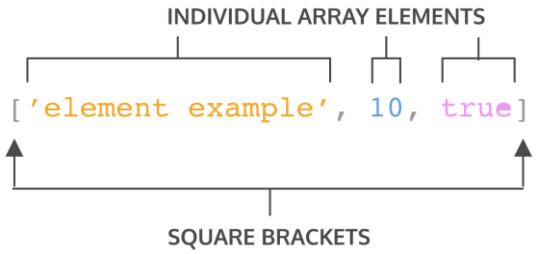
Contents

1.0 Arrays

1.1 Create an Array

One way we can create an array is to use an *array literal*. An array literal creates an array by wrapping items in square brackets []. Remember from the previous exercise, arrays can store any data type — we can have an array that holds all the same data types or an array that holds different data types.

Let's take a closer look at the syntax in the array example:



- The array is represented by the square brackets [] and the content inside.
- Each content item inside an array is called an *element*.
- There are three different elements inside the array.
- Each element inside the array is a different data type.

We can also save an array to a variable. You may have noticed we did this in the previous exercise:

```
let newYearsResolutions = ['Keep a journal', 'Take a falconry class', 'Learn to juggle'];
```

Instructions:

1. Declare a variable using `const` named `hobbies` and set it equal to an array with three strings inside of it.
2. Use `console.log()` to print `hobbies` to the console.

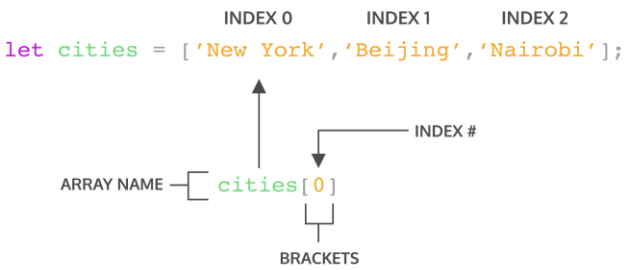
Solution:

```
const hobbies = ["hello", "hi", "bye"];
console.log(hobbies)
```

1.2 Accessing Elements

Each element in an array has a numbered position known as its *index*. We can access individual items using their index, which is similar to referencing an item in a list based on the item's position.

Arrays in JavaScript are *zero-indexed*, meaning the positions start counting from 0 rather than 1. Therefore, the first item in an array will be at position 0. Let's see how we could access an element in an array:



In the code snippet above:

- cities is an array that has three elements.
- We're using bracket notation, [] with the index after the name of the array to access the element.
- cities[0] will access the element at index 0 in the array cities. You can think of cities[0] as accessing the space in memory that holds the string 'New York'.

You can also access individual characters in a string using bracket notation and the index. For instance, you can write:

```
const hello = 'Hello World';
console.log(hello[6]);
// Output: W
```

The console will display W since it is the character that is at index 6.

Instructions:

1. Individual elements in arrays can also be stored to variables.
   Create a variable named listItem and set it equal to the first item in the famousSayings array using square bracket notation ([]).Then use console.log() to print the listItem variable to the console.
2. Now, console.log() the third element in the famousSayings array using bracket notation to access the element. Do not save the element to a new variable before you log it.

Solution:

```
const famousSayings = ['Fortune favors the brave.', 'A joke is a very serious thing.', 'Where there is love there is life.'];

const listItem = famousSayings[0];
// create a variable and set equal to the first item in the array using square bracket notation.
console.log(famousSayings);
// Print the variable to the console
console.log(famousSayings[2])
// Print the third element of the array
console.log(famousSayings[3])
```

1.3 Update Elements

In the previous exercise, you learned how to access elements inside an array or a string by using an index. Once you have access to an element in an array, you can update its value.

```
let seasons = ['Winter', 'Spring', 'Summer', 'Fall'];

seasons[3] = 'Autumn';
console.log(seasons);
//Output: ['Winter', 'Spring', 'Summer', 'Autumn']
```

In the example above, the `seasons` array contained the names of the four seasons. However, we decided that we preferred to say `'Autumn'` instead of `'Fall'`. The line, `seasons[3] = 'Autumn';` tells our program to change the item at index 3 of the `seasons` array to be `'Autumn'` instead of what is already there.

Solution:

```
let groceryList = ['bread', 'tomatoes', 'milk'];


groceryList[1] = "avocados";
// change the second elemeny of the array
```

1.4 Arrays with let and const

You may recall that you can declare variables with both the `let` and `const` keywords. Variables declared with `let` can be reassigned.

Variables declared with the `const` keyword cannot be reassigned. However, elements in an array declared with `const` remain **mutable.** Meaning that we can change the contents of a `const` array, but cannot reassign a new array or a different value.

The instructions below will illustrate this concept more clearly. Pay close attention to the similarities and differences between the `condiments` array and the `utensils` array as you complete the steps.

Solution:

```
let condiments = ['Ketchup', 'Mustard', 'Soy Sauce', 'Sriracha'];

const utensils = ['Fork', 'Knife', 'Chopsticks', 'Spork'];

condiments[0] = "Mayo";
// Re-assign the element in the array
console.log(condiments);
// print the updated to the console

condiments = ["Mayo"];
// Re-assign new array with only "Mayo" string
console.log(condiments);
// Log the result to the console
utensils[3] = "Spoon";
// Re-assign the last item to "Spoon"
console.log(utensils);
// log the updated array to the console
```

1.5 The .length property

One of an array's built-in properties is `length` and it returns the number of items in the array. We access the `.length` property just like we do with strings. Check the example below:

```
const newYearsResolutions = ['Keep a journal', 'Take a falconry class'];

console.log(newYearsResolutions.length);
// Output: 2
```

In the example above, we log newYearsResolutions.length to the console using the following steps:

- We use *dot notation*, chaining a period with the property name to the array, to access the length property of the newYearsResolutions array.
- Then we log the length of newYearsResolution to the console.
- Since newYearsResolution has two elements, so 2 would be logged to the console.

When we want to know how many elements are in an array, we can access the .length property.

Solution:

```
const objectives = ['Learn a new languages', 'Read 52 books', 'Run a marathon'
];

console.log(objectives.length);
// Find how many elements in the array and log it to the console
// Output: 3
```

1.6 The .push() Method

Let's learn about some built-in JavaScript methods that make working with arrays easier. These methods are specifically called on arrays to make common tasks, like adding and removing elements, more straightforward.

One method, .push() allows us to add items to the end of an array. Here is an example of how this is used:

```
const itemTracker = ['item 0', 'item 1', 'item 2'];

itemTracker.push('item 3', 'item 4');

console.log(itemTracker);
// Output: ['item 0', 'item 1', 'item 2', 'item 3', 'item 4'];
```

So, how does .push() work?

- We access the push method by using dot notation, connecting push to itemTracker with a period.
- Then we call it like a function. That's because .push() is a function and one that JavaScript allows us to use right on an array.

