Inc.", country: "U.S.A."},
        promote: function() {
            let assembly = "";
            if(this.assemblyRequired) assembly = "Some assembly
required.";
            return `\Get ${this.company.name}'s new ${this.buzzwords[0]},
${this.buzzwords[1]}, ${this.buzzwords[this.buzzwords.length-1]}
${this.prodName}!
            \nMade in ${this.company.country}. Patent number:
${this.patentNum}. ${assembly}`
        }
    }
```

2. Log the following properties of the object:

- the whole object
- the boolean, including with it the label "Assembly Required:"
- the first item in the buzzwords array
- the last item in buzzwords, accessed via the array length
- the company name and country as: "All Doo-Dad, Inc. products are made in U.S.A."
- the promote method: Get Doo-Dad Inc.'s new high-tech, labor-saving, water-resistant Wonder Widget! Made in U.S.A. Patent number: 7654321. Some assembly required.

```
    console.log(product); // {} (click arrow to open and see all
properties)
    console.log("Assembly Required:", product.assemblyRequired); //
Assembly Required: true
    console.log(product.buzzwords[0]); // high-tech
    console.log(product.buzzwords[product.buzzwords.length-1]); // water-
resistant
    console.log(`All ${product.company.name} products are made in
${product.company.country}`);
    // All Doo-Dad, Inc. products are made in U.S.A.
    console.log(product.promote());
    // Get Doo-Dad Inc.'s new high-tech, labor-saving, water-resistant
Wonder Widget!
    // Made in U.S.A. Patent number: 7654321. Some assembly required.
```

3. Add four new properties outside of the object. In other words, do not add these new properites directly into the object declaration:
    - msrp (number) -- MSRP (Manufacturer's Suggested Retail Price)
    - discount (number) -- as decimal: .2 means 20%
    - salePrice (result of math done with mspr and discount)
    - caution (string)

```
    product.msrp = 40;
    product.discount = .2;
    product.salePrice = product.msrp * (1 - product.discount);
    product.caution = "Water-resistant does NOT mean waterproof. Do not
submerge.";
```

4. Update the promote method to include the new properties.
    - do not edit the original object method HINT: product.promote = function()
    - The promote method's updated return value should be: "Get Doo-Dad Inc.'s new high-tech, labor-saving, water-resistant Wonder Widget! Made in U.S.A. Made in U.S.A. Patent number: 7654321. Some assembly required. MSRP: $40 * * * ON SALE! 20% OFF * * * NOW ONLY: $32 *** (CAUTION: Water-resistant does NOT mean waterproof. Do not submerge.)
    - log the promote method to see the changes.

```
    product.promote = function() {
        let assembly = "";
```

```
        if(this.assemblyRequired) assembly = "Some assembly required";
        return `Get ${this.company.name}'s new ${this.buzzwords[0]},
        ${this.buzzwords[1]}, ${this.buzzwords[this.buzzwords.length-1]}
${this.prodName}!
        Made in ${this.company.country}. Patent number: ${this.patentNum}.
${assembly}
        *** MSRP: $${this.msrp} *** ON SALE! ${this.discount * 100}% OFF
*** NOW ONLY: $${this.salePrice}`;
    }

    console.log(product.promote());
    // Get Doo-Dad Inc.'s new high-tech, labor-saving, water-resistant
Wonder Widget!
    // Made in U.S.A. Patent number: 7654321. Some assembly required.
    // *** MSRP: $40 *** ON SALE! $20% OFF *** NOW ONLY: $32;
```

5. Update / change properties and log the whole object and the method again:
    - flip the boolean to false
    - increase the MSRP to $75
    - increase the discount to 50%
    - push another buzzword into the array: "shock-resistant"
    - update the promote method to include the new array item
    - log the promote method to see the changes.

```
    product.assemblyRequired = false;
    product.msrp = 75;
    product.discount = .5;
    product.buzzwords.push('shock-resistant');

    product.promote = function() {
        let assembly = "";
        if(this.assemblyRequired) assembly = "Some assembly required";
        return `Get ${this.company.name}'s new ${this.buzzwords[0]},
${this.buzzwords[1]},
        ${this.buzzwords[2]}, ${this.buzzwords[this.buzzwords.length-1]}
${this.prodName}!
        Made in ${this.company.country}. Patent number: ${this.patentNum}.
${assembly}
        *** MSRP: $${this.msrp} *** ON SALE! ${this.discount * 100}% OFF
*** NOW ONLY: $${this.salePrice}`;
    }

    console.log(product.promote());
    // Get Doo-Dad Inc.'s new high-tech, labor-saving, water-resistant,
shock-resistant Wonder Widget!
    // Made in U.S.A. Patent number: 7654321. Some assembly required.
    // *** MSRP: $75 *** ON SALE! $50% OFF *** NOW ONLY: $37.50;
```

NEXT: Lesson 04.03

Array Methods

push(), pop(), sort()

unshift(), shift(), concat()

splice(), slice(), includes()

indexOf(), lastIndexOf()

join(), flat(), reverse()

Array methods are called on arrays and perform operations. We looked at a few array methods in a previous lesson, namely: `push()`, `pop()` and `sort()`. Let's recap those three before moving on:

1. Declare an array of a few items, and use `push()` to add a couple of items at the end:

```
const fruits = ['kiwi', 'cherry', 'banana'];
fruits.push('orange');
fruits.push('grape');

console.log(fruits);
// ['kiwi', 'cherry', 'banana', 'orange', 'grape']
```

2. Remove the last item using `pop()`. The method returns the popped item, so save that to a variable:

```
let poppedItem = fruits.pop();
console.log(poppedItem); // grape

console.log(fruits);
// ['kiwi', 'cherry', 'banana', 'orange']
```

3. Arrange the items in alphabetical order with `sort()`:

```
fruits.sort();

console.log(fruits); // ['banana', 'cherry', 'kiwi', 'orange']
```

Now, for some more array methods:

- **unshift()** -- add item to beginning
- **shift()** -- remove item from beginning
- **concat()** -- combine two or more arrays
- **splice()** -- remove or swap out items
- **slice()** -- make 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
- **reverse()** -- reverse the order of items

shift() and unshift()

- **unshift()** adds an item to the beginning of an array.
- **shift()** removes the first item and returns it.

To help remember which is which, "unshift" is a longer word than "shift", just as "push" is longer than "pop". The longer words make the array longer, while the shorter words make the array shorter.

4. Use **unshift()** to add an item to the beginning of the fruits array:

```
fruits.unshift('apple');

console.log(fruits);
// ['apple', 'banana', 'cherry', 'kiwi', 'orange']
```

5. Use **shift()** to remove and return the first item, saving that to a variable:

```
let shiftedItem = fruits.shift();

console.log(shiftedItem); // apple
console.log(fruits);
// ['banana', 'cherry', 'kiwi', 'orange']
```

concat()

As the name implies, **concat()** concatentates (combines) two or more things, in this case arrays. You call **concat()** on one array, and pass the method the other array(s) as its argument(s).

6. Declare three arrays, and then concat them into one:

```
const tropicalFruits = ['mango', 'kiwi', 'banana', 'pineappple'];

const citrusFruits = ['orange', 'lemon', 'lime', 'tangerine'];
```

```
    const blossomFruits = ['apple', 'peach', 'cherry', 'plum'];

    const fruitCocktailArr = tropicalFruits.concat(citrusFruits,
blossomFruits);

    console.log(fruitCocktailArr);
    // ['mango', 'kiwi', 'banana', 'pineappple', 'orange', 'lemon',
'lime', 'tangerine', 'apple', 'peach', 'cherry', 'plum']
```

7. Sort the fruit cocktail array. The `sort()` method changes the original array; it does not return a new array.

```
    fruitCocktailArr.sort();

    console.log(fruitCocktailArr);
    // ['apple', 'banana', 'cherry', 'kiwi', 'lemon', 'lime', 'mango',
'orange', 'peach', 'pineappple', 'plum', 'tangerine']
```

splice()

The **splice()** method removes or swaps out items at a specified index or range of indices. With **splice()**, you can remove or replace any item, or sequence of items, anywhere in the array.

- `splice()` takes two arguments: the index of the first item to remove, and the number of items to remove.
- `splice()` returns the removed item(s). If more than one item was removed, it returns an array.

8. Remove the item at index 9, which is 'pineapple':

```
    fruitCocktailArr.splice(9,1);

    console.log(fruitCocktailArr);
    // ['apple', 'banana', 'cherry', 'kiwi', 'lemon', 'lime', 'mango',
'orange', 'peach', 'plum', 'tangerine']
```

9. Starting at index 2, splice out four consecutive items, saving the result to a new array:

```
    let splicedItems = fruitCocktailArr.splice(2,4);

    console.log(splicedItems);
    // ['cherry', 'kiwi', 'lemon', 'lime']
    console.log(fruitCocktailArr);
    // ['apple', 'banana', 'mango', 'orange', 'peach', 'plum',
'tangerine']
```

To swap an item with `splice()`, pass in the new item as additional argument(s).

10. Using `splice()`, swap 'peach' for 'papaya':

```
    fruitCocktailArr.splice(4, 1, 'papaya');

    console.log(fruitCocktailArr);
    // ['apple', 'banana', 'mango', 'orange', 'papaya', 'plum',
'tangerine']
```

To swap more than one item with `splice()`, just add more arguments.

11. Replace 'banana' and 'mango' with 'blueberry' and 'strawberry'.

```
    fruitCocktailArr.splice(1, 2, 'blueberry', 'strawberry');

    console.log(fruitCocktailArr);
    // ['apple', 'blueberry', 'strawberry', 'orange', 'papaya', 'plum',
'tangerine']
```

splice for avoiding repeats of random array items

If you keep choosing items at random from an array, you will eventually get repeats. To avoid repeats, splice out each item as you go.

12. Pick two random fruits from `fruitCocktailArr`, and log them both:

```
    let r1 = Math.floor(Math.random() * fruitCocktailArr.length);
    let rFruit1 = fruitCocktailArr[r1];

    let r2 = Math.floor(Math.random() * fruitCocktailArr.length);
    let rFruit2 = fruitCocktailArr[r2];

    console.log(rFruit1, rFruit2);
```

13. Rerun the console.log until you get a repeat fruit. To rerun the console command, hit the Up Arrow and then hit Enter.

To avoid repeats, each time a random fruit is chosen, splice it out of the array with `splice(r, 1)`. The index, `r`, is the random item and the number of items to remove is 1.

14. Choose two random fruits again, this time splicing out the first fruit before picking the second one:

```
    r1 = Math.floor(Math.random() * fruitCocktailArr.length);
    rFruit1 = fruitCocktailArr[r1];
```

```
fruitCocktailArr.splice(r1, 1);

r2 = Math.floor(Math.random() * fruitCocktailArr.length);
rFruit2 = fruitCocktailArr[r2];
```

15. Rerun the console.log command repeatedly to verify that no repeats occur. Soon, the array will be empty and the results `undefined`:

```
console.log(rFruit1, rFruit2);
```

slice()

The **slice()** method is called on an array and takes two arguments: a starting and ending index.

- `slice()` returns a new array without affecting the original array.
- `slice()` end index is exclusive, so *not* included in the new array.

16. Starting at index 2 and ending at index 5 (exclusive), get a new an array of 3 items:

```
let fruitSlices = fruitCocktailArr.slice(2, 5);

console.log(fruitSlices);
// ['strawberry', 'orange', 'papaya']
```

If you omit the second argument, it slices from the start index (first argument) all the way to the end:

17. Starting at index 4, slice all the way to the end:

```
let slicedFruit = fruitCocktailArr.slice(4);

console.log(slicedFruit);
// ['papaya', 'plum', 'tangerine']
```

includes()

The **includes()** method is called on an array and returns true if its argument is found in the array, and false if it is not:

18. Call the `includes()` method to get one true and one false result:

```
console.log(slicedFruit.includes('plum'));
// true
```

```
    console.log(slicedFruit.includes('pear'));
    // 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.

19. Declare an array of reptiles and run the indexOf() method a few times:

```
    const reptiles = ['iguana', 'snake', 'turtle', 'snake', 'gekko',
'snake', 'lizard'];

    console.log(reptiles.indexOf('snake')); // 1
    console.log(reptiles.indexOf('turtle')); // 2
    console.log(reptiles.indexOf('Komodo dragon')); // -1
```

To specify a starting index, pass in a second argument. This skips earlier instances of the word:

20. Starting at index 2, get the position of the first snake after that:

```
    console.log(reptiles.indexOf('snake', 2)); // 3
```

lastIndexOf()

The **lastIndexOf()** method returns the index of the last occurrence of the argument. If it is not found, it returns -1:

```
    console.log(reptiles.lastIndexOf('snake')); // 4
    console.log(reptiles.lastIndexOf('Gila monster')); // -1
```

join()

The **join()** method is called on an array and returns a string of all the items, separated by commas. It does not change the array.

21. Join the reptiles into a long slithering string:

```
    let reptilesStr = reptiles.join();
    console.log('reptilesStr', reptilesStr);
    // iguana,snake,turtle,snake,gekko,snake,lizard
```

The join method can take a *delimiter* argument--character(s) that will appear between the items in the resulting string.

22. Put an asterisk surrounded by spaces between each fruit in the string:

```
    let starryFruits = fruitCocktailArr.join(' * ');
    console.log(starryFruits);
    // apple * blueberry * strawberry * orange * papaya * pineappple *
plum * tangerine
```

flat()

The **flat()** method takes a nested array (2D, matrix) as its argument and returns a flat, one-dimensional array. It does not change the original array.