- `.push()` can take a single argument or multiple arguments separated by commas. In this case, we're adding two elements: `'item 3'` and `'item 4'` to `itemTracker`.
- Notice that `.push()` changes, or *mutates*, `itemTracker`. You might also see `.push()` referred to as a *destructive* array method since it changes the initial array.

If you're looking for a method that will mutate an array by adding elements to it, then `.push()` is the method for you!

Solution:

```
const chores = ['wash dishes', 'do laundry', 'take out trash'];


chores.push("clean the bedroom", "wiping the countertop");
// Add two elements to the end of the array using .push()
console.log(chores);
// Log the result
```

1.7 The .pop() Method

Another array method, `.pop()`, removes the last item of an array.

```
const newItemTracker = ['item 0', 'item 1', 'item 2'];

const removed = newItemTracker.pop();

console.log(newItemTracker);
// Output: [ 'item 0', 'item 1' ]
console.log(removed);
// Output: item 2
```

- In the example above, calling `.pop()` on the `newItemTracker` array removed `item 2` from the end.
- `.pop()` does not take any arguments, it simply removes the last element of `newItemTracker`.
- `.pop()` returns the value of the last element. In the example, we store the returned value in a variable `removed` to be used for later.
- `.pop()` is a method that mutates the initial array.

When you need to mutate an array by removing the last element, use `.pop()`.

Solution:

```
const chores = ['wash dishes', 'do laundry', 'take out trash', 'cook dinner',
'mop floor'];

chores.pop()
// remove the last element from chores
console.log(chores)
// log the result
```

1.8 More Array Methods

There are many more array methods than just .push() and .pop(). You can read about all of the array methods that exist on the Mozilla Developer Network (MDN) array documentation.

.pop() and .push() mutate the array on which they're called. However, there are times that we don't want to mutate the original array and we can use non-mutating array methods. Be sure to check MDN to understand the behaviour of the method you are using.

Some arrays methods that are available to JavaScript developers include: .join(), .slice(), .splice(), .shift(), .unshift(), and .concat() amongst many others. Using these built-in methods make it easier to do some common tasks when working with arrays.

Below, we will explore some methods that we have not learned yet. We will use these methods to edit a grocery list. As you complete the steps, you can consult the MDN documentation to learn what each method does!

Solution:

```
const groceryList = ['orange juice', 'bananas', 'coffee beans', 'brown rice',
'pasta', 'coconut oil', 'plantains'];

groceryList.shift();
console.log(groceryList);
// Remove the element to the beginning of the array and log the result
groceryList.unshift("popcorn");
console.log(groceryList);
// Adds the elements to the beginning of the array
console.log(groceryList.slice(1, 4));
// Use .slice() to return selected elements as a new array
console.log(groceryList);
```

```
// Log the result to see .slice() is non-
mutating as the array contains the old items

const pastaIndex = groceryList.indexOf("pasta");
// The indexOf() method returns the first index at which a given element can b
e found in the array, or -1 if it is not present.

console.log(pastaIndex);
// Output: 4
```

1.9 Arrays and Functions

Throughout the lesson we went over arrays being mutable, or changeable. Well what happens if we try to change an array inside a function? Does the array keep the change after the function call or is it scoped to inside the function?

Take a look at the following example where we call `.push()` on an array inside a function. Recall, the `.push()` method mutates, or changes, an array:

```
const flowers = ['peony', 'daffodil', 'marigold'];

function addFlower(arr) {
  arr.push('lily');
}

addFlower(flowers);

console.log(flowers); // Output: ['peony', 'daffodil', 'marigold', 'lily']
```

Let's go over what happened in the example:

- The `flowers` array that has 3 elements.
- The function `addFlower()` has a parameter of `arr` uses `.push()` to add a `'lily'` element into `arr`.
- We call `addFlower()` with an argument of `flowers` which will execute the code inside `addFlower`.
- We check the value of `flowers` and it now includes the `'lily'` element! The array was mutated!

So when you pass an array into a function, if the array is mutated inside the function, that change will be maintained outside the function as well. You might also see this concept explained as *pass-by-reference* since what we're actually passing to the function is a reference to where the variable memory is stored and changing the memory.

Instruction

1. In **main.js**, there is an array `concept`. There is also a function `changeArr` that will assign the element in index `3` of an array to `'MUTATED'`. The function `changeArr` was called with an argument of `concept`. Underneath the function call, log `concept` to the console to check if this reassignment mutated the array.
2. Under the `console.log()` statement, define another function named `removeElement` that takes a parameter of `newArr`. Inside the function body call `.pop()` on `newArr`.
3. Call `removeElement()` with an argument of `concept`.

Solution:

```
const concept = ['arrays', 'can', 'be', 'mutated'];

function changeArr(arr){
  arr[3] = 'MUTATED';
}
changeArr(concept);
console.log(concept);

const removeElement = function(newArr){
  newArr.pop();
}
removeElement(concept);
console.log(concept)
```

1.10 Nested Arrays

Earlier we mentioned that arrays can store other arrays. When an array contains another array it is known as a *nested array*. Examine the example below:

```
const nestedArr = [[1], [2, 3]];
```

To access the nested arrays we can use bracket notation with the index value, just like we did to access any other element:

```
const nestedArr = [[1], [2, 3]];

console.log(nestedArr[1]); // Output: [2, 3]
```

Notice that `nestedArr[1]` will grab the element in index 1 which is the array `[2, 3]`. Then, if we wanted to access the elements within the nested array we can *chain*, or add on, more bracket notation with index values.

```
const nestedArr = [[1], [2, 3]];

console.log(nestedArr[1]); // Output: [2, 3]
console.log(nestedArr[1][0]); // Output: 2
```

In the second `console.log()` statement, we have two bracket notations chained to `nestedArr`. We know that `nestedArr[1]` is the array `[2, 3]`. Then to grab the first element from that array, we use `nestedArr[1][0]` and we get the value of `2`.

Solution:

```
const numberClusters = [[1, 2], [3, 4], [5, 6]];
// Declare variable and assign three array elements


const target = numberClusters[2][1];
// Declare variable and assign to access the element 6 inside numberClusters
```

Review Arrays

Nice work! In this lesson, we learned these concepts regarding arrays:

- Arrays are lists that store data in JavaScript.
- Arrays are created with brackets `[]`.
- Each item inside of an array is at a numbered position, or index, starting at `0`.
- We can access one item in an array using its index, with syntax like: `myArray[0]`.
- We can also change an item in an array using its index, with syntax like `myArray[0] = 'new string';`
- Arrays have a `length` property, which allows you to see how many items are in an array.
- Arrays have their own methods, including `.push()` and `.pop()`, which add and remove items from an array, respectively.
- Arrays have many methods that perform different tasks, such as `.slice()` and `.shift()`, you can find documentation at the Mozilla Developer Network website.
- Some built-in methods are mutating, meaning the method will change the array, while others are not mutating. You can always check the documentation.
- Variables that contain arrays can be declared with `let` or `const`. Even when declared with `const`, arrays are still mutable. However, a variable declared with `const` cannot be reassigned.
- Arrays mutated inside of a function will keep that change even outside the function.
- Arrays can be nested inside other arrays.
- To access elements in nested arrays chain indices using bracket notation.

Learning how to work with and manipulate arrays will help you work with chunks of data!

Practice Project: Secret Message

```
let secretMessage = ['Learning', 'is', 'not', 'about', 'what', 'you', 'get', '
easily', 'the', 'first', 'time,', 'it', 'is', 'about', 'what', 'you', 'can', '
figure', 'out.', '-2015,', 'Chris', 'Pine,', 'Learn', 'JavaScript'];

// remove the last element from secretMessage
secretMessage.pop();
// console.log(secretMessage.length);
// Add two elements to the end of the array
secretMessage.push("to", "Program");
// console.log(secretMessage);
// Replace the word "easily" to "right"
secretMessage[7] = "right";
// console.log(secretMessage);
// Remove the first item of the array
secretMessage.shift();
// console.log(secretMessage);
// Add the string programming to the start
secretMessage.unshift("Programming");
// console.log(secretMessage);
// Remove get,right,the,first & replace with know,
secretMessage.splice(6, 5, "know,");
// console.log(secretMessage)
// Print the array as a sentence
console.log(secretMessage.join(" "))
```

2.0 Loops

A *loop* is a programming tool that repeats a set of instructions until a specified condition, called a *stopping condition* is reached. As a programmer, you'll find that you rely on loops all the time! You'll hear the generic term *iterate* when referring to loops; iterate simply means "to repeat".

When we need to reuse a task in our code, we often bundle that action in a function. Similarly, when we see that a process has to repeat multiple times in a row, we write a loop. Loops allow us to create efficient code that automates processes to make scalable, manageable programs.

As illustrated in the diagram, loops iterate or repeat an action until a specific condition is met. When the condition is met, the loop stops and the computer moves on to the next part of the program.

2.1 Repeating Tasks Manually

Before we write our own loops let's take a moment to develop an appreciation for loops. The best way to do that is by showing you how cumbersome it would be if a repeated task required you to type out the same code every single time.

```
const vacationSpots = ["Mogadishu", "Tokyo", "Amsterdam"];
console.log(vacationSpots[0])
console.log(vacationSpots[1])
console.log(vacationSpots[2])
```

2.2 The For Loop

Instead of writing out the same code over and over, loops allow us to tell computers to repeat a given block of code on its own. One way to give computers these instructions is with a `for` loop.

The typical `for` loop includes an *iterator variable* that usually appears in all three expressions. The iterator variable is initialized, checked against the stopping condition, and assigned a new value on each loop iteration. Iterator variables can have any name, but it's best practice to use a descriptive variable name.

A `for` loop contains three expressions separated by `;` inside the parentheses:

1. an *initialization* starts the loop and can also be used to declare the iterator variable.
2. a *stopping condition* is the condition that the iterator variable is evaluated against— if the condition evaluates to `true` the code block will run, and if it evaluates to `false` the code will stop.
3. an *iteration statement* is used to update the iterator variable on each loop.

The `for` loop syntax looks like this:

```
for (let counter = 0; counter < 4; counter++) {
  console.log(counter);
}
```

In this example, the output would be the following:

```
0
1
2
3
```

Let's break down the example:

- The initialization is `let counter = 0`, so the loop will start counting at `0`.
- The stopping condition is `counter < 4`, meaning the loop will run as long as the iterator variable, `counter`, is less than 4.
- The iteration statement is `counter++`. This means after each loop, the value of `counter` will increase by 1. For the first iteration `counter` will equal `0`, for the second iteration `counter` will equal 1, and so on.
- The code block is inside of the curly braces, `console.log(counter)`, will execute until the condition evaluates to `false`. The condition will be false when `counter` is greater than or equal to 4 — the point that the condition becomes false is sometimes called the *stop condition*.

This `for` loop makes it possible to write `0`, `1`, `2`, and `3` programmatically.

Solution:

```
// Loop from 5 to 10 and log each number
for (let counter = 5; counter < 11; counter++) {
  console.log(counter);
}
```

2.3 Looping in Reverse

What if we want the `for` loop to log `3`, `2`, `1`, and then `0`? With simple modifications to the expressions, we can make our loop run backward!

To run a backward `for` loop, we must:

- Set the iterator variable to the highest desired value in the initialization expression.
- Set the stopping condition for when the iterator variable is less than the desired amount.
- The iterator should decrease in intervals after each iteration.

We'll practice by changing the `for` we wrote previously to now go in reverse. When writing/changing loops, there is a chance that our stopping condition isn't met and we get a dreaded *infinite loop* which essentially stops our programming from running anything else! To exit out of an infinite loop in an exercise, refresh the page - then fix the code for your loop.

```
// Loop backwards from 3 to 0
for (let counter = 3; counter >= 0; counter--){
  console.log(counter);
}
// Output: 3, 2, 1, 0
```

```
const nums = [1, 2, 3];

for (let i = nums.length - 1; i >= 0; i--) {
  console.log(nums[i]);
}

console.log('Time is up!');
// Output: 3, 2, 1, "Time is up!"
```

2.4 Looping through Arrays

`for` loops are very handy for iterating over data structures. For example, we can use a `for` loop to perform the same operation on each element on an array. Arrays hold lists of data, like customer names or product information. Imagine we owned a store and wanted to increase the price of every product in our catalog. That could be a lot of repeating code, but by using a `for` loop to iterate through the array we could accomplish this task easily.

To loop through each element in an array, a `for` loop should use the array's `.length` property in its condition.

Check out the example below to see how `for` loops iterate on arrays:

```
const animals = ['Grizzly Bear', 'Sloth', 'Sea Lion'];
for (let i = 0; i < animals.length; i++){
  console.log(animals[i]);
}
```

This example would give you the following output:

```
Grizzly Bear
Sloth
Sea Lion
```

In the loop above, we've named our iterator variable `i`. This is a variable naming convention you'll see in a lot of loops. When we use `i` to iterate through arrays we can think of it as being short-hand for the word **i**ndex. Notice how our stopping condition checks that `i` is less than `animals.length`. Remember that arrays are zero-indexed, the index of the last element of an array is equivalent to the length of that array minus 1. If we tried to access an element at the index of `animals.length` we will have gone too far!

With `for` loops, it's easier for us to work with elements in arrays.