23. Declare a 3x3 / 2D / nested array:

```
    const ticTacToe = [
        ['X', 'O', null],
        [null, 'X', 'O'],
        ['O', null, 'X']
    ];
```

24. Flatten the array:

```
    const flatArray = ticTacToe.flat();

    console.log(flatArray);
    // console.log('flatArr', flatArr);
    // ['X', 'O', null, null, 'X', 'O', 'O', null, 'X']
```

copying an array with **slice(0)

An array can be copied in a number of ways. One way is `slice(0)`. Starting at index 0 and going all the way to the end by omitting the second argument, it returns a new array which includes all items in the original.

25. Declare an array and make a copy of it:

```
    const animals = ['giraffe', 'leopard', 'ostrich', 'zebra',  'panda',
'moose', 'bison', 'aardvark', 'baboon', 'rhinoceros'];

    const animalsCopy = animals.slice(0);

    console.log(animalsCopy);
    // ['giraffe', 'leopard', 'ostrich' etc.]
```

reverse()

The **reverse()** method reverses the order of the items in an array. It is often combined with **sort()** to flip the sorted order:

26. Sort and then reverse the animals array:

```
    console.log(animals.sort());
    console.log(animals.reverse());
    // ['zebra', 'rhinoceros', 'panda', 'ostrich', 'moose', 'leopard',
'giraffe', 'bison', 'baboon', 'aardvark']
```

methods chaining

Methods can be called one after another, on the same line. This is known as *methods chaining*.

27. In one line of code, sort and then reverse the `animalsCopy` array:

```
    animalsCopy.sort().reverse();

    console.log('animalsCopy', animalsCopy);
```

END Lesson 04.03

NEXT: Lab 04.03

array methods

1. Make a pets array containing 'cat', 'dog' and 'hamster'.

```
// your code
console.log(pets); // ['cat', 'dog', 'hamster']
```

array.push() adds item(s) to end of array

2. Push 'iguana' into the array.
3. Push 'parrot' into the array in one line.
4. Push 'snake' into the array in one line.

```
// your code
console.log(pets);
// ['cat', 'dog', 'hamster', 'iguana', 'parrot', 'snake']
```

array.pop() removes item from end of array

5. Remove (pop) the last item, which is 'snake'

```
// your code
console.log(pets);
// ['cat', 'dog', 'hamster', 'iguana', 'parrot']
```

6. Put (push) back 'snake':

```
// your code
console.log(pets);
// ['cat', 'dog', 'hamster', 'iguana', 'parrot', 'snake']
```

7. Do a pop() again, but this time save the return value:

```

```

```
    // your code
    console.log(pets);
    // ['cat', 'dog', 'hamster', 'iguana', 'parrot']
```

array.unshift() adds item to beginning of array

8. Add 'bunny' to the beginning of pets array:

```
    // your code
    console.log(pets);
    // ['bunny', 'cat', 'dog', 'hamster', 'iguana', 'parrot']
```

***array.shift() method removes first item of array**

9. Remove 'bunny' from the beginning of the pets array:

```
    // your code
    console.log(pets);
    // ['cat', 'dog', 'hamster', 'iguana', 'parrot']
```

array1.concat(array2) combines 2 or more arrays

10. Combine pets and morePets into the petsGalore array:

```
    // given:
    const morePets = ['canary', 'gerbil', 'kitten', 'python', 'turtle'];
    // combine pets and morePets into a new array, petsGalore:
    // your code
    // console.log(petsGalore);
    // ['cat', 'dog', 'hamster', 'iguana', 'parrot', 'canary', 'gerbil',
'kitten', 'python', 'turtle']
```

array.sort() from A-Z

11. Sort the **petsGalore** array:

```
    // your code
    // ['canary', 'cat', 'dog', 'gerbil', 'hamster', 'iguana', 'kitten',
'parrot', 'python', 'turtle']
```

array.splice(index, count)

12. remove the first two items, saving them to a variable called twoSplicedPets:

```
    // your code
    console.log(twoSplicedPets); // ['canary', 'cat']
    console.log(petsGalore); // ['dog', 'gerbil', 'hamster', 'iguana',
'kitten', 'parrot', 'python', 'turtle']
```

13. starting at the 4th item in petsGalore, splice 3 items in a row; save the items as threeSplicedPets

```
    // your code
    // console.log(threeSplicedPets);
    // ['iguana', 'kiten', 'parrot']
    // console.log(petsGalore);
    // ['dog', 'gerbil', 'hamster', 'python', 'turtle']
```

14. Using concat(), restore petsGalore to its original state of 10 pets and sort it from A-Z:

```
    // your codde
    console.log(petsGalore);
    // ['canary', 'cat', 'dog', 'gerbil', 'hamster', 'iguana', 'kitten',
'parrot', 'python', 'turtle']
```

15. Make a string connected by & of all items

```
    // console.log('petsStr:', petsStr);
    // canary & cat & dog & gerbil .. etc.
```

END: Lab 04.03
SEE Lab 04.03 Solution

04.03 Lab Exercises - SOLUTION

array methods

1. Make a pets array containing 'cat', 'dog' and 'hamster':

```
const pets = ['cat', 'dog', 'hamster'];
console.log('pets:', pets); // ['cat', 'dog', 'hamster']
```

array.push() adds item to end of array

2. Push 'iguana' into the array:

```
pets.push('iguana');
console.log('pets:', pets); // ['cat', 'dog', 'hamster', 'iguana']
```

3. Push 'parrot' into the array:

```
pets.push('parrot');
console.log('pets:', pets);
// ['cat', 'dog', 'hamster', 'iguana', 'parrot']
```

4. Push 'snake' into the array:

```
pets.push('snake');
console.log('pets:', pets);
// ['cat', 'dog', 'hamster', 'iguana', 'parrot', 'snake']
```

array.pop() removes item from end of array

5. remove (pop) the last item, which is 'snake':

```
pets.pop(); // remove last item, which is the snake
console.log('pets:', pets);
pets.push('snake');
```

6. Put (push) back 'snake':

```
console.log('pets:', pets);
// ['cat', 'dog', 'hamster', 'iguana', 'parrot', 'snake']
let poppedPet = pets.pop();
```

7. Pop 'snake' again, but this time save it as the return value:

```
    let poppedPet = pets.pop();
    console.log('poppedPet:', poppedPet); // snake
    console.log('pets:', pets);
    // ['cat', 'dog', 'hamster', 'iguana', 'parrot']
```

array.unshift() adds item to beginning of array

8. Add 'bunny' to the beginning of pets array

```
    pets.unshift('bunny');
    console.log('pets:', pets);
    // ['bunny', 'cat', 'dog', 'hamster', 'iguana', 'parrot']
```

array.shift() method removes first item of array

9. Remove 'bunny':

```
    pets.shift();
    console.log('pets:', pets); // ['cat', 'dog', 'hamster', 'iguana',
'parrot']
```

array1.concat(array2) combines 2 or more arrays

10. Combine pets and morePets into the petsGalore array:

```
    const morePets = ['canary', 'gerbil', 'kitten', 'python', 'turtle'];
    const petsGalore = pets.concat(morePets);
    // combine pets and morePets into new array, petsGalore
    console.log('pets:', pets);
    console.log('morePets:', morePets);
    console.log('petsGalore:', petsGalore);
    // ['cat', 'dog', 'hamster', 'iguana', 'parrot', 'canary', 'gerbil',
'kitten', 'python', 'turtle']
```

array.sort() from A-Z

11. Sort the petsGalore array:

```
    petsGalore.sort();
    console.log('petsGalore sorted:', petsGalore);
```

```
    // ['canary', 'cat', 'dog', 'gerbil', 'hamster', 'iguana', 'kitten',
'parrot', 'python', 'turtle']
```

array.splice(index, count)

12. get rid of the first two items but save them to a var called twoSplicedPets:

```
    let twoSplicedPets = petsGalore.splice(0, 2);
    console.log('twoSplicedPets:', twoSplicedPets); // ['canary', 'cat']
    console.log('petsGalore spliced:', petsGalore);
    // ['dog', 'gerbil', 'hamster', 'iguana', 'kitten', 'parrot',
'python', 'turtle']
```

13. starting at the 4th item in petsGalore, splice 3 items in a row; save the items as threeSplicedPets:

```
    let threeSplicedPets = petsGalore.splice(3, 3);
    console.log('threeSplicedPets:', threeSplicedPets);
    // ['iguana', 'kiten', 'parrot']
    console.log('petsGalore spliced:', petsGalore);
    // ['dog', 'gerbil', 'hamster', 'python', 'turtle']
```

14. Using concat(), restore petsGalore to its original state of 10 pets and sort it from A-Z:

```
    const petsGaloreRestored = petsGalore.concat(twoSplicedPets,
threeSplicedPets).sort();
    console.log('petsGalore restored:', petsGaloreRestored);
```

15. Make a string connected by & of all items:

```
    let petsStr = petsGalore.join( ' & '); // array.join() is called on an
array and returns a string of all items
    console.log('petsStr:', petsStr);
```

NEXT: Lesson 04.04

string methods

String methods are actions that are called on strings.

string[index] With arrays, you get items by index. With strings you get *characters* by index.

1. Declare a string and get the first and third characters:

```
let fruit = 'cherry';
// get the first and third characters:
console.log(fruit[0]); // c
console.log(fruit[2]); // e
```

string.length

As with arrays, the length prpoerty of a string returns the number of items, in this case characters.

2. Get the length of a password and then run an if-statement that tells if the password is too short (less than 15 characters):

```
let password = '35khD%ewG@1';
let pswdLen = password.length;
console.log('Password length:', pswdLen);
if(password.length < 15) {
    console.log('Password is too short!');
}
```

string.replace(a, b)

The **replace()** method is called on a string. It takes two arguments:

- what you want to replace
- what you want to replace with

It returns a new string, without changing the original. Change 'cherry' to 'berry':

```
let newStr = fruit.replace('ch', 'b');
console.log(newStr); // berry
```

Change 'n' to 't' in an attempt to change 'banana' to 'batata' (a kind of sweet potato):

```
fruit = 'banana';
let foodItem = fruit.replace('n', 't');
console.log(foodItem); // batana
```

Notice that we only got 'batana', which means that the replace() method only changes the first occurrence of the target character(s).

The **replace()** method can replace a space with a word. Turn 'fresh salad' into 'fresh garden salad':

```
appetizer = appetizer.replace(' ', ' garden '));
console.log(appetizer); // fresh garden salad
```

replaceAll()

To replace all instances of a character, use replaceAll():

```
console.log(fruit); // banana
foodItem = fruit.replaceAll('n', 't');
console.log(foodItem); // batata
```

includes()

The **includes()** method takes a character(s) as its argument and returns true or false, based on whether or not the character(s) are in the string;

```
let bev = 'papaya smoothie';
console.log(bev.includes('smooth')); // true
console.log(bev.includes('mango')); // false
```

indexOf()

The **indexOf()** method takes a character(s) as its argument and returns the index of the first occurrence:

```
console.log(fruit); // banana
console.log(fruit.indexOf('n')); // 2
```

lastIndexOf()

The **lastIndexOf()** method takes a character(s) as its argument and returns the index of the last occurrence:

```
console.log(fruit); // banana
console.log(fruit.lastIndexOf('n')); // 4
```

If the char is not found in the string, indexOf returns -1:

```
console.log(fruit.indexOf('x')); // -1
```

The **indexOf()** method is useful for finding the space between two words, which gives the length of the first word:

```
let appetizer = 'fresh salad';
console.log(appetizer.indexOf(' ')); // 5
```

charAt()

The **charAt()** method is called on a string. It takes an integer as an argument and returns the character at that index:

```
let fruit = 'apple';
console.log(fruit.charAt(0)); // a
console.log('Hello World'.charAt(6)); // W
```

slice()

The **slice()** method returns a substring of the string it is called on, without changing the original string. It takes two arguments: a starting and an ending index. The end index is exclusive, meaning it is not included in the returned string. If the end index is omitted, it slices to the end.

```
let dynamicDuo = 'Batamn and Robin';
let hero1 = dynamicDuo.slice(0, 7);
console.log(hero1);
```

indexOf() and lastIndexOF() with slice()

But what if we didn't know the length of the word. We would have to get the index of the first space and then use that as the end index:

```
let superCouple = 'Superman and Lois Lane';
let indxSpace = superCouple.indexOf(' ');
let superhero = superCouple.slice(0, indxSpace);
console.log(superhero);
```

To get 'Lois Lane', we could get the index of the first 'L' and then just slice to the end of the string:

```
let Lindex = superCouple.indexOf('L');
let supermansGF = superCouple.slice(Lindex);
console.log(supermansGF);
```

To get the last word of a string, get the index of the last space, and then slice from there to the end:

```
let lastSpaceIndex = superCouple.lastIndexOf(' ');
let lastWord = superCouple.slice(lastSpaceIndex);
console.log(lastWord);
```

split()

The **split()** method is called on a string and returns an array.

```
let movieQuote = 'Show me the money';
var movieQuoteArr1 = movieQuote.split();
console.log(movieQuoteArr1); // ['Show me the money']
```

Notice that there is only one item in the resulting array. To have each word become an array item, pass the split() method the space between the words as its **delimiter**:

```
let movieQuoteArr2 = movieQuote1.split(' ');
console.log(movieQuoteArr2); // ['Show', 'me', 'the', 'money']
```

toLowerCase()

The **toLowerCase()** method is called on a string and returns an all-lowercase version of the string, without changing the original:

```
let newsFlash = 'Yankees Win World Series';
console.log(newsFlash.toLowerCase());
console.log(newsFlash); // unchanged
```

string.split() + array.join() + toLowerCase() Split and join can be used to break a string into an array and then put it back as a string, with some transformation being done in the process.