Solution:

```
const vacationSpots = ['Bali', 'Paris', 'Tulum'];
// Log each element in the array after the string

for (let i = 0; i < vacationSpots.length; i++ ){
  console.log(`I would love to visit ${vacationSpots[i]}`);
}
// Output: I would love to visit Bali
// Output: I would love to visit Paris
// Output: I would love to visit Tulum
```

2.5 Nested Loops

When we have a loop running inside another loop, we call that a *nested loop*. One use for a nested `for` loop is to compare the elements in two arrays. For each round of the outer `for` loop, the inner `for` loop will run completely. Let's look at an example of a nested `for` loop:

```
const myArray = [6, 19, 20];
const yourArray = [19, 81, 2];
for (let i = 0; i < myArray.length; i++) {
  for (let j = 0; j < yourArray.length; j++) {
    if (myArray[i] === yourArray[j]) {
      console.log(`Both loops have the number: ${yourArray[j]}`)
    }
  }
};
```

Let's think about what's happening in the nested loop in our example. For each element in the outer loop array, `myArray`, the inner loop will run in its entirety comparing the current element from the outer array, `myArray[i]`, to each element in the inner array, `yourArray[j]`. When it finds a match, it prints a string to the console.

```
let str = "LOL";
for (let i = 0; i <= 4; i++){
  console.log("Outer:", i);
for (let j = 0; j < str.length; j++){
  console.log("  Inner:", str[j]);
}
}
```

Nested for of Loop:

```
const magicSquare = [
[2, 6, 7],
[3, 4, 8],
[5, 9, 1]
];

for (let row of magicSquare){
  let sum = 0
for (let num of row){
  sum += num;
}
console.log(`Row of ${row} sums to ${sum}`);
}
```

Now it's your turn to write a nested loop!

Note: To exit out of an infinite loop in an exercise, refresh the page - then fix the code for your loop(s).

Instruction & solution :

- Create a nested loop that iterates through `bobsFollowers` as the array for the outer loop and `tinasFollowers` as the array for the inner loop. If the current element from the outer loop is the same as the current element from the inner loop, push that element into the `mutualFollowers` array.

```
// Declare variable & set to an array of Bob's friends
let bobsFollowers = ["Jason", "Ben", "Jane", "Jill"];
// Declare variable & set it an array of Tina's friends
let tinasFollowers = ["Jill", "Jane", "Mary"];
// Declare a variable & set it to an empty array
let mutualFollowers = [];

for (let i = 0; i < bobsFollowers.length; i++){ // Array for the outer loop
  for (let j = 0; j < tinasFollowers.length; j++){ // Array for the inner loop
    if (bobsFollowers[i] === tinasFollowers[j]){
      mutualFollowers.push(bobsFollowers[i]); // Push the matching element
    }
  }
};
```

2.6 The While Loop

You're doing great! We're going to teach you about a different type of loop: the while loop. To start, let's convert a for loop into a while loop:

```
// A for loop that prints 1, 2, and 3
for (let counterOne = 1; counterOne < 4; counterOne++){
  console.log(counterOne);
}

// A while loop that prints 1, 2, and 3
let counterTwo = 1;
while (counterTwo < 4) {
  console.log(counterTwo);
  counterTwo++;
}
```

Let's break down what's happening with our while loop syntax:

- The counterTwo variable is declared before the loop. We can access it inside our while loop since it's in the global scope.
- We start our loop with the keyword while followed by our stopping condition, or *test condition*. This will be evaluated before each round of the loop. While the condition evaluates to true, the block will continue to run. Once it evaluates to false the loop will stop.
- Next, we have our loop's code block which prints counterTwo to the console and increments counterTwo.

What would happen if we didn't increment counterTwo inside our block? If we didn't include this, counterTwo would always have its initial value, 1. That would mean the testing condition counterTwo < 4 would always evaluate to true and our loop would never stop running! Remember, this is called an *infinite loop* and it's something we always want to **avoid**. Infinite loops can take up all of your computer's processing power potentially freezing your computer.

So you may be wondering when to use a while loop! The syntax of a for loop is ideal when we know how many times the loop should run, but we don't always know this in advance. Think of eating like a while loop: when you start taking bites, you don't know the exact number you'll need to become full. Rather you'll eat while you're hungry. In situations when we want a loop to execute an undetermined number of times, while loops are the best choice.

```
let num = 0;
while (num < 10){
  console.log(num)
  num++;
// while loops continue running as long as the test condition is true( 1-9
```

Example:

```
let i= 0
while (i < 11){
  console.log(i);
  i++
}
// Output: 1,2,3,4,5,6,7,8,9,10
```

Solution:

```
const cards = ['diamond', 'spade', 'heart', 'club'];

// Write your code below
let currentCard = []
while (currentCard !== "spade") {
  currentCard = cards[Math.floor(Math.random() * 4)];
console.log(currentCard);
}
```

2.7 Do...While Statements

In some cases, you want a piece of code to run at least once and then loop based on a specific condition after its initial run. This is where the `do...while` statement comes in.

A `do...while` statement says to do a task once and then keep doing it until a specified condition is no longer met. The syntax for a `do...while` statement looks like this:

```
let countString = '';
let i = 0;

do {
  countString = countString + i;
  i++;
} while (i < 5);

console.log(countString);
```

In this example, the code block makes changes to the `countString` variable by appending the string form of the `i` variable to it. First, the code block after the `do` keyword is executed once. Then the condition is evaluated. If the condition evaluates to `true`, the block will execute again. The looping stops when the condition evaluates to `false`.

Note that the `while` and `do...while` loop are different! Unlike the `while` loop, `do...while` will run at least once whether or not the condition evaluates to `true`.

```
const firstMessage = 'I will print!';
const secondMessage = 'I will not print!';

// A do while with a stopping condition that evaluates to false
do {
 console.log(firstMessage)
} while (true === false);

// A while loop with a stopping condition that evaluates to false
while (true === false){
  console.log(secondMessage)
};
```

Solution:

Within the block of our `do...while` loop, we'll increment the value of `cupsAdded`. Our stopping condition will compare `cupsAdded` to `cupsOfSugarNeeded`:

```
let cupsOfSugarNeeded = 4
let cupsAdded = 0


do {
  cupsAdded += 1;
   console.log(cupsAdded);
} while (cupsAdded < cupsOfSugarNeeded);
```

2.8 The break Keyword

Imagine we're looking to adopt a dog. We plan to go to the shelter every day for a year and then give up. But what if we meet our dream dog on day 65? We don't want to keep going to the shelter for the next 300 days just because our original plan was to go for a whole year. In our code, when we want to stop a loop from continuing to execute even though the original stopping condition we wrote for our loop hasn't been met, we can use the keyword `break`.

The break keyword allows programs to "break" out of the loop from within the loop's block.

Let's check out the syntax of a break keyword:

```
for (let i = 0; i < 99; i++) {
  if (i > 2 ) {
     break;
  }
  console.log('Banana.');
}

console.log('Orange you glad I broke out the loop!');
```

This is the output for the above code:

```
Banana.
Banana.
Banana.
Orange you glad I broke out the loop!
```

break statements can be especially helpful when we're looping through large data structures! With breaks, we can add test conditions besides the stopping condition, and exit the loop when they're met.

Another example of using break Keyword:

```
const strangeBirds = ['Shoebill', 'Cockatrice', 'Basan',
'Terrorbird','Parotia','Kakapo'];

for (const bird of strangeBirds) {
  if (bird === 'Basan'){
    break;
  }
  console.log(bird);
}
// output: Shoebill, Cockatrice
```

Solution:

```
const rapperArray = ["Lil' Kim", "Jay-Z", "Notorious B.I.G.", "Tupac"];
// Log each element from rapperArray in a for loop with the iterator variable
i.
for (let i = 0; i < rapperArray.length; i++){
  console.log(rapperArray[i]);
  if (rapperArray[i] === "Notorious B.I.G."){
    break;
  }
}
  console.log("And if you don't know, now you know.");
```

2.9 continue Keyword:

The continue statement is used to skip one iteration of the loop. For example: In the strangeBirds array it will iterate through the array and print out every value except "Cow".

```
const strangeBirds = ['Shoebill', 'Cockatrice', 'Basan', 'Cow', 'Terrorbird', 'Parotia', 'Kakapo'];

for (const bird of strangeBirds) {
  if  (bird === 'Cow'){
    continue;
  }
  console.log(bird);
}
```

```
const pokemonList = ['Pikachu', 'Charizard', 'Squirtle', 'Yoshi', 'Snorlax'];

// Write your code below
for (const pokemon of pokemonList){
  if (pokemon === "Yoshi"){
    continue;
  }
  console.log(`You caught a ${pokemon}!`)
}
```

2.10 The for...of Loop

While powerful, the for loop can be a bit cumbersome to set up, introduces room for errors, and could lead to difficult-to-read code.

As a remedy to some of the for loop's shortcomings, the ES6 version of JavaScript introduced the shorter and more concise for...of loop.

For of Loop:

```
const animals = ['Grizzly Bear', 'Sloth', 'Sea Lion'];
for (const animal of animals){
  console.log(animal);
}
```

for Loop:

```
const animals = ['Grizzly Bear', 'Sloth', 'Sea Lion'];
for (let i = 0; i < animals.length; i++){
  console.log(animals[i]);
}
```

They both give the same output:

```
Grizzly Bear
Sloth
Sea Lion
```

Another example of for of Loop:

```
const vacationSpots = ['Bali', 'Paris', 'Tulum'];
// Log each element in the array after the string

for (const vacation of vacationSpots){
  console.log(`I would love to visit ${vacation}`);
}
```

Given the spellingWord, use a for...of loop to log each letter using console.log().

```
const spellingWord = 'hippopotamus';

// Write your code below
for (const hippo of spellingWord){
  console.log(hippo)
}
```

```
const pokemonList = ['Pikachu', 'Charizard', 'Squirtle', 'Yoshi', 'Snorlax'];

// Write your code below
for (const pokemon of pokemonList){
  if (pokemon === "Yoshi"){
    continue;
  }
  console.log(`You caught a ${pokemon}!`)
}
```

Review

Great job! In this lesson, we learned how to write cleaner code with loops. You now know:

- Loops perform repetitive actions so we don't have to code that process manually every time.
- How to write `for` loops with an iterator variable that increments or decrements
- How to use a `for` loop to iterate through an array
- A nested `for` loop is a loop inside another loop
- `while` loops allow for different types of stopping conditions
- Stopping conditions are crucial for avoiding infinite loops.
- `do...while` loops run code at least once— only checking the stopping condition after the first execution
- The `break` keyword allows programs to leave a loop during the execution of its block

Practice Project: Whale Talk

```
let input = "turpentine and turtles";

const vowels = ["a", "e", "i", "o", "u"];

const resultArray = [];

for (let i = 0; i < input.length; i++ ){
      //  console.log(`i = ` +i);
    for (let j = 0; j < vowels.length; j++){
      //  console.log(`j = ` +j)
  if (input[i] === vowels[j]) {
    if (input[i] === "e"){
       resultArray.push("ee");
    }else if (input[i] === "u"){
       resultArray.push("uu");
    } else{
      resultArray.push(input[i]);
    }
    }
  }
}
// console.log(resultArray)
const resultString = resultArray.join("").toUpperCase()
console.log(resultString)
```

3.0 Objects

3.1 Introduction to Objects

It's time to learn more about the basic structure that permeates nearly every aspect of JavaScript programming: objects.

You're probably already more comfortable with objects than you think, because JavaScript loves objects! Many components of the language are actually objects under the hood, and even the parts that aren't— like strings or numbers— can still act like objects in some instances.

There are only seven fundamental data types in JavaScript, and six of those are the primitive data types: string, number, boolean, null, undefined, and symbol. With the seventh type, objects, we open our code to more complex possibilities. We can use JavaScript objects to model real-world things, like a basketball, or we can use objects to build the data structures that make the web possible.

At their core, JavaScript objects are containers storing related data and functionality, but that deceptively simple task is extremely powerful in practice. You've been using the power of objects all along, but now it's time to understand the mechanics of objects and start making your own

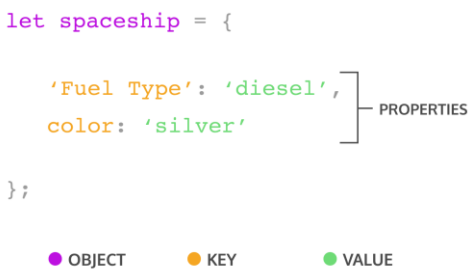
3.2 Creating Object Literals

Objects can be assigned to variables just like any JavaScript type. We use curly braces, `{}`, to designate an *object literal*:

```
let spaceship = {}; // spaceship is an empty object
```

We fill an object with unordered data. This data is organized into *key-value pairs*. A key is like a variable name that points to a location in memory that holds a value. A key's value can be of any data type in the language including functions or other objects.

We make a key-value pair by writing the key's name, or *identifier*, followed by a colon and then the value. We separate each key-value pair in an object literal with a comma (`,`). Keys are strings, but when we have a key that does not have any special characters in it, JavaScript allows us to omit the quotation marks:



```
// An object literal with two key-value pairs
let spaceship = {
  'Fuel Type': 'diesel',
  color: 'silver'
};
```

The spaceship object has two properties Fuel Type and color. 'Fuel Type' has quotation marks because it contains a space character.

Solution:

```
// Make a spaceship object named fasterShip with the color silver & Fuel Type equal to 'Turbo Fuel'
let fasterShip = {
  'Fuel Type': 'Turbo Fuel',
  color: 'silver'
};
```

3.2 Accessing Properties – dot notation

There are two ways we can access an object's property:

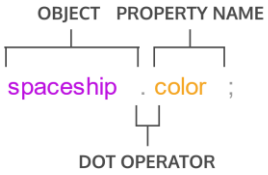
Let's explore the first way further— dot notation, .