Turn this news headline into a hyphenated file name.

```
let news = 'Mets Win World Series';
let newsArr = headline.split(' ');
console.log(newsArr);
// ['Mets', 'Win', 'World', 'Series']
let imgFile = newsArr.join('-') + '.jpg';
console.log(imgFile);
// Mets-Win-World-Series.jpg;
let imgFileLC = imgFile.toLowerCase();
 console.log(imgFileLC);
// mets-win-world-series.jpg;
```

toUpperCase()

The **toUpperCase()** method is called on a string and returns an all-uppercase version of the string, without changing the original:

```
let headline = 'Jets Win Superbowl';
let shouting = headline.toUpperCase());
console.log(shouting); // JETS WIN SUPERBOWL
```

toUpperCase() + slice() to capitalize a word

There is no method for capitalizing a word, but we can get the first character and uppercase that. We also slice off the rest of the string, and then reconnect that part to the uppercased first letter.

Get the first letter and uppercase it:

```
let firstName = 'alexandria';
let firstChar = firstName[0];
let firstCharUC = firstChar.toUpperCase();
console.log(firstCharUC);
```

Get the rest of the string:

```
let restOfName = firstName.slice(1);
```

Connect the capitalized first letter with the rest of the string:

```
let capitalizedName = firstCharUC + restOfName;
console.log(capitalizedName);
```

NEXT: 04.04 Lab

Dynamic Object Properties

Seaching by Object Properties

Given this array of vegetables:

```
const veggies = ['Cucumber', 'Tomato', 'Kale'];
```

1. Use the **replace()** method and array items to log the expected message to the console:

```
let msg1 = 'Have you ever ordered the Sandwich?';
console.log(msg1); // Have you ever ordered the Cucumber Sandwich?
```

2. Use the replace() method and array items to log the expected message to the console:

```
let msg2 = 'The "T" in "BLT" stands for "Terminator."';
console.log(msg2); // The "T" in "BLT" stands for "Tomato."
```

3. Use the replace() method and array items to log the expected message to the console:

```
let msg3 = 'Bunny goes to Yale University.';
console.log(msg3); // Bunny goes to Kale University.
```

Given this array of phrases:

```
const phrases = ["A real customer", "A breeze", "Hangin' with the
kids", "Snoopy is a dog"];
```

4. Using string methods and concatenation, make "cool" the next to the last word, no matter which phrase is chosen. Expected output is one of the following:

- a real cool customer
- a cool breeze
- hangin' with the cool kids
- Snoopy is a cool dog

Since we can't predict what the random phrase will be, we cannot use the **replace()** method on a hard-coded last word of the string. So, where does cool need to go? It needs to go in at the last space location. What string method returns the last space of a phrase: **lastIndexOf(" ")** test this on **phrases[0]**, which is the first item in phrases Save the last index of the the space to a variable:

```
    let lastSpaceIndex;
```

then use the **slice()** method twice: once to get everything up to (but not including) the first space; use the **slice()** method again to get everything from the last space to the end of the string; concatenate the two slices together--with " cool " in between.

```
    let coolPhrase;
    console.log("Cool phrase test:", coolPhrase);
```

test the result with a random number in the range of the array

```
    let r;
    console.log("Random cool phrase:", coolPhrase);
```

Given this array of fruits and vegetables:

```
    const words = ['Avocado', 'Broccoli', 'Cauliflower', 'Dragonfruit',
'Eggplant'];
```

5. Use the includes() method and if-else logic to check if the word starts with a consonant or a vowel

- If the word is "Avocado", log: Avocado starts with "A", which is a vowel.
- If the word is "Broccoli", log: Broccoli starts with "B", which is a consonant.

```
    let word;
    let firstChar;
    console.log('firstChar', firstChar);
    let vowels;
    if(vowels) {
    }
```

REFACTOR: Since the if-else console logs differ only in one word: vowel / consonant, better would be to have a variable "consonant" that the if statements sets to "vowel" and then concat that "consonant" or "vowel" vvariable into the output:
Try it again:

```
    let consonantOrVowel;
    if(vowels) {
    }
```

Given this file name:

```
    let fileName = "Mets-Lead-Off-Game-By-Hitting-Three-Straight-
Homers.html";
```

6. Turn the file name into a news headline, with all words capitalized:

Expected result:
Mets Lead Off Game By Hitting Three Straight Homers

```
    let wordsArr;
    console.log(wordsArr); // ['Mets', 'Lead', 'Off', 'Game', 'By',
'Hitting' .. etc.]
    let headline;
```

Remove the file extension, but don't assume '.html' -- detect the dot and remove everything from the dot to the end of the string

```
    let indexOfDot;
```

Use slice() method to get everything from the beginning of the string to to dot (but not includding the dot)

See: Lab 04.04 Solution

Dynamic Object Properties

Seaching by Object Properties

Given this array of vegetables:

```
const veggies = ['Cucumber', 'Tomato', 'Kale'];
```

1. Use the **replace()** method and array items to log the expected message to the console:

```
let msg1 = 'Have you ever ordered the Sandwich?';
msg1 = msg1.replace('Sandwich', veggies[0] + ' Sandwich');
console.log(msg1); // Have you ever ordered the Cucumber Sandwich?
```

2. Use the replace() method and array items to log the expected message to the console:

```
let msg2 = 'The "T" in "BLT" stands for "Terminator."';
msg2 = msg2.replace('Terminator', veggies[1]);
console.log(msg2); // The "T" in "BLT" stands for "Tomato."
```

3. Use the replace() method and array items to log the expected message to the console:

```
let msg3 = 'Bunny goes to Yale University.';
msg3 = msg3.replace('Yale', veggies[2]);
console.log(msg3); // Bunny goes to Kale University.
```

Given this array of phrases:

```
const phrases = ["A real customer", "A breeze", "Hangin' with the
kids", "Snoopy is a dog"];
```

4. Using string methods and concatenation, make "cool" the next to the last word, no matter which phrase is chosen. Expected output is one of the following:

- a real cool customer
- a cool breeze
- hangin' with the cool kids
- Snoopy is a cool dog

Since we can't predict what the random phrase will be, we cannot use the **replace()** method on a hard-coded last word of the string. So, where does cool need to go? It needs to go in at the last space location. What string method returns the last space of a phrase: **lastIndexOf(" ")** test this on **phrases[0]**, which is the first item in phrases Save the last index of the the space to a variable:

```
    let lastSpaceIndex = phrases[0].lastIndexOf(' ');
```

then use the **slice()** method twice: once to get everything up to (but not including) the first space; use the **slice()** method again to get everything from the last space to the end of the string; concatenate the two slices together--with " cool " in between.

```
    let coolPhrase = phrases[0].slice(0, lastSpaceIndex) + " cool " +
phrases[0].slice(lastSpaceIndex+1);
    console.log("Cool phrase test:", coolPhrase);
```

test the result with a random number in the range of the array

```
    let r = Math.floor(Math.random() * phrases.length);
    // get the index of the last space of the new, randomly selected
phrase
    lastSpaceIndex = phrases[r].lastIndexOf(' ');
    // concatenate the two slices together--with " cool " in between.
    coolPhrase = phrases[r].slice(0, lastSpaceIndex) + " cool " +
phrases[r].slice(lastSpaceIndex+1);
    console.log("Random cool phrase:", coolPhrase);
```

Given this array of fruits and vegetables:

```
    const words = ['Avocado', 'Broccoli', 'Cauliflower', 'Dragonfruit',
'Eggplant'];
```

5. Use the includes() method and if-else logic to check if the word starts with a consonant or a vowel

- If the word is "Avocado", log: Avocado starts with "A", which is a vowel.
- If the word is "Broccoli", log: Broccoli starts with "B", which is a consonant.

```
    r = Math.floor(Math.random() * words.length);
    let word = words[r];
    let firstChar = word[0];
    console.log('firstChar', firstChar);
    let vowels = 'aeiouAEIOU';
    if(vowels.includes(firstChar)) {
        console.log(`${word} starts with "${firstChar}", a vowel.`);
```

```
    } else {
        console.log(`${word} starts with "${firstChar}", a consonant.`);
    }
```

REFACTOR: Since the if-else console logs differ only in one word: vowel / consonant, better would be to have a variable "consonant" that the if statements sets to "vowel" and then concat that "consonant" or "vowel" vvariable into the output:
Try it again:

```
    r = Math.floor(Math.random() * words.length);
    let consonantOrVowel = "consonant";
    word = words[r];
    firstChar = word[0];
    if(vowels.includes(firstChar)) {
        consonantOrVowel = "vowel";
    }
    console.log(`${word} starts with "${firstChar}", which is a
${consonantOrVowel}`);
```

Given this file name:

```
    let fileName = "Mets-Lead-Off-Game-By-Hitting-Three-Straight-
Homers.html";
```

6. Turn the file name into a news headline, with all words capitalized:

Expected result:
Mets Lead Off Game By Hitting Three Straight Homers

```
    let wordsArr = fileName.split('-');
    console.log(wordsArr); // ['Mets', 'Lead', 'Off', 'Game', 'By',
'Hitting' .. etc.]
    let headline = wordsArr.join(' ');
```

Remove the file extension, but don't assume '.html' -- detect the dot and remove everything from the dot to the end of the string

```
    let indexOfDot = headline.indexOf('.');
```

Use slice() method to get everything from the beginning of the string to to dot (but not includding the dot)

```

```

```
    headline = headline.slice(0, indexOfDot);
    console.log(headline);
```

END: Lab 04.04

NEXT: Lesson 05.01

For Loops

Interating (Looping) Arrays

A loop repeatedly executes a block of code, 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**:

```
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:

```
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 do not run this infinite loop, as doing so may freeze your browser. Just study it before commenting it out.

```
/* infinite loop; do not run
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
}
```

continue

The `continue` keyword skips an iteration of a loop. It is used with conditional logic to specify when to skip:

9. Output all numbers from 1-10, with the exception of 7:

```
for(let i = 0; i < 10; i++) {
    if(i == 7) {
        continue; // skip 7
    }
    console.log(i); // 0,1,2,3,4,5,6,8,9
}
```

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. Add a few more items to the end of the array:

```
    fruits.push('apricot');
    fruits.push('papaya')
    fruits.push('grape');
```

12. Add a few items to the beginning of the array:

```
    fruits.unshift('grapefruit');
    fruits.unshift('watermelon');
    fruits.unshift('tangerine');
```

13. Output the array and its length:

```
    console.log(fruits, fruits.length);
    // ['tangerine', 'watermelon', 'grapefruit', 'apple', 'blueberry',
'cherry', 'kiwi', 'lime', 'orange', 'plum', 'apricot', 'papaya', 'grape']
13
```

14. Starting at index 3, splice out 3 items ('apple', 'blueberry', 'cherry'), and replace them with 'lemon' and 'pear':

```
    fruits.splice(3, 3, 'lemon', 'pear');

    console.log(fruits, fruits.length);
    // ['tangerine', 'watermelon', 'grapefruit', 'lemon', 'pear', 'kiwi',
'lime', 'orange', 'plum', 'apricot', 'papaya', 'grape'] 12
```

15. At index 7, insert 4 items without removing any. The new fruits will go in before 'orange':

```
    fruits.splice(7, 0, 'apple', 'blueberry', 'cherry', 'peach');

    console.log(fruits, fruits.length);
    //  ['tangerine', 'watermelon', 'grapefruit', 'lemon', 'pear', 'kiwi',
'lime', 'apple', 'blueberry', 'cherry', 'peach', 'orange','plum',
'apricot', 'papaya', 'grape'] 16
```

16. Iterate the array of 16 items with a for loop. Each time through the loop, make a fruit jellybean.

```
    for(let i = 0; i < 16; i++) {
        let bean = fruits[i] + ' jellybean';
        console.log(bean); // tangerine jellybean, etc.
    }
```

17. Push in a few more fruits, and then sort the array:

```
    fruits.push('banana');
    fruits.push('pineapple');
    fruits.push('mango');
    fruits.sort();

    console.log(fruits, fruits.length);
    // ['apple', 'apricot', 'banana', 'blueberry', 'cherry', 'grape',
'grapefruit', 'kiwi', 'lemon', 'lime', 'mango', 'orange', 'papaya',
'peach', 'pear', 'pineapple', 'plum', 'tangerine', 'watermelon'] 19
```

18. Run the loop again:

```
    for(let i = 0; i < 16; i++) {
       let bean = fruits[i] + ' jellybean';
       console.log(bean);
       // apple jellybean, apricot jellybean, etc.
    }
```

The last three fruits ('plum', 'tangerine', 'watermelon') are not showing up. The problem is, the loop stops when `i=16`, but we now have 19 items.

array.length as loop condition

For a loop that iterates an array, do not use a hard-coded number in the condition. Instead, use the dynamic condition `i < array.length`. This way, the loop fully iterates the array--and then stops.

19. Make a new loop with the dynamic condition `i < fruits.length`:

```
    for(let i = 0; i < fruits.length; i++) {
        let bean = fruits[i] + ' jellybean';
        console.log(bean);
        // apple jellybean, apricot jellybean, etc.
    }
```

string.length

As we recall, the `length` property applies not only to arrays, but to strings, as well.

conditional logic inside a loop

Loops that iterate arrays often include conditional logic to evaluate the individual items. Let's give this a try.