You've used dot notation to access the properties and methods of built-in objects and data instances:

```
'hello'.length; // Returns 5
```

With property dot notation, we write the object's name, followed by the dot operator and then the property name (key):

```
let spaceship = {
  homePlanet: 'Earth',
  color: 'silver'
};
spaceship.homePlanet; // Returns 'Earth',
spaceship.color; // Returns 'silver',
```



If we try to access a property that does not exist on that object, undefined will be returned.

```
spaceship.favoriteIcecream; // Returns undefined
```

```
let spaceship = {
  homePlanet: 'Earth',
  color: 'silver',
  'Fuel Type': 'Turbo Fuel',
  numCrew: 5,
  flightPath: ['Venus', 'Mars', 'Saturn']
};
```

Solution:

```
// Create a variable & assign spaceship's numCrew property to it
let crewCount = spaceship.numCrew
 // Create a variable & assign spaceship's flightPath property to it
let planetArray = spaceship.flightPath
```

3.3 Bracket Notation

The second way to access a key's value is by using bracket notation, [ ].

You've used bracket notation when indexing an array:

```
['A', 'B', 'C'][0]; // Returns 'A'
```

To use bracket notation to access an object's property, we pass in the property name (key) as a string.

We **must** use bracket notation when accessing keys that have numbers, spaces, or special characters in them. Without bracket notation in these situations, our code would throw an error.

```
let spaceship = {
  'Fuel Type': 'Turbo Fuel',
  'Active Duty': true,
  homePlanet: 'Earth',
  numCrew: 5
};
spaceship['Active Duty'];   // Returns true
spaceship['Fuel Type'];   // Returns  'Turbo Fuel'
spaceship['numCrew'];   // Returns 5
spaceship['!!!!!!!!!!!!!!!'];   // Returns undefined
```

With bracket notation you can also use a variable inside the brackets to select the keys of an object. This can be especially helpful when working with functions:

```
let returnAnyProp = (objectName, propName) => objectName[propName];

returnAnyProp(spaceship, 'homePlanet'); // Returns 'Earth'
```

If we tried to write our returnAnyProp() function with dot notation (objectName.propName) the computer would look for a key of 'propName' on our object and not the value of the propName parameter.

Solution:

```
// Declare variable & assign spaceship's Active Mission
let isActive = spaceship["Active Mission"];
// console log the value of "Active Mission" property using Bracket Notation
console.log(spaceship["Active Mission"])
```

More solution:

To recap on how to access properties for both dot notation and bracket notation:

```
// To retrieve a value:
spaceship.color; // dot notation
spaceship["Fuel Type"]; // Bracket Notation
```

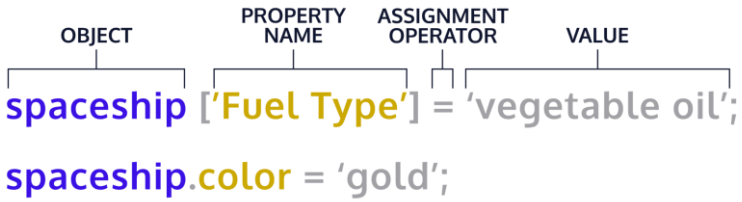
```
const refrigerator = {
  dairy: ['cheese', 'milk', 'sour cream'],
  temperature: 35,
  'produce drawer': {
    vegetables: ['lettuce', 'broccoli', 'peas'],
    fruit: ['apples', 'berries', 'grapes']
  }
}
refrigerator.temperature // output 35
refrigerator.dairy[1] // output milk
refrigerator['produce drawer'].fruit[0] // output apples
```

3.4 Property Assignment

Once we've defined an object, we're not stuck with all the properties we wrote. Objects are *mutable* meaning we can update them after we create them!

We can use either dot notation, `.`, or bracket notation, `[]`, and the assignment operator, `=` to add new key-value pairs to an object or change an existing property.

```
//updating values
spaceship.color = "red"; spaceship["Fuel Type"]; // dot notation
spaceship["color"] = "red" // bracket notation(preferred)
```



One of two things can happen with property assignment:

- If the property already exists on the object, whatever value it held before will be replaced with the newly assigned value.
- If there was no property with that name, a new property will be added to the object.

It's important to know that although we can't reassign an object declared with `const`, we can still mutate it, meaning we can add new properties and change the properties that are there.

```
const spaceship = {type: 'shuttle'};
spaceship = {type: 'alien'}; // TypeError: Assignment to constant variable.
spaceship.type = 'alien'; // Changes the value of the type property
spaceship.speed = 'Mach 5'; // Creates a new key of 'speed' with a value of
'Mach 5'
```

You can delete a property from an object with the `delete` operator.

```
const spaceship = {
  'Fuel Type': 'Turbo Fuel',
  homePlanet: 'Earth',
  mission: 'Explore the universe'
};

delete spaceship.mission;  // Removes the mission property
```

Solution:

```
/ Reassign the color property of the spaceship and set equal to glorious gold
spaceship["color"] = "glorious gold";
// Add numEngines property with num value between 1 and 10
spaceship["numEngines"] = Math.floor(Math.random() * 10 + 1);
//spaceship.numEngines = 7; (another valid answer)
// Use delete operator to remove Secret Mission property
delete spaceship["Secret Mission"];
```

More solution

```
//How can we add a property to the object?
let bikes = {
  schwinn: 'blue',
  trek: 'black'
}
bikes['specialized'] = 'red';
```

3.5 Methods

When the data stored on an object is a function we call that a *method*. A property is what an object has, while a method is what an object does.

Do object methods seem familiar? That's because you've been using them all along! For example `console` is a global javascript object and `.log()` is a method on that object. `Math` is also a global javascript object and `.floor()` is a method on it.

We can include methods in our object literals by creating ordinary, comma-separated key-value pairs. The key serves as our method's name, while the value is an anonymous function expression.

```
const alienShip = {
  invade: function () {
    console.log('Hello! We have come to dominate your planet. Instead of
Earth, it shall be called New Xaculon.')
  }
};
```

With the new method syntax introduced in ES6 we can omit the colon and the function keyword.

```
const alienShip = {
  invade () {
    console.log('Hello! We have come to dominate your planet. Instead of Earth, it shall be called New Xaculon.')
  }
};
```

Object methods are invoked by appending the object's name with the dot operator followed by the method name and parentheses:

```
alienShip.invade(); // Prints 'Hello! We have come to dominate your planet. Instead of Earth, it shall be called New Xaculon.'
```

Solution:

```
// Declare alienship object with methods: .retreat() & .takeOff()
const alienShip = {
  retreat () {
    console.log(retreatMessage)
  },
  takeOff() {
    console.log("Spim... Borp... Glix... Blastoff!");
  }
};
//Invoke or call your two methods
alienShip.retreat()
alienShip.takeOff()
```

3.6 Nested Objects

In application code, objects are often nested— an object might have another object as a property which in turn could have a property that's an array of even more objects!

In our spaceship object, we want a crew object. This will contain all the crew members who do important work on the craft. Each of those crew members are objects themselves. They have properties like name, and degree, and they each have unique methods based on their roles. We can also nest other objects in the spaceship such as a telescope or nest details about the spaceship's computers inside a parent nanoelectronics object.

```
let spaceship = {
  passengers: null,
  telescope: {
    yearBuilt: 2018,
    model: "91031-XLT",
    focalLength: 2032
  },
  crew: {
    captain: {
      name: 'Sandra',
      degree: 'Computer Engineering',
      encourageTeam() { console.log('We got this!') },
     'favorite foods': ['cookies', 'cakes', 'candy', 'spinach'] }
  },
  engine: {
    model: "Nimbus2000"
  },
  nanoelectronics: {
    computer: {
      terabytes: 100,
      monitors: "HD"
    },
    'back-up': {
      battery: "Lithium",
      terabytes: 50
    }
  }
};
```

We can chain operators to access nested properties. We'll have to pay attention to which operator makes sense to use in each layer. It can be helpful to pretend you are the computer and evaluate each expression from left to right so that each operation starts to feel a little more manageable.

```
spaceship.nanoelectronics['back-up'].battery; // Returns 'Lithium'
```

In the preceding code:

- First the computer evaluates `spaceship.nanoelectronics`, which results in an object containing the `back-up` and `computer` objects.
- We accessed the `back-up` object by appending `['back-up']`.
- The `back-up` object has a `battery` property, accessed with `.battery` which returned the value stored there: `'Lithium'`

Solution

```
// Create a variable capFave & assign the favorite food (1st element in the ar
ray) to it
let capFave = spaceship.crew.captain['favorite foods'][0];
// Assign spaceship's passenger as its valye an array of objects
spaceship.passengers = [{name: "Ben Silver"}]
// Create a variable & assign 1st element of spaceship.passengers array to it
let firstPassenger = spaceship.passengers[0]
```

3.7 Pass By Reference

Objects are *passed by reference*. This means when we pass a variable assigned to an object into a function as an argument, the computer interprets the parameter name as pointing to the space in memory holding that object. As a result, functions which change object properties actually mutate the object permanently (even when the object is assigned to a `const` variable).

```
const spaceship = {
  homePlanet : 'Earth',
  color : 'silver'
};

let paintIt = obj => {
  obj.color = 'glorious gold'
};

paintIt(spaceship);

spaceship.color // Returns 'glorious gold'
```

Our function `paintIt()` permanently changed the color of our `spaceship` object. However, reassignment of the `spaceship` variable wouldn't work in the same way:

```
let spaceship = {
  homePlanet : 'Earth',
  color : 'red'
};
let tryReassignment = obj => {
  obj = {
    identified : false,
    'transport type' : 'flying'
  }
  console.log(obj) // Prints {'identified': false, 'transport type': 'flying'}
```

```
};
tryReassignment(spaceship) // The attempt at reassignment does not work.
spaceship // Still returns {homePlanet : 'Earth', color : 'red'};

spaceship = {
  identified : false,
  'transport type': 'flying'
}; // Regular reassignment still works.
```

Let's look at what happened in the code example:

- We declared this spaceship object with let. This allowed us to reassign it to a new object with identified and 'transport type' properties with no problems.
- When we tried the same thing using a function designed to reassign the object passed into it, the reassignment didn't stick (even though calling console.log() on the object produced the expected result).
- When we passed spaceship into that function, obj became a reference to the memory location of the spaceship object, but *not* to the spaceship variable. This is because the obj parameter of the tryReassignment() function is a variable in its own right. The body of tryReassignment() has no knowledge of the spaceship variable at all!
- When we did the reassignment in the body of tryReassignment(), the obj variable came to refer to the memory location of the object {'identified' : false, 'transport type' : 'flying'}, while the spaceship variable was completely unchanged from its earlier value.

Solution:

```
let spaceship = {
  'Fuel Type' : 'Turbo Fuel',
  homePlanet : 'Earth'
};
// Write code below:

// Write a function that has object as a parameter & sets object's "Fuel Type"
 property to "avocado oil"
let greenEnergy = obj => {
  obj["Fuel Type"] = "avocado oil";
};
// Write a function that has object as a parameter & sets object's "disabled"
property to true
let remotelyDisable = function(obj){
  obj["disabled"] = true;
};
// Call the two function with spaceship object
```

```
greenEnergy(spaceship)
remotelyDisable(spaceship)
// log the spaceship object to verify the changes
console.log(spaceship)
```

3.8 Looping Through Objects

Loops are programming tools that repeat a block of code until a condition is met. We learned how to iterate through arrays using their numerical indexing, but the key-value pairs in objects aren't ordered! JavaScript has given us alternative solution for iterating through objects with the `for...in` syntax .

`for...in` will execute a given block of code for each property in an object.

```
let spaceship = {
  crew: {
    captain: {
      name: 'Lily',
      degree: 'Computer Engineering',
      cheerTeam() { console.log('You got this!') }
    },
    'chief officer': {
      name: 'Dan',
          degree: 'Aerospace Engineering',
          agree() { console.log('I agree, captain!') }
      },
      medic: {
          name: 'Clementine',
          degree: 'Physics',
          announce() { console.log(`Jets on!`) } },
      translator: {
          name: 'Shauna',
          degree: 'Conservation Science',
          powerFuel() { console.log('The tank is full!') }
      }
    }
};

// for...in
for (let crewMember in spaceship.crew) {
  console.log(`${crewMember}: ${spaceship.crew[crewMember].name}`);
}
```

Our `for...in` will iterate through each element of the `spaceship.crew` object. In each iteration, the variable `crewMember` is set to one of `spaceship.crew`'s keys, enabling us to log a list of crew members' role and `name`.

Here is the hint for the solution below of the for..in loop:

```
for (let variableName in outerObject.innerObject) {
  console.log(`${variableName}: ${outerObject.innerObject[variableName].proper
tyName}`)
};
```

Solution:

```

// Use for..in loop to iterate through spaceship.crew and log a list of roles
& names
 for (let crewMember in spaceship.crew) {
   console.log(`${crewMember}: ${spaceship.crew[crewMember].name}`);
   }
// Use for..in loop to iterate through spaceship.crew and log a list of names
& degrees
for (let crewMember in spaceship.crew) {
  console.log(`${spaceship.crew[crewMember].name}: ${spaceship.crew[crewMember
].degree}`);
   }
```

Another example of for...in loop which can be used to iterate over the keys of an object.

```
const car = {
  numDoors: 4,
  isDirty: true,
  color: 'red'
};
// This code will iterate through the car object and print just the keys
for (let key in car) {
  console.log(key)
}
// This code will iterate through the car object and print just the values
for (let key in car) {
  console.log(car[key])
}
// This code will iterate through the car object and print key-values pair
for (let key in car) {
   console.log(`${key}: ${car[key]}`);
}
```

Review

Way to go! You're well on your way to understanding the mechanics of objects in JavaScript. By building your own objects, you will have a better understanding of how JavaScript built-in objects work as well. You can also start imagining organizing your code into objects and modeling real world things in code.