20. Make jellybeans only if the fruit has no more than 5 letters. Btw, we don't really need the bean variable, so just log the string directly:

```
    for(let i = 0; i < fruits.length; i++) {
        if(fruits[i].length <= 5) {
            console.log(fruits[i] + ' jellybean');
            // apple jellybean, grape jellybean, etc.
        }
    }
```

making a new array inside a loop

The above loops are not saving the jellybeans anywhere; the beans are just being "spilled" out onto the console.

21. Do another loop that saves the jellybeans by pushing them into a new array, `jellybeans`, declared above the loop:

```
    const jellybeans = [];

    for(let i = 0; i < fruits.length; i++) {
        if(fruits[i].length <= 5) {
            jellybeans.push(fruits[i] + ' jellybean');
        }
    }

    console.log(jellybeans);
    // ['apple jellybean', 'grape jellybean', 'kiwi jellybean', 'lemon
jellybean, 'lime jellybean', 'mango jellybean', 'peach jellybean', 'pear
jellybean', 'plum jellybean']
```

Let's try some if-else if-else logic in a loop:

- `if` the fruit has a max of 5 letters, it goes into the `lilFruits` array.

- `else if` the fruit has 6-8 letters, it goes into the `medFruits` array.
- `else`, the fruit of 9 or more letters it goes into the `bigFruits` array

22. Declare the three arrays, all empty:

```
    const lilFruits = [];
    const medFruits = [];
    const bigFruits = [];
```

23. Run the loop with the conditional logic. Each part pushes eligible fruits into its respective array:

```
    for(let i = 0; i < fruits.length; i++) {
        if(fruits[i].length <= 5) {
            lilFruits.push(fruits[i]);
        } else if(fruits[i].length <= 8) {
            medFruits.push(fruits[i]);
        } else { // 9+ chars
            bigFruits.push(fruits[i]);
        }
    }

    console.log('lilFruits', lilFruits);
    // ['apple', 'grape', 'kiwi', 'lemon', 'lime', 'mango', 'peach',
'pear', 'plum']
    console.log('medFruits', medFruits);
    // ['apricot', 'banana', 'cherry', 'orange', 'papaya']
    console.log('bigFruits', bigFruits);
    // ['blueberry', 'grapefruit', ' 'pineapple', 'tangerine',
'watermelon']
```

loop with nested if-else

Let's finish the lesson with some nested logic. Building upon what we have so far, if the "big fruit" is a berry, save it to the `berries` array. For this we need a few more berries.

We will use `splice()` to maintain alphabetical order as we add 'boysenberry', 'raspberry' and 'strawberry'. Here is the current array of 19 items for reference:

```
['apple', 'apricot', 'banana', 'blueberry', 'cherry', 'grape',
'grapefruit', 'kiwi', 'lemon', 'lime', 'mango', 'orange', 'papaya',
'peach', 'pear', 'pineapple', 'plum', 'tangerine', 'watermelon']
```

24. We need to put 'boysenberry' needs between 'blueberry' and 'cherry'. Starting at index 4 ('cherry'), remove zero items and add 'boysenberry':

```
    fruits.splice(4, 0, 'boysenberry');
```

splice() with negative index values

The splice method recognizes negative indexes, with the last item at -1. We want to add 'raspberry' and 'strawberry', consecutively, right before 'tangerine', which, as the next to the last item, occupies index -2.

25. Add 'raspberry' and 'strawberry' before 'tangerine':

```
    fruits.splice(-2, 0, 'raspberry', 'strawberry');

    console.log(fruits, fruits.length);
    // ['apple', 'apricot', 'banana', 'blueberry', 'boysenberry',
'cherry', 'grape', 'grapefruit', 'kiwi', 'lemon', 'lime', 'mango',
'orange', 'papaya', 'peach', 'pear', 'pineapple', 'plum', 'raspberry',
'strawberry', 'tangerine', 'watermelon']
```

26. Declare all new empty arrays for a fresh start:

```
    const smFruits = [];
    const mdFruits = [];
    const lgFruits = [];
    const berries = [];
```

27. Do the for loop with nested if-else. To check if a fruit *includes* 'berry', call `includes('berry')` on the item:

```
    for(let i = 0; i < fruits.length; i++) {
        if(fruits[i].length <= 5) {
            smFruits.push(fruits[i]);
        } else if(fruits[i].length <= 8) {
            mdFruits.push(fruits[i]);
        } else { // 9+ chars
            if(fruits[i].includes('berry')) {
                berries.push(fruits[i]);
            } else {
                lgFruits.push(fruits[i]);
            }
        }
    }

    console.log('smFruits', smFruits);
    // ['apple', 'grape', 'kiwi', 'lemon', 'lime', 'mango', 'peach',
'pear', 'plum' ]
    console.log('mdFruits', mdFruits);
    // ['apricot', 'banana', 'cherry', 'orange', 'papaya']
    console.log('lgFruits', lgFruits);
    // ['grapefruit', ' 'pineapple', 'tangerine', 'watermelon']
```

```
    console.log('berries', berries);
    // ['blueberry', 'boysenberry', 'raspberry', 'strawberry']
```

- **END: Lesson 05.01**
- **NEXT: Lab 05.01**

LOOPS & ARRAYS

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 mix = [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"];
```

See: Lab 05.01 Solution

05.01 Lab Exercises - SOLUTION

LOOPS & ARRAYS

1. Write a for loop that makes the following array:

[3, 5, 7, 9, 11, 13, 15, 17, 19, 21]

Solution explained:

- New, empty array, holds the output
- Counter variable, i, starts at 3
- Loop continues as long as counter i <= 21
- Values are increasing by 2, so we need incrementer: i += 2.

```
const myNums = [];
for(let i = 3; i <= 21; i += 2) {
    myNums.push(i);
}
console.log(myNums);
```

2. Write a for loop that makes the following array:

[100, 80, 60, 40, 20, 0, -20, -40, -60, -80, -100]

Solution explained:

- New, empty array, holds the output
- Counter variable, `i`, starts at 100
- Loop continues as long as counter `i >= -100`
- Values are decreasing by 20, so we need decrementer: `i -= 20`.

```
const numsArr = [];
for(let i = 100; i >= -100; i -= 20) {
    console.log(i);
}
console.log(numsArr);
```

3. Given an array of numbers, use a for loop to add up all the numbers. Save the total to a variable, sum.

```
let nums = [154, 236, 314, 467, 532, 689, 703];
let sum = 0;
```

Solution expained:

- Loop through the array.

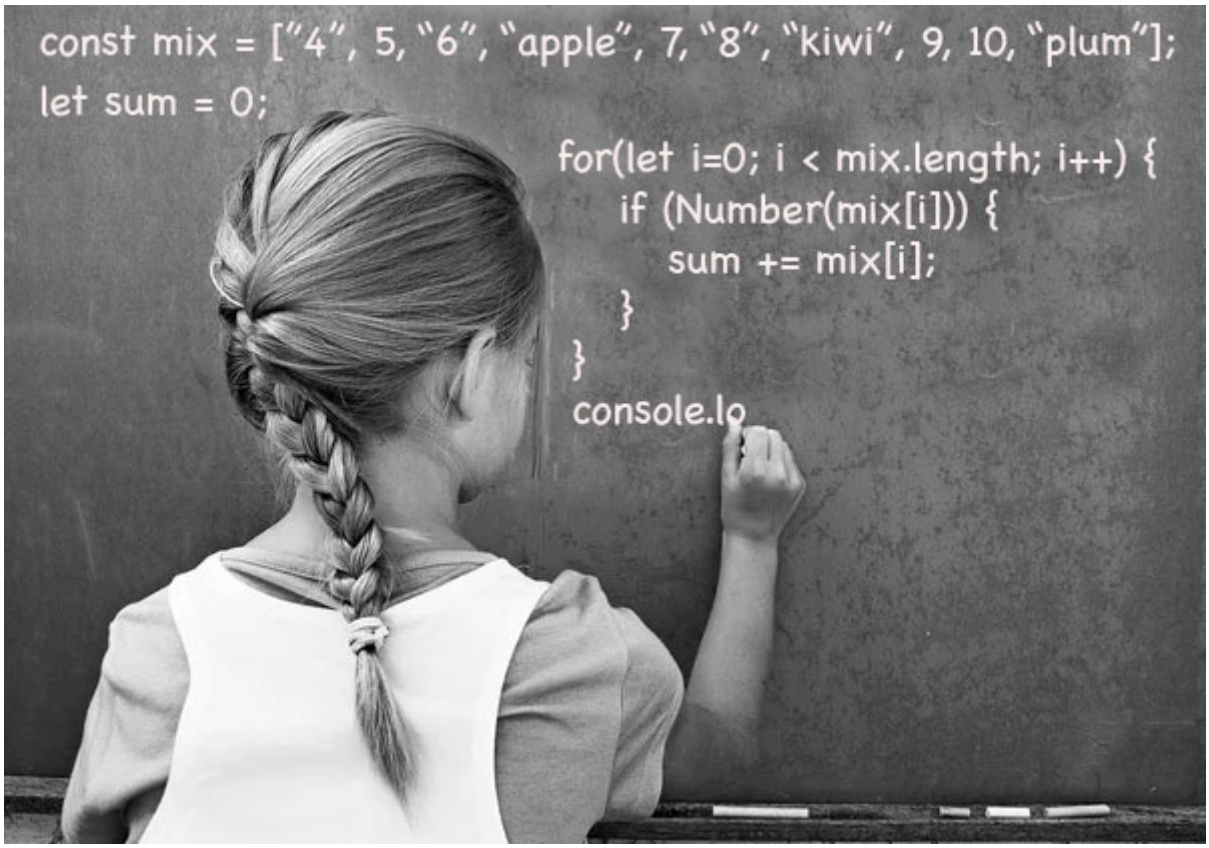
- Each time through, add the current item, nums[i] to sum.

```
for(let i = 0; i < nums.length; i++) {
    sum += nums[i];
}
console.log('nums sum:', sum); // 3095
```

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”];
```

Solution explained



- Convert the 'num-like strings' to numbers, so that we can add them.
- Skip the fruits, since they cannot be converted to numbers.
- Add each eligible item as we loop through the array.
- Number('55') returns 55
- Number('apple') returns NaN.
- NaN is falsey, which returns false in a boolean context, e.g. in an if-statement.
- Therefore, pass each item to the Number() method and pass that to an if() statement. Any values that cannot be converted to a number -- these being the fruits -- will return NaN, and so the if-code will

not run.

- Inisde the if, add up values that are NOT NaN.

```
    let tot = 0;

    for(let i = 0; i < mix.length; i++) {
        if(Number(mix[i]) { // NaN is falsey
            tot += mix[i]; // only add truthy values
        }
    }
    console.log('mix tot:', tot);
```

Looping Arrays of Objects

making a video player interface

looping an array of objects

1. Open `05.02-Looping-Arrays-of-Objects.html` and preview it in the browser.

The interface loads a grid of 18 animal images at left and a video player with description underneath, at right. The first animal, 'American bison' appears on page load.

2. Click an animal picture to swap the video and description.

This is the application that we will be building in this lesson. Let's have a look at where all this animal data is located.

3. Open `animals.js` and have a look. The file contains one variable, an array called `animalsArr`.

Each array item is an object of six properties: - `name` ('giraffe', 'ostrich', etc.) - `class` ('mammal', 'bird', 'reptile') - `herbivore` (boolean: true or false) - `continent` ('Asia', 'Africa', etc.) - `youTube` (YouTube video embed code) - `desc` (description of the animal)

4. Switch to the lesson file's JS file, `05.02-Looping-Arrays-of-Objects-START.js`.

To make the application, we will loop through the animals array:

- Each time through the loop, we will concatenate an `img` tag.
- The `img` tag will have an `onclick` event that calls a function called `swapImage(i)`.
- The `i` argument is the index of the current item.
- The argument `i` tells the function which video and description it needs to load.
- The YouTube video is embedded with an `iframe` tag, which is the same for each video, except for an 11-character code specific to that particular video.

5. First, get all the elements that we need for the output:

- `animal-pics` div for the grid of animal pics
- `video-player` div for holding the video
- `h2` for displaying the animal name
- `p` for displaying the description.

```
// get animals pic div to hold images
let animalPicsDiv = document.getElementById('animal-pics');
```

```
    // get video player div
    let videoPlayerDiv = document.getElementById('video-player');

    // get h2 for animal name
    let h2 = document.querySelector('h2');
    h2.textContent = animalsArr[0].name;

    // get p tag for description
    let p = document.querySelector('p');
    p.textContent = animalsArr[0].desc;
```

6. Output the name and description of the first animal immediately on page load.

- the first array item is `animalsArr[0]`
- each array item is an object, so the name of the first animal is `animalsArr[0].name`

```
    h2.textContent = animalsArr[0].name;
    p.textContent = animalsArr[0].desc;
```

7. Display the video for the first animal.

- the `iframe` for each video is identical, except for an 11-character code
- the code is stored as the object's `youTube` property
- we access the property with `animalsArr[0].youTube` and concatenate that into the `iframe`:

```
    // display video for first animal
    videoPlayerDiv.innerHTML = '<iframe width="560" height="315"
src="https://www.youtube.com/embed/' + animalsArr[0].youTube + '"
title="YouTube video player" frameborder="0" allow="accelerometer;
autoplay clipboard-write; encrypted-media; gyroscope; picture-in-picture"
allowfullscreen></iframe>';
```

8. Make sure the html page is using START.js and reload the page in the browser. We should have the video and description at right, and an empty box at left.

Now, to output the animal images to the `animal-pics` div, the empty box at left. Each image needs to be clickable to call a function to swap the video and description.

9. Set up a loop that iterates `animalsArr`. First thing to do in the loop is to save the current object to a variable:

```
    for (let i = 0; i < animalsArr.length; i++) {
        let animal = animalsArr[i];
    }
```

10. Open the `images` folder. Notice that the two-word file names are hyphenated, whereas the animal names in the data have spaces--not hyphens:

- FILE: american-bison.jpg, andean-bear.jpg
- DATA: 'American bison', 'Andean bear'

We will use object names ('American bison') to concatenate image file paths, so we need to replace the spaces with hyphens.

11. Using backticks, concatenate an entire `img` tag, using `${name}` to add the file name, dynamically:

```
    for (let i = 0; i < animalsArr.length; i++) {
        let animal = animalsArr[i];
        let name = animal.name.replace(' ', '-');
        let tag = `<img src="images/${name}.jpg">`;
    }
```

12. We need each tag to be clickable to call the `swapAnimal()` function, so add that to the tag as an `onclick` event handler:

```
    for (let i = 0; i < animalsArr.length; i++) {
        let animal = animalsArr[i];
        let name = animal.name.replace(' ', '-');
        let tag = `<img src="images/${name}.jpg" onclick="swapAnimal()">`;
    }
```

13. The function needs to know which animal was clicked, so pass in the index, `i`, as the argument.

```
    for (let i = 0; i < animalsArr.length; i++) {
        let animal = animalsArr[i];
        let name = animal.name.replace(' ', '-');
        let tag = `<img src="images/${name}.jpg"
onclick="swapAnimal(${i})">`;
        animalPicsDiv.innerHTML += tag;
    }
```

14. Output the `img` tag. Since this is an html tag, set the `innerHTML` rather than the `textContent`:

```
    for (let i = 0; i < animalsArr.length; i++) {
        let animal = animalsArr[i];
        let name = animal.name.replace(' ', '-');
        let tag = `<img src="images/${name}.jpg"
onclick="swapAnimal(${i})">`;
        animalPicsDiv.innerHTML += tag;
    }
```

15. Save and reload the page; the images should all be there although clicking them doesn't work since we have yet to write the `swapAnimal()` function. That's next.

16. Define the function. It has a parameter `i` which comes in when the function is called as the index of the clicked animal.

```
    function function swapAnimal(i) {
    }
```

16. Using the `i` argument, look up the animal in the array and set the heading and description to that animal's `name` and `desc` properties, respectively:

```
    function function swapAnimal(i) {
        h2.textContent = animalsArr[i].name;
        p.textContent = animalsArr[i].desc;
    }
```

17. Now for the big `iframe` tag for the video. Just slot in the chosen animal's 11-digit YouTube video code, which we access as `animalsArr[i].youTube`:

```
    function swapAnimal(i) {
        h2.textContent = animalsArr[i].name;
        p.textContent = animalsArr[i].desc;

        videoPlayerDiv.innerHTML = '<iframe width="560" height="315"
src="https://www.youtube.com/embed/' +
        animalsArr[i].youTube + '" title="YouTube video player"
frameborder="0" allow="accelerometer; autoplay; clipboard-write;
encrypted-media; gyroscope; picture-in-picture" allowfullscreen>
</iframe>';
    }
```

18. Save and reload the page. Click an animal to load its video and description.

END: Lesson 05.02 NEXT: Lesson 05.03

while loops

do while loops

break

nested for loops

while loop**

A while loop has the essential three ingredients found in a for loop: a counter, a condition and an incrementer. The difference is that in a for loop, these three things are all neatly bundled in parentheses. In a while loop, only the condition is inside the parentheses. The counter is actually declared *outisde* the loop, above it in the global scope; the counter is incremented *inside* the loop's curly braces.

1. Write this while loop:

```
let n = 0;
while (n <= 10) {
    console.log(n);
    n++;
}
console.log(n);
```

do while loop

In a "do while loop", the condition is evaluated only *after* the loop has already run, and so it must run at least once.

2. Write this 'do while' loop:

```
let num = 0;
do {
    console.log(num);
    num++;
} while(num <= 10);
```

Overall, while and do while loops are pretty similar:

- both declare the counter *outside (before)* the loop

- both increment the counter *inside* the loop

3. Write these while and do while loops that concatenate the counter, resulting in output of 1, 12, 123, 1234, etc:

```
let i = 0, str = "";

while (i <= 10) {
    str += i;
    i++;
    console.log(str);
}

str = "";
i = 10;

do {
    str += i;
    i--;
    console.log(str);
} while (i >= 1);
```

break

Recall that **continue** skips an iteration of a loop. The keyword **break** exits the loop.

4. Write a while loop that ends the loop when the counter gets to 7:

```
let x = 0;
while(x <= 10) {
    if(x == 7) {
        console.log('break');
        break;
    }
    console.log('x', x);
    x++;
}
```

`break` is good for ending a loop once we have found something in an array. After all, if we found what we were looking for, there's no need to check the other items.

5. Write a while loop that iterates an array, looking for a number divisible by 7:

- *if* the number is found, save it to `target` and `break`.
- *if* no number divisible by 7 is found, set the index to -1.
- when it's all over, log the `target` and its index.

```
let nums = [53, 37, 123, 88, 112, 136, 155];
let indx = 0, target = 0;
while(indx <= nums.length) {
    if(nums[indx] % 7 == 0) {
        target = nums[indx];
        console.log('break');
        break;
    }
    indx++;
}
if (!target) indx = -1;
console.log('target', target, 'index', indx);
```

Recall that 0 is falsey, meaning that it returns false in an if-statement. So, just to be fancy, we're using `if(!target)` rather than `if(target == 0)` -- and putting it all on one line with no curly braces.

6. Change the array values so that the loop runs with and without finding a number divisible by 7.

nested loops:

A **nested loop** is a loop-inside-a-loop. Each time the outer loop runs once, it iterates over the entire inner loop. So, if each loop is running ten times, the total is 10x10 = 100 iterations.

7. Run a nested loop where we output the values of both inner and outer counter as we go:

```
for (let i = 1; i <= 10; i++) {
    for (let j = 1; j <= 10; j++) {
        console.log('i:', i, 'j:', j);
    }
}
```

This next one involves an array of city abbreviations. Once again, we will use a nested loop. Both loops will iterate over the same array. The inner loop will use the outer and inner counters, **i** and **j**, respectively, to concatenate a pair of cities. The names will be hyphenated, like departures-arrivals: NY-MIA

8. Declare an array of city abbreviations:

```
const cities = ['CHI', 'NY', 'PHI', 'BOS', 'MIA', 'LA', 'SF'];
```

We will iterate the array with a for loop and console log all possible pairings of cities. We need a nested loop for this, since each city needs to be paired with every other city.

9. Pair each city with every other city--including itself:

```
for(let i = 0; i < cities.length; i++) {
    for(let j = 0; j < cities.length; j++) {
```

```
            console.log(cities[i] + '-' + cities[j]);
        }
    }
```

10. Use conditional logic to only pair cities when `i` and `j` are not equal:

```
    for(let i = 0; i < cities.length-1; i++) {
        for(let j = 0; j < cities.length; j++) {
            if(i != j) { // don't pair a city with itself
                console.log(cities[i] + '-' + cities[j]);
            }
        }
    }
```

What if we didn't want any two-word combo to repeat, not even in the opposite direction. So, MIA-NY means no NY-MIA. For flights this may not be appropriate, because planes fly in both directions, but for some pairings, it may be best to only have the one pairing.

The challenge is to "make smoothies" as pairs of fruits. We want all possible combinations, but without any repeats in reverse, so "banana-peach" is good, but "peach-banana" is bad.

11. Declare an array of fruits:

```
    const fruits = ['apple', 'apricot', 'banana', 'blueberry', 'cherry',
'kiwi', 'mango', 'orange', 'peach', 'pear', 'plum', 'raspberry'];
```

Start with a rough version that gives you all 144 possible pairings--the 12x12 max, including self-pairs like "apple-apple", and revserve pairs like "apple-apricot" and "apricot-apple"). Include a counter that we increment with each pair, so that we can know if the total is going down as we proceed:

12. Make all 144 possible 2-fruit combinations, and keep count:

```
    let counter = 1;

    for(let i = 0; i < fruits.length; i++) {
        for(let j = 0; j < fruits.length; j++) {
            console.log(counter, fruits[i] + '-' + fruits[j]);
            counter++;
        }
    }
```

13. Reset the counter, and run a new loop, where you only get 132 pairs, because the self-pairings ("apple-apple") have been eliminated. This is done with conditional logic: we only make the pair if `i` and `j` are *not* equal:

```
counter = 1;

for(let i = 0; i < fruits.length; i++) {
    for(let j = 0; j < fruits.length; j++) {
        if(i != j) { // don't pair a fruit with itself
            console.log(counter, fruits[i] + '-' + fruits[j]);
            counter++;
        }
    }
}
```

Upgrade again, this time to the final version:

- Each time through the loop, set the just-finished item to "done".
- Add another condition to the if-statement requiring that the current fruit has not yet been 'done'.
- The result is pairs that do not repeat in reverse. We get "apple-banana", but not "banana-apple".
- Save the pairs to a smoothies array, rather than just logging them to the console.

14. Write the final version of the "smoothies maker", in accordance with the above guidelines:

```
counter = 1;
const smoothies = [];

for(let i = 0; i < fruits.length; i++) {
    for(let j = 0; j < fruits.length; j++) {
        // don't pair a fruit with itself or reuse a 'done' fruit
        if(i != j && fruits[j] != 'done') {
            smoothies.push(fruits[i] + '-' + fruits[j]);
            counter++;
        }
    }
    fruits[i] = 'done'; // replace the done fruit with 'done'
}
console.log('2-fruit smoothies', smoothies);
```

15. Make it into a function that takes the array as its argument, runs the "smoothie maker" code and, in good input-output fashion, returns the smoothies:

```
function makeSmoothies(arr) {

    counter = 1;
    const smoothesArr = [];

    for(let i = 0; i < arr.length; i++) {
        for(let j = 0; j < arr.length; j++) {
            // don't pair item w itself or reuse 'done'
            if(i != j && arr[j] != 'done') {
                smoothesArr.push(arr[i] + '-' + arr[j]);
```

```
                    counter++;
                }
            }
            arr[i] = 'done'; // replace done item with 'done'
        }

        return smoothesArr;

    }
```

16. Log fruits to see that all its items have been replaced with 'done':

```
    console.log('fruits', fruits, fruits.length);
```

17. Redeclare fruits under a new name:

```
    const froots = ['apple', 'apricot', 'banana', 'blueberry', 'cherry',
'kiwi', 'mango', 'orange', 'peach', 'pear', 'plum', 'raspberry'];
```

18. Declare an array of tropical fruit, so that we can test the function on more than just one array:

```
    const tropicalFruits = ['banana', 'pineapple', 'kiwi', 'mango',
'mangosteen', 'cherimoya', 'acai', 'noni', 'papaya', 'dragon fruit',
'passion fruit', 'guava'];
```

19. Call the function twice, passing in a different array each time. Save the function call to a variable so that you capture the return value (the smoothies):

```
    let smoothies1 = makeSmoothies(froots);
    console.log('froot smoothies', smoothies1);

    let smoothies2 = makeSmoothies(tropicalFruits);
    console.log('tropical smoothies', smoothies2);
```

- **END Lesson 05.03**
- **NEXT: Lesson 05.043**

Nested Loops to make Deck of Cards

nested loop to make a deck of playing cards

In this lesson we will use a nested loop to make a deck of cards from which we will deal a 5-card hand of poker.

1. Open **05.04-Deck-of-Cards.html** and see that it has a div of five image tags, under which is a DEAL CARDS button:

```
    <div id="card-box">
        <img src="images/10-of-Diamonds.png">
        <img src="images/Jack-of-Diamonds.png">
        <img src="images/Queen-of-Diamonds.png">
        <img src="images/King-of-Diamonds.png">
        <img src="images/Ace-of-Diamonds.png">
    </div>

    <button>DEAL CARDS</button>
```

2. Preview the html file in the browser. It's a 5-card hand of poker--a royal flush, no less.
3. Click the DEAL CARDS button. Five random cards replace the royal flush.
4. Open the **images** folder and click a card to see it. Each card has two identifiers: a **kind** and a **suit** for a file name structure of **kind-of-suit.png**:

- 2-of-Clubs.png - Queen-of-Spades.png

5. Switch the html file to use START .js.
6. Open **05.04-Deck-of-Cards-START.js**. The **kind** and **suit** arrays are given, but we also need a new empty array to store the cards.
7. Declare a new empty array, called **deck**:

```
    const kinds = [2, 3, 4, 5, 6, 7, 8, 9, 10, 'Jack', 'Queen', 'King',
'Ace'];
    const suits = ['Diamonds', 'Hearts', 'Spades', 'Clubs'];
    const deck = [];
```

We will begin by just making a deck of file names. Once we get that to work, we will upgrade the deck so that each card is an object with several properties.

8. Set up a nested loop, where the outer loop iterates over the **kinds** array, and the inner loop iterates over **suits**:

```
    for (let i = 0; i < kinds.length; i++) {
        for (let j = 0; j < suits.length; j++) {
        }
    }
```

The outer loop runs 13 times, and the inner loop runs four times, for a total of 52 iterations, one for each card.

9. In the inner loop, concatenate the card file name, and push the result into the deck:

```
        let cardFileName = `${kinds[i]}-of-${suits[j]}.png`;
        deck.push(cardFileName);
```

10. After the loop ends, log the deck to see what we get.

```
    console.log(deck);
    /*
    ['2-of-Diamonds.png', '2-of-Hearts.png', '2-of-Spades.png',
    '2-of-Clubs.png', '3-of-Diamonds.png', '3-of-Hearts.png',
        -- ETC. --
    'King-of-Spades.png', 'King-of-Clubs.png', 'Ace-of-Diamonds.png',
'Ace-of-Hearts.png', 'Ace-of-Spades.png', 'Ace-of-Clubs.png']
    */
```

deck of cards as an array of objects

Let's run the nested loop again, but this time we will make each card an object, in which the file name is just one of several properties. The five properties will be:

- **name** ("2 of Diamonds", etc.)
- **file** ("2-of-Diamonds.png", etc.)
- **kind** (2, 3, 4..Jack, Queen, King, Ace)
- **suit** (Diamonds', 'Hearts', 'Spades', 'Clubs')
- **valu** (numeric value; face cards = 10, Ace = 11)

valu is for storing the numeric value of each card, from 1-11, using the blackjack scoring system, where aces start with a value of 11, face cards are worth 10 and the numbered cards are their respective numeric values.

11. Declare a new, empty array to hold our 52 card objects.

```
const deckOfCards = [];
```

12. Set up the nested loop;

```
for (let i = 0; i < kinds.length; i++) {
    for (let j = 0; j < suits.length; j++) {
    }
}
```

13. Simplify the current array items by passing them to variables:

```
for (let i = 0; i < kinds.length; i++) {
    for (let j = 0; j < suits.length; j++) {
        let kind = kinds[i];
        let suit = suits[j];
    }
}
```

14. Concatenate the card and image file names. The "Queen of Diamonds" has a file name of "Queen-of-Diamonds.png":

```
        let kind = kinds[i];
        let suit = suits[j];
        let name = `${kind} of ${suit}`;
        let file = `${kind}-of-${suit}.png`;
```

15. Declare a variable, **valu**, with an inital value of 0:

```
        let name = `${kind} of ${suit}`;
        let file = `${kind}-of-${suit}.png`;
        let valu = 0;
```

16. Run conditional logic to set the **valu** property. Only "Jack", "Queen" and "King" have more than three characters, which is what **(kind.length > 3** checks for:

```
        let valu = 0;
        if(kind == 'Ace')  {
            valu = 11;
        } else if(kind.length > 3) {
            valu = 10;
        } else {
```

```
                valu = kind;
            }
```

17. Declare an object called **card**. Its properties equal the variables we have made:

```
            let card = { name: name, file: file, kind: kind,
                         suit: suit, valu: valu };
```

18. Push the card object into the **deckOfCards** array:

```
            deckOfCards.push(card);
```

19. Below the nested loop, output the array of 52 objects:

```
    console.log(deckOfCards);
    /*
    0: {name: '2 of Diamonds', file: '2-of-Diamonds.png', kind: 2, suit:
'Diamonds', valu: 2}
    51: {name: 'Ace of Clubs', file: 'Ace-of-Clubs.png', kind: 'Ace',
suit: 'Clubs', valu: 11}
    */
```

ouputting cards to the DOM

Let's deal a hand of 5-card poker. Clicking the DEAL CARDS button will:

- call a function called **dealCards**
- the function will get all five of the card img tags from the DOM
- the function will run a loop five times, once for each card image
- each iteration of the loop will generate a random int from 0-51
- the random integer will be used to get a card by its array index
- the item's file name will be concatenated into an image path
- the img object will have its src property set to the image path

This will be the simplest implementation possible, meaning that there will only be one player hand dealt and all five cards will appear at once, without the benefit of a realistic-looking time delay between cards.

20. Get the button and tell it to run the function when clicked:

```
    const btn = document.querySelector('button');
    btn.addEventListener('click', dealCards);
```

21. Get the five images. The **querySelectorAll** method will get all img tags and make an array of them called a Node List:

```
const imgArr = document.querySelectorAll('img');
```

22. Now for the function, which loops through the array of images:

```
function dealCards() {
    for (let i = 0; i < imgArr.length; i++) {
    }
}
```

23. Each time the loop runs, generate a random number from 0-51:

```
function dealCard() {
    for (let i = 0; i < imgArr.length; i++) {
        let r = Math.floor(Math.random() * deckOfCards.length);
    }
}
```

24. Get a random card from the deck by index:

```
    for (let i = 0; i < imgArr.length; i++) {
        let r = Math.floor(Math.random() * deckOfCards.length);
        let card = deckOfCards[r];
    }
```

25. Set the src of the image in the imgArray by concatenating the file pate using the file property of the card object:

```
        let r = Math.floor(Math.random() * deckOfCards.length);
        let card = deckOfCards[r];
        imgArr[i].src = "images/" + card.file;
```

26. Reload the page and click the DEAL CARDS button. Five new cards should appear.

If you keep clicking the DEAL CARDS button, you will notice a bug: the same card showing up more than once in the same hand. To fix this, remove cards from the deck as they are dealt.

27. Using the splice() method, remove the dealt card from the deck:

```
            let card = deckOfCards[r];
            imgArr[i].src = "images/" + card.file;
            deckOfCards.splice(r, 1);
```

Now we have a new bug: after 10 hands we have used 50 cards and are down to our last two. To fix this, we can use conditional logic to replenish the deck once the supply runs too low. To do this, we need to make a "backup" -- a copy -- of the original deck.

28. Outside of the function, copy the array using **slice(0)**, which returns a copy from index 0 to the end of the array:

```
    let deckCopy = deckOfCards.slice(0);
```

29. Comment out the dealCards function, and rewrite it. Start with an if statement that makes a fresh copy if there are fewer than 5 cards left:

```
    function dealCard() {

        if(deckCopy.length < 5) {
            deckCopy = deckOfCards.slice(0);
        }

    }
```

30. Switch to using deckCopy throughout; after each hand, log how many cards are left, so you can see when the card supply is replenished:

31. Rerun the page. Now you should be able to deal hands indefinitely, without ever running out of cards.

END: Lesson 05.04 NEXT: Lesson 05.05

UNIT 05
LESSON 05.05

Passing Arrays to Functions

In this lesson, we will be writing functions that take arrays as their arguments.

function #1: add article in front of word

The first function will take an array of words and add the article "a" or "an" in front of each word. If the word starts with "a", "e", "i" or "o", it should start with "an". The final product ("an apple", "a banana", etc.) will be saved to a new array.

The idea is that we can pass in any array of strings and the function will produce the same result, so we start by declaring two arrays:

1. Declare two arrays of strings:

```
    const fruits = ['apple', 'apricot', 'banana', 'blueberry',
'blueberry', 'cherry', 'elderberry', 'grape', 'kiwi', 'lemon', 'lime',
'mango', 'orange', 'papaya', 'peach', 'pear', 'pineapple', 'plum',
'raspberry', 'strawberry', 'tangerine', 'watermelon'];

    const cars = ['Acura', 'Alfa Romeo', 'Audi', 'BMW', 'Chevrolet',
'Dodge', 'Edsel', 'Eagle', 'Fiat', 'Ford', 'Honda', 'Kia', 'Infiniti',
'Isuzu', 'Mazda', 'Nissan', 'Opel', 'Porsche', 'Rolls Royce', 'Tesla',
'Toyota', 'Volvo'];
```

2. Declare the function and assign it a parameter, **arr**:

```
    function addArticle(arr) {
    }
```

3. The function needs a few variables:

- an empty array to hold the results
- a variable set equal to the article followed by a space
- a variable for checking if a word starts with a, e, i, o

```
```

```
function addArticle(arr) {
    let newArr = [];
    let article = 'a ';
    let vowels = 'aeio';
}
```

4. The function needs to check each item in the array, so set up a loop that iterates the array:

```
function addArticle(arr) {
    let newArr = [];
    let article = 'a ';
    let vowels = 'aeio';

    for(let i = 0; i < arr.length; i++) {
    }
}
```

5. Each time through the loop, save the current item, as well as the first letter of that item:

```
function addArticle(arr) {
    let newArr = [];
    let article = 'a ';
    let vowels = 'aeio';

    for(let i = 0; i < arr.length; i++) {
        let word = arr[i]; // apple
        let firstChar = word[0]; // a
    }

}
```

6. Add an if-statement that uses the **string.includes()** method to check if the first character, set to lowercase, is "a", "e", "i" or "u". If it is, set the value of **article** to "an":

```
    for(let i = 0; i < arr.length; i++>) {
        let word = arr[0]; // apple
        let firstChar = word[0]; // a
        if(vowels.includes(firstChar)) {
            article = 'an ';
        }
    }
```

7. Just for practice, comment out the if-else and switch to a ternary expression:

```
vowels.includes(firstChar) ? article='an ' : article='a ';
```

8. Concatenate **article** with the original word the original word, **arr[i]**, which may be capitalized. Push it into newArr, and return it:

```
    for(let i = 0; i < arr.length; i++>) {
        let word = arr[0]; // apple
        let firstChar = word[0]; // a
        if(vowels.includes(firstChar)) {
            article = "an ";
        }
        newArr.push(article + word);
    }
    return newArr;
```

Be sure to push *inside* the loop and return *outside* the loop. If the **return** is inside the loop, the loop will only run once, because **return** always ends a function.

9. Call the function twice, passing in a different array of strings each time. Save the return values to variables and log them:

```
let fruitsArr = addArticle(fruits);
console.log(fruitsArr);

let carsArr = addArticle(cars);
console.log(carsArr);
```

function #2: find target number in array

This next function has not just an array parameter, but also a number param. The function loops through the array, looking for the number.

- If the number is found, the function returns the index.
- If the number is NOT found, the function returns -1.

10. Declare an array of numbers, followed by a function:

```
const numsArr = [5, 7, 9, 12, 14, 16, 20, 25, 30, 40, 50];
```

11. Write the function with two parameters.

```
function findNum(arr, num) {
}
```

12. Iterate the array with a for loop:

```
function findNum(arr, num) {
    for(let i = 0; i < arr.length; i++) {
    }
}
```

13. Check if the current array item equals the target number, and if it does, return the index:

```
function findNum(arr, num) {
    for(let i = 0; i < arr.length; i++) {
        if(arr[i] === num) {
            return i;
        }
    }
}
```

14. If the loop ends without finding the target number, return -1:

```
function findNum(arr, num) {
    for(let i = 0; i < arr.length; i++) {
        if(arr[i] === num) {
            return i;
        }
    }
    return -1;
}
```

15. Call the function. The function returns a value, so set the call equal to a variable to "capture" the return value:

```
let find9 = findNum(numsArr, 12);
console.log(find9);  // 3

let find8 = findNum(numsArr, 123);
console.log(find8);  // -1
```

16. Instead of just returning the index, let's return a string, specifying the number being looked for and where it was found. Do this as a new function:

```
function findNumber(arr, num) {
    for(let i = 0; i < arr.length; i++) {
        if(num === arr[i]) {
            return `${num} found at index ${i}`;
```

```
            }
        }
        return `${num} not found`;
    }

    let find25 = findNumber(numsArr, 25);
    console.log(find25);  // 25 found at index 7

    let find250 = findNumber(numsArr, 250);
    console.log(find250);  // 250 not found
```

function #3: check array to see if all numbers are even

Let's try one more function that takes an array for its argument. The function checks each number to see if it is odd or even.

- If all numbers are even, the function returns true.
- If an odd number or float are found, the function returns false.

17. Declare three arrays of numbers:

- The first array contains only even numbers.
- The second array contains an odd number; the rest are even.
- The third array contains a float; the rest are even integers.

```
    let nums1 = [2, 4, 6, 8, 10];
    let nums2 = [2, 4, 7, 8, 10];
    let nums3 = [2, 4, 6.78, 10];
```

18. Write a function that expects an array to be passed to it:

```
    function allEven(arr) {
    }
```

19. Have the function iterate over the array:

```
    function allEven(arr) {
        for(let i = 0; i < arr.length; i++) {
        }
    }
```

20. Each time through the loop, run an if statement that uses the modulo operator **%** to divide the current number by 2. If the reminder is *not equal to* **!=** 0, the number is *not* even, the number is *not* even, so return false:

```
function allEven(arr) {
    for(let i = 0; i < arr.length; i++) {
        if(arr[i] % 2 != 0) {
            return false;
        }
    }
}
```

21. If the loop ends without finding a non-even number, all numbers in the array must be even, so return true:

```
function allEven(arr) {

    for(let i = 0; i < arr.length; i++) {
        if(arr[i] % 2 != 0) { // if num is not even
            return false;
        }
    }
    return true;
}
```

22. Call the function three times, once for each array; save the return value to a variable and log it:

```
let allEven1 = allEven(nums);
console.log('allEven1', allEven1); // true

let allEven2 = allEven(nums);
console.log('allEven2', allEven1); // false

let allEven3 = allEven(nums);
console.log('allEven3', allEven1); // false
```

END Lesson 05.05 NEXT: Lab 05.05

Passing Arrays to Functions

1. Write a function that takes the nums array as its argument. The function returns the first odd number along with its index:

```
    const nums = [4, 6, 10, 12, 18, 21, 24, 28, 33, 36];
```

2. Write a function that takes the nums array as its argument. Find the value and position of the first even number.

3. Write a function that takes the nums array as its argument. Return an array that only contains those numbers that are multiples of 3:

4. Write a function that takes the nums array as its argument. Return an array called multis that has all the values from nums multiplied by its index. So, if the number at index 3 is 5, the new value to push into multis is 5 x 3.

5. Write a function that takes in the fruits array and returns an object with the following properties

- fruitCount (number of words in the array)
- totChars (number of characters in the array)
- totVowels (number of vowels in the array)
- initVowels (number of words that start with a vowel)
- berrries (number of berries) HINT: The returned object does not need a name. The function call should be set equal to a variable, and in this way, the object will be assigned its name.

```
    const fruits = ["apple", "apricot", "banana",
"blackberry","blueberry", "boysenberry", "cherry", "cranberry",
"dragonfruit", "elderberry"];

    // 6. Make the function work for any array, by passing in the veggies
array and returning the same object:

    const veggies = ["arugula", "broccoli", "carrot", "celery","cucumber",
"daikon", "eggplant", "iceberg lettuce", "onion", "radish"];
```

NEXT: Lesson 05.05 Solution

05.05 Lab Solution

Passing Arrays to Functions

```
const nums = [4, 6, 10, 12, 18, 21, 24, 28, 33, 36];
```

1. Write a function that takes the nums array as its argument. The function returns the first odd number along with its index:

```
function findFirstOddNum(arr) {

    for(let i = 0; i < arr.length; i++) {
        if(arr[i] % 2 == 1) { // if num is not even
            return arr[i];
        }
    }
    return 'no odd card found';

}

let firstOddity = findFirstOddNum(nums);
console.log('firstOddity:', firstOddity);
```

2. Write a function that takes the nums array as its argument. Find the value and position of the first even number.

```
function findFirstEvenNum(arr) {

    for(let i = 0; i < arr.length; i++) {
        if(arr[i] % 2 == 0) { // if num is even
            return `The first even number, ${arr[i]}, is at index
${i}.`;
        }
    }
    return 'no even number found';

}

let firstEven = findFirstEvenNum(cards);
console.log('firstEven:', firstEven);
```

3. Write a function that takes the nums array as its argument. Return an array that only contains those numbers that are multiples of 3:

```
    function collectMultiplesOf3(arr) {

    let multiplesOf3 = [];

    for(let i = 0; i < arr.length; i++) {
        if(arr[i] % 3 == 0) { // if num is divisible by 3
            multiplesOf3.push(arr[i]);
        }
    }
    return multiplesOf3;

    }

    let multiplesOf3 = collectMultiplesOf3(nums);
    console.log('multiplesOf3:', multiplesOf3);
```

4. Write a function that takes the nums array as its argument. Return an array called multis that has all the values from nums multiplied by its index. So, if the number at index 3 is 5, the new value to push into multis is 5 x 3.

```
    function multiplyNumberByItsIndex(arr) {

    let multis = [];

    for(let i = 0; i < arr.length; i++) {
        let m = arr[i] * i;
        multis.push(m);
    }
    return multis;

    }

    let multi = multiplyNumberByItsIndex(nums);
    console.log('multi:', multi);
```

5. Write a function that takes in the fruits array and returns an object with the following properties

- fruitCount (number of words in the array)
- totChars (number of characters in the array)
- totVowels (number of vowels in the array)
- initVowels (number of words that start with a vowel)
- berrries (number of berries) HINT: The returned object does not need a name. The function call should be set equal to a variable, and in this way, the object will be assigned its name.

```
    const fruits = ["apple", "apricot", "banana",
"blackberry","blueberry", "boysenberry", "cherry", "cranberry",
"dragonfruit", "elderberry"];
```

6. Make the function work for any array, by passing in the veggies array and returning the same object:

```
    const veggies = ["arugula", "broccoli", "carrot", "celery","cucumber",
"daikon", "eggplant", "iceberg lettuce", "onion", "radish"];

    function makeWordObj(fruits) {

    }
```

BONUS:

7. Make a function that takes in a big string, such as a page or chapter of text as its argument, and returns an object with each unique word as a key the value of which is the number of times the word occus in the passage. So if the passage is:

"It is often said that that is the way that the proverbial cookie crumbles." The returned object should be:

```
    { "It": 1, "is": 2, "often": 1, "said": 1, "that": 3, "the": 2, "way":
1, "proverbial": 1, "cookie": 1, "crumbles": 1 }
```

NEXT: Lesson 05.06

Classic Programming Job Interview Challenges:

A. Fizz-Buzz

B. Find nth Fibonacci Number

C. Check for duplicates in array

A. Fizz-Buzz

Output to the console all integers from 1-100 along with "Fizz" and/or "Buzz" according to the following rules:

- If the number is evenly divisible by both 3 and 5, output the number followed by ' Fizz-Buzz'.
- If the number is evenly divisible by 3 (but not 5), output the number followed by ' Fizz'.
- If the number is evenly divisible by 5 (but not 3), output the number followed by ' Buzz'.
- If the number is divisible by neither 3 nor 5, output just the number.

One aim of the challenge is to see if the candidate figures out that the **modulo** operator is the key to the solution. Before tackling "Fizz-Buzz", let's try an example of modulo (%), which returns the remainder of the first number divided by the second number:

```
let remainder = 17 % 5;
console.log('remainder', remainder); // 2
```

And now for "Fizz-Buzz"...

1. Set up a for loop that goes from 1-100:

```
for(let i = 1; i <= 100; i++) {
}
```

Another aim of the challenge is to see if the candidate can figure out to start by checking if the number is divisible by both and and 5, as opposed to starting by seeing if it's divisible by one or the other. This test requires the **&&** (AND) operator.

2. Use the **&&** operator to see if the current number yields a remainder of 0 when divided by 3 and 5:

```
for(let i = 1; i <= 100; i++) {
    if(i % 3 == 0 && i % 5 == 0) {
        console.log(i + ' Fizz BUzz');
    }
}
```

3. If the number is not divisible by both 3 and 5, check if it's divisible by 3 only. If it is, log the number followed by 'Fizz':

```
for(let i = 1; i <= 100; i++) {
    if(i % 3 == 0 && i % 5 == 0) {
        console.log(i + ' Fizz BUzz');
    } else if(i % 3 == 0) {
        console.log(i + ' Fizz');
    }
}
```

4. Add another *else if* to see if the number is divisible by just 5. If it is, log the number followed by 'Buzz':

```
for(let i = 1; i <= 100; i++) {
    if(i % 3 == 0 && i % 5 == 0) {
        console.log(i + ' Fizz BUzz');
    } else if(i % 3 == 0) {
        console.log(i + ' Fizz');
    } else if(i % 5 == 0) {
        console.log(i + ' Buzz');
    }
}
```

5. Finally, if the number is divisible by neither 3 nor 5, log just the number:

```
for(let i = 1; i <= 100; i++) {
    if(i % 3 == 0 && i % 5 == 0) {
        console.log(i + ' Fizz BUzz');
    } else if(i % 3 == 0) {
        console.log(i + ' Fizz');
    } else if(i % 5 == 0) {
        console.log(i + ' Buzz');
    } else {
        console.log(i);
    }
}
```

B. Find nth Fibonacci number

Another popular coding challenge is to produce a sequence of Fibonacci numbers. Part of the point is to see if the candidate even knows what a Fibonacci sequence is. The wording can vary, but is typically something like:

"Find the 20th number in the Fibonacci sequence."

In a Fibonacci sequence, each number is the sum of the previous two. A starter array **[0, 1]** is usually given. The sequence continues as: **[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, etc.]**.

In programming challenges, it is often required that solutions be implemented as a function, with any givens provided as arguments. Our function will be called **makeFibos** and will take two arguments: the starter array and **n**, the target, which is this case is 20.

The **makeFibos(arr, n)** function will do the following:

- iterate the array, which to start off is just **[0, 1]**; each iteration adds a new item to keep the loop going
- our loop runs **n-2** (18) times, because we have a 2-item head start
- each time the loop runs, get the current item: **fibos[i]**
- we also need the next item: **fibos[i+1]**, which is also the last item
- add the numbers: **fibos[i] + fibos[i+1]**. The sum is the next number in the sequence.
- **push()** the sum into the array. The array just got longer by 1, so the loop can run one more time. This repeats until the loop has run **n-2** (18) times.
- when the loop is over, log the last item of the 20-item array, which is the item at index **n-1** (19).

6. Start by declaring an array, **fibosArr**, containing the two starter values. Also declare a number variable to store the target, which is 20:

```
const fibosArr = [0, 1];
let n = 20;
```

In the interview challenge, the givens will not likely be provided as variables; you have to declare them, as we just did.

7. Set up the function with parameters **arr** and **n**:

```
function makeFibos(fibos, n) {
}
```

8. Set up a loop that runs **n-2** times:

```
function makeFibos(fibos, n) {
    for(let i = 0; i < n-2; i++) {
    }
}
```

9. Add the current and next item, which are the last two numbers: **fibos[i] + fibos[i+1]**. Push the sum into the array:

```
function makeFibos(fibos, n) {
    for(let i = 0; i < n-2; i++) {
        let nextFibo = fibos[i] + fibos[i+1];
        fibos.push(nextFibo);
    }
}
```

10. Return the answer. Console log the full array inside the function as a test, and return the last item, which is the nth (20th) item:

```
function makeFibos(fibos, n) {
    for(let i = 0; i < n-2; i++) {
        let nextFibo = fibos[i] + fibos[i+1];
        fibos.push(nextFibo);
    }
    console.log(fibos);
    return fibos[fibos.length-1];
}
```

11. Call the function, saving the call to a variable which captures the return value:

```
let fibo20th = makeFibos(fibosArr, n);
```

12. Log the return value, which is the 20th fibo:

```
console.log(fibo20th);
```

One refinement: to make the loop more dynamic, we should not hard-code **n-2** (18). After all, what if the provided starter array is **[0, 1, 1, 2]**? Then we would need to run the loop **n-4** (16) times to get to the 20th item. The answer is to save **n-arr.length** to a variable above the loop.

13. Above the loop, save the target **n** minus the length of the array argument to a variable, **loopTimes**. This lets the starter array can have more than two items:

```
function makeFibos(fibos, n) {
    let loopTimes = n-fibos.length;
    for(let i = 0; i < loopTimes; i++) {
        let nextFibo = fibos[i] + fibos[i+1];
        fibos.push(nextFibo);
    }
```

```
        console.log(fibos);
        return fibos[fibos.length-1];
    }
```

14. Make a new starter array with a few more "fibos" in it:

```
    const fibos = [0, 1];
    let n = 20;
```

15. Call the function, passing in the five-item starter array, saving the return value to a new variable:

```
    let twentiethFibo = makeFibos(fiveFibos, n);
    console.log(twentiethFibo);
```

C. Find duplicate items in an array

"Given an array of numbers or strings, find all duplicates and return it/them as a new array containing one instance of each duplicate. If there are no duplicates, return 'none'"

The solution function takes an array as its argument. An example array may be given, but make sure your function works on any array of numbers or strings.

16. Assuming **nums1** is given, declare three more arrays, to give us more testing scenarios:

```
    const nums1 = [2, 4, 5, 7, 8, 9, 4, 11, 6, 8, 7, 9, 11, 6, 7];
    const nums2 = [2, 4, 5, 7, 1, 9, 11, 3, 6, 8, 10];
    const fruits1 = ['apple', 'banana', 'apple', 'cherry', 'kiwi',
'banana', 'mango', 'pear', 'peach', 'kiwi', 'grape'];
    const fruits2 = ['apple', 'banana', 'cherry', 'mango'];
```

17. Define a function that takes an array as its argument:

```
    function findDuplicates(arr) {
    }
```

18. Iterate the array:

```
    function findDuplicates(arr) {
        for(let i = 0; i < arr.length; i++) {
        }
    }
```

The key to the solution is to keep track of which values have come up so far. We can do this by saving each unique item as a key of an object, which we declare inside the function.

- check if the current array item is already an object key.
- if the current item is already a key, it is a duplicate.
- if the current item is not yet a key, make a key for it. The syntax uses square brackets for both array index and dynamic object property accessor.

19. Define an object above the loop. Inside the loop, check if the current array item is **not !** a key. If it is not, make a key for that item:

```
function findDuplicates(arr) {
    let obj = {};
    for(let i = 0; i < arr.length; i++) {
        if(!obj[arr[i]]) {
            obj[arr[i]] = arr[i];
        }
    }
}
```

20. But if the current array item, **arr[i]** is already a key, we have a duplicate item, so push it into the array of duplicates. Also, declare a new empty array, **dups**, above the loop for storing the duplicates:

```
function findDuplicates(arr) {
    let obj = {};
    let dups = [];
    for(let i = 0; i < arr.length; i++) {
        if(!obj[arr[i]]) {
            obj[arr[i]] = arr[i];
        } else {
            dups.push(arr[i]);
        }
    }
}
```

avoid looking up array items again and again

Here's a refinement that interviewers will be sure to appreciate. The function has to repeatedly look up the value of the current array item (**arr[i]**). More memory-efficient would be to save **arr[i]** to a variable and just refer to the variable. This also makes the code cleaner by avoiding nested square brackets: **obj[arr[i]]**.

21. Above the if-else, save arr[i] to a variable::

```
function findDuplicates(arr) {
    let obj = {};
    let dups = [];
    for(let i = 0; i < arr.length; i++) {
```

```
            let item = arr[i];
            if(!obj[item]) {
                obj[item] = item;
            } else {
                dups.push(item);
            }
        }
    }
```

22. After the loop ends, check if the **dups** array has any items it it. If it doesn't, return the array. Else, return "none":

```
    function findDuplicates(arr) {
        let obj = {};
        let dups = [];
        for(let i = 0; i < arr.length; i++) {
            if(!obj[item]) {
                obj[item] = item;
            } else {
                dups.push(item);
            }
        }
        if(dups.length > 0) {
            return dups;
        } else {
            return "none";
        }
    }
```

23. Call the function four times, once per array. Save the function call to a variable to store the return value, and log the variable:

```
    let n1 = findDuplicates(nums1);
    console.log('duplicates:', n1);

    let n2 = findDuplicates(nums2);
    console.log('duplicates:', n2);

    let f1 = findDuplicates(fruits1);
    console.log('duplicates:', f1);

    let f2 = findDuplicates(fruits2);
    console.log('duplicates:', f2);
```

24. We are getting duplicates of duplicates, such as: ['apple', 'kiwi', 'banana', 'kiwi']). Let's add logic so that the duplicate gets pushed into the array only if it is *not* already there:

```
    } else {
        if(!dups.includes(item)) {
            dups.push(item);
        }
    }
```

- **END: Lesson 05.06**
- **NEXT: Lesson 05.07**

Calculator Application

Assigning Click Event Listeners on a Loop

Working with Math Operator Buttons

This Calculator application is a simplified version of a calculator.

Here's what the calculator can do:

- Add, subtract, multiply or divide 2 numbers

Here's what the calculator cannot do:

- Perform any other math operations
- Calculate with more than 2 numbers
- Store answers (all is reset when you click "clear")

The user interface consists of buttons that call functions. The functions are called in order, as follows

I. onNumberClick()

- The user clicks a digit (0-9) or the decimal point to call the **onNumberClick()** function, which:
  - concatenates the digit (or decimal point) as **numStr**
  - updates the display in the ouput box.

II. onOperatorClick()

- The user clicks an operator (+,-,*,/) to call the **onOperatorClick()** function, which:
  - converts **numStr** to **num**, an actual number
  - pushes **num** into the **nums** array
  - saves the operator to a variable, **oper**
  - updates the ouput box to include the operator

III. onNumberClick() again

- The user inputs the second number, which again calls **onNumberClick()**

IV. calculateAnswer()

- After entering the second number, the user clicks the equal sign (**=**) to call the **calculateAnswer()** function, which:
  - pushes the second number into the **nums** array

    - runs a series of if-statements, to determine what the operator is and to perform the correct calculation:
        - If operator is **'+'**, add the two numbers
        - If operator is **'-'**, subtract the second number from the first
        - If operator is **'*'**, multiply the two numbers
        - If operator is **'/'**, divide the first number by the second
    - updates the ouput box to include the answer

V. clearBox()

- Clicking **clr** (clear) calls **clearBox()** function, which:
    - empties the box
    - resets the variables

output box NOT directly editable
To reduce the amount of code required for this application, the output box has been intentionally *not* been made editable. Since the user cannot click inside the box and type, the only way to enter content into the box is by clicking the buttons.

numbers cannot be erased / deteted
Also to reduce the amount of code required for this application, there is no delete button to remove content or change the inputted numbers.

1. Open the file **05.07-Calculator.html**, and take it for a spin in the browser:

- enter a number (as many digits as you like)
- click one of the four operators: *+, -, , /
- enter another number
- click the **=** (equals) sign to get the answer
- click **clear** to reset

2. Have a look at the tags in the html file. Note that the "buttons" are actually divs:

```
    <section>
        <div id="0" class="num-btns">0</div>
        <div id="1" class="num-btns">1</div>
        <div id="2" class="num-btns">2</div>
        <div id="3" class="num-btns">3</div>
        <div id="4" class="num-btns">4</div>
        <div id="5" class="num-btns">5</div>
        <br>
    </section>

    <section>
        <div id="6" class="num-btns">6</div>
        <div id="7" class="num-btns">7</div>
        <div id="8" class="num-btns">8</div>
        <div id="9" class="num-btns">9</div>
        <div id="." class="num-btns">.</div>
        <div id="=" class="equals-btn">=</div>
    </section>
```

```
    <section>
        <div id="+" class="oper-btns">+</div>
        <div id="-" class="oper-btns">-</div>
        <div id="*" class="oper-btns">*</div>
        <div id="/" class="oper-btns">/</div>
        <div id="c" class="clear-btn">clear</div>
    </section>

    <div id="num-box">
        <!-- output appears here -->
    </div>
```

- The **num-btns** class occurs 10 times, once for each digit.
- The **oper-btns** class occurs 4 times, once for each operator.
- The **delete-btn**, **clear-btn** and **equals-btn** class occurs once each.

3. In the script tags, comment out the **FINAL.js** file and reactivate **START.js**.

4. Our task is to recreate **FINAL.js** from scratch, so open **05.07-Calculator-START.js**

We start by bringing in the "buttons". We will use **document.querySelectorAll()** to bring in multiple elements at once. These come in as an array of objects, known as a **Node List**.

5. Get the divs that have a class of **num-btns**. These are the digits from 0-9:

```
    const numBtnsArr = document.querySelectorAll('.num-btns');
```

6. Loop through the **numBtnsArr** array and assign event listeners to the objects:

```
    for(let i = 0; i < numBtnsArr.length; i++) {
        numBtnsArr[i].addEventListener('click', onNumberClick);
    }
```

7. Get the operator divs (+,-,*,/), which have a class of **oper-btns**:

```
    const operBtnsArr = document.querySelectorAll('.oper-btns');
```

8. Loop through the **operBtnsArr** arrays, adding listeners to each item:

```
    for(let i = 0; i < operBtnsArr.length; i++) {
        operBtnsArr[i].addEventListener('click', onOperatorClick);
    }
```

9. Get the other three "buttons" which call functions; these are the **=** (equals) and **clear** buttons:

```
const equalsBtn = document.querySelector('.equals-btn');
equalsBtn.addEventListener('click', calculateAnswer);

const deleteBtn = document.querySelector('.delete-btn');
deleteBtn.addEventListener('click', delete);

const clearBtn = document.querySelector('.clear-btn');
clearBtn.addEventListener('click', clearBox);
```

10. Get the **num-box**, which is the div for the output:

```
const numBox = document.getElementById('num-box');
numBox.textContent = "";
```

11. Next, declare several global variables for use by the functions:

- **numStr**: for storing the inputted digit(s) as a string
- **nums**: an array for storing the two **numStr** values
- **num**: the numeric version of **numStr**, for doing the math
- **oper**: a string for storing the operator symbol
- **answer**: for storing the calculated result

```
let numStr = "";
let num = 0;
let nums = [];
let oper = '';
let answer = 0;
```

12. Write the **onNumberClick()** function, which runs on click of any digit from 0-9 or the decimal point.

The **onNumberClick()** function concatenates the clicked digit onto **numStr**.

- The keyword **this** in a function always refers to the object which called the function, so **this.id** is the digit
- If the user clicks 1, 2, 3 in order, then **numStr** equals "123"
- The value of **numStr** is displayed in the box

```
function onNumberClick() {
    numStr += this.id; // concatenate the clicked digit
    numBox.textContent += this.id; // updates the output box
}
```

13. Write the **onOperatorClick()** function, which runs when an operator "button" is clicked:

- saves **this.id** (the clicked object id) to the **oper** variable
- pushes **numStr**, the second "number-like string", into the **nums** array
- resets **numStr** to make way for the second number
- outputs the operator, surrounded by spaces, to the box

```
function onOperatorClick() {
    oper = this.id;
    nums.push(numStr);
    numStr = '';
    numBox.textContent += ' ' + oper + ' ';
}
```

The user inputs the second of the two numbers. With each digit or decimal click, the **onNumberClick()** function is called, which concatenate the second "number-like string", saves it to **nums** and displays it in the box.

When the user is done inputting the second number, they click the equal sign, which calls the **calculateAnswer()** function.

14. Write the **calculateAnswer()** function, which:

- pushes the second "number-like string" into the **nums** array
- converts the two array items to actual numbers, using the **Number()** method and saves the results as variables, **num1** and **num2**
- runs a series of if-statements to identify the operator and does the correct mathematical calculation based on the operator
- runs an if-else-statement, which passes the **answer** to the **Number.isInteger()** method, which returns true if the answer is an integer. Else, the answer is a float, which gets passed to the **toFixed()** method, which rounds it to 7 decimal places
- The **answer** is outputted, preceded by **' = '**

```
function calculateAnswer() {
    nums.push(numStr);
    let num1 = Number(nums[0]);
    let num2 = Number(nums[1]);
    if(oper == '+') answer = num1 + num2;
    if(oper == '-') answer = num1 - num2;
    if(oper == '*') answer = num1 * num2;
    if(oper == '/') answer = num1 / num2;
    if(Number.isInteger(answer)) {
        numBox.textContent += ' = ' + answer;
    } else { // not an integer
        numBox.textContent += ' = ' + answer.toFixed(7);
    }
}
```

15. Write the **clearBox()** function, which empties the output box and resets the global variables. The calculator is ready for fresh input.

```
function clearBox() {
    numBox.textContent = '';
    nums = [];
    num = 0;
    oper = '';
    total = 0;
    numStr = '';
}
```

END: Lesson 05.07
NEXT: Lesson 05.08

Word Cloud

Making a Word Frequency Map Object

Word Frequency Map for a Word Cloud

- A Word Cloud is a visual representation of the frequency of words in a string, such as found in a blog post or other article.
- In a word cloud, the most frequently occurring words appear in the largest font size in order to do this, the frequency of words must be known.
- Therefore, before we can make a Word Cloud, we have to make what is called a Word Frequency Map from of the words. This takes the form of an Object, where the keys are unique words and the value of each key is the number of times the word occurs.
- To make the Word Frequency Map, we have to convert the text to an array, with each item a word.
- Then we loop through the array of words. every time a unique word is found, the object is assigned that word as a new key with an initial value of 1.
- The next time the word is encounterd, no new key is made, but rather the value of the existing key is incremented by 1.
- The resulting Word Frequency Map can then be used to make a Word Cloud by setting the font size of each word based on the frequency, with most frequent words biggest also
- A Word Cloud only contains interesting keywords, so there needs to be a filter that prevents what are known as **stopwords** from being included in the Word Frequency Map. Stopwords include such common words as **'the', 'and', 'of', 'on', 'with'**, etc.

1. Open **stopwords.js** and have a look at the array of stopwords. These are the words that must be excluded from our Word Cloud.

2. Define a function called **makeWordFreqMap** with two parameters: **str** and **stopwords**. The **str** will be an entire passage, such as an article, story or blog post.

```
function makeWordFreqMap(str, stopwords) {
}
```

3. Call the **trim()** method on the string. This removes all empty, extra spaces. Also chain on **toLowerCase()** so that "Land" and "land" are not regarded as different words:

```
function makeWordFreqMap(str, stopwords) {
    str = str.trim().toLowerCase();
```

```
    }
```

4. Remove punctuation so that "world." and "world" are not saved as separate keys. You can individually target punctuation markes, or, better, use **RegEx** (Regular Expressions):

```
        // str = str.replace(",", "");
        // str = str.replace(".", "");
        // str = str.replace(";", "");
        // str = str.replace(":", "");
        // str = str.replace("?", "");
        // str = str.replace("!", "");
        // or use fancy RegEx move to strip all non-alphanumeric
characters
        str = str.replace(/[^\w\s]|_/g, "").replace(/\s+/g, "
").replace(/[0-9]/g, '');
```

- **RegEx gibberish decoded**
- **\w** is any digit, letter, or underscore.
- **\s** is any whitespace.
- **[^\w\s]** is anything that's not a digit, letter, whitespace, or underscore.
- **[^\w\s]|_** is the same as #3 except with the underscores added back in.
- **[0-9]** is all digits
- **/g** is globally replace (everywhere)
- **str = str.trim()** get rid of any extra whitespace that still may remain

5. Use the **split()** method to make an array from all the words in **str**:

```
        let arr = str.split(" ");
```

6. Declare a new object. This is for storing key-value pairs of word frequencies:

```
        let obj = {};
```

7. Loop through the array of words made from the big string:

```
        for(let i = 0; i < arr.length; i++) {
        }
```

8. Inside the loop, save the current array item as **word**:

```
            let word = arr[i];
```

9. Use conditional logic to see if the current word is NOT included in the stopwords array:

```
            if(!stopwords.includes(word)) { // if the current
             }
```

10. If the current word is not already an object key, make it a key, with an initial value of 1; else increment the value of the word key by 1. A ternary is ideal for this. This ends the loop:

```
                !obj[word] ? obj[word] = 1 : obj[word]++;

            } // end if-else
        } // end for loop
```

11. Return the object. This is the Word Cloud / Word Frequency Map. This ends the function

```
            return obj; // output the function
    } // end function
```

12. Call the function twice, each time passing in a separate story, imported into the html page:

```
    let fairyTaleWordFreq = makeWordFreqMap(textPassage, stopwords);
    console.log('fairyTaleWordFreq:', fairyTaleWordFreq);

    let treehouseWordFreq = makeWordFreqMap(treehouse, stopwords);
    console.log('treehouseWordFreq:', treehouseWordFreq);
```

END: Lesson 05.08
NEXT: Lesson 06.01