Let's review what we learned in this lesson:

- Objects store collections of *key-value* pairs.
- Each key-value pair is a property—when a property is a function it is known as a method.
- An object literal is composed of comma-separated key-value pairs surrounded by curly braces.
- You can access, add or edit a property within an object by using dot notation or bracket notation.
- We can add methods to our object literals using key-value syntax with anonymous function expressions as values or by using the new ES6 method syntax.
- We can navigate complex, nested objects by chaining operators.
- Objects are mutable—we can change their properties even when they're declared with `const`.
- Objects are passed by reference— when we make changes to an object passed into a function, those changes are permanent.
- We can iterate through objects using the `For...in` syntax.

4.0 Advanced Objects

4.1 Advanced Objects Introduction

Remember, objects in JavaScript are containers that store data and functionality. In this lesson, we will build upon the fundamentals of creating objects and explore some advanced concepts.

So if there are no objections, let's learn more about objects!

In this lesson we will cover these topics:

- how to use the `this` keyword.
- conveying privacy in JavaScript methods.
- defining getters and setters in objects.
- creating factory functions.
- using destructuring techniques.

4.2 The this Keyword

Objects are collections of related data and functionality. We store that functionality in methods on our objects:

```
const goat = {
  dietType: 'herbivore',
  makeSound() {
    console.log('baaa');
  }
};
```

In our goat object we have a .makeSound() method. We can invoke the .makeSound() method on goat.

```
goat.makeSound(); // Prints baaa
```

Nice, we have a goat object that can print baaa to the console. Everything seems to be working fine. What if we wanted to add a new method to our goat object called .diet() that prints the goat's dietType?

```
const goat = {
  dietType: 'herbivore',
  makeSound() {
    console.log('baaa');
  },
  diet() {
    console.log(dietType);
  }
};
goat.diet();
// Output will be "ReferenceError: dietType is not defined"
```

That's strange, why is dietType not defined even though it's a property of goat? That's because inside the scope of the .diet() method, we don't automatically have access to other properties of the goat object.

Here's where the this keyword comes to the rescue. If we change the .diet() method to use the this, the .diet() works! :

```
const goat = {
  dietType: 'herbivore',
  makeSound() {
    console.log('baaa');
  },
  diet() {
    console.log(this.dietType);
  }
};

goat.diet();
// Output: herbivore
```

The `this` keyword references the *calling object* which provides access to the calling object's properties. In the example above, the calling object is `goat` and by using `this` we're accessing the `goat` object itself, and then the `dietType` property of `goat` by using property dot notation.

Let's get comfortable using the `this` keyword in a method.

Solution:

```
const robot = {
//Add property(model) & assign value "1E78V2",
  model: "1E78V2",
// Add property(energyLevel) & assign value 100
  energyLevel: 100,
// Add method & return string with model & energyLevel using this Keyword
  provideInfo() {
    return `I am ${this.model} and my current energy level is ${this.energyLev
el}.`
  }
};
// Log the result by calling .provideInfo() method on robot
console.log(robot.provideInfo());
```

4.3 Arrow Functions and this

We saw in the previous exercise that for a method, the calling object is the object the method belongs to. If we use the `this` keyword in a method then the value of `this` is the calling object. However, it becomes a bit more complicated when we start using arrow functions for methods. Take a look at the example below:

```
const goat = {
  dietType: 'herbivore',
  makeSound() {
    console.log('baaa');
  },
  diet: () => {
    console.log(this.dietType);
  }
};

goat.diet(); // Prints undefined
```

In the comment, you can see that goat.diet() would log `undefined`. So what happened? Notice that the .diet() method is defined using an arrow function.

Arrow functions inherently *bind*, or tie, an already defined `this` value to the function itself that is NOT the calling object. In the code snippet above, the value of `this` is the *global object*, or an object that exists in the global scope, which doesn't have a `dietType` property and therefore returns `undefined`.

To read more about either arrow functions or the global object check out the MDN documentation of the global object and arrow functions.

The key takeaway from the example above is to *avoid* using arrow functions when using `this` in a method!

Solution:

```
const robot = {
  energyLevel: 100,
  // Change from arrow function to short hand function
  checkEnergy() {
    console.log(`Energy is currently at ${this.energyLevel}%.`)
  }
}
robot.checkEnergy();
```

4.4 Privacy

Accessing and updating properties is fundamental in working with objects. However, there are cases in which we don't want other code simply accessing and updating an object's properties. When discussing *privacy* in objects, we define it as the idea that only certain properties should be mutable or able to change in value.

Certain languages have privacy built-in for objects, but JavaScript does not have this feature. Rather, JavaScript developers follow naming conventions that signal to other developers how to interact with a property. One common convention is to place an underscore `_` before the name of a property to mean that the property should not be altered. Here's an example of using `_` to prepend a property.

```
const bankAccount = {
  _amount: 1000
}
```

In the example above, the `_amount` is not intended to be directly manipulated.

Even so, it is still possible to reassign `_amount`:

```
bankAccount._amount = 1000000;
```

In later exercises, we'll cover the use of methods called *getters* and *setters*. Both methods are used to respect the intention of properties prepended, or began, with `_`. Getters can return the value of internal properties and setters can safely reassign property values. For now, let's see what happens if we can change properties that don't have setters or getters.

Solution:

```
const robot = {
  _energyLevel: 100,
  recharge(){
    this._energyLevel += 30;
    console.log(`Recharged! Energy is currently at ${this._energyLevel}%.`)
  }
};
// Reassign _energyLevel property to "high"
robot._energyLevel = "high";
// call .recharge() on robot
robot.recharge();
```

4.5 Getters

Getters are methods that get and return the internal properties of an object. But they can do more than just retrieve the value of a property! Let's take a look at a getter method:

```
const person = {
  _firstName: 'John',
  _lastName: 'Doe',
  get fullName() {
    if (this._firstName && this._lastName){
      return `${this._firstName} ${this._lastName}`;
    } else {
      return 'Missing a first name or a last name.';
    }
  }
}

// To call the getter method:
person.fullName; // 'John Doe'
```

Notice that in the getter method above:

- We use the `get` keyword followed by a function.

- We use an `if...else` conditional to check if both `_firstName` and `_lastName` exist (by making sure they both return truthy values) and then return a different value depending on the result.
- We can access the calling object's internal properties using `this`. In `fullName`, we're accessing both `this._firstName` and `this._lastName`.
- In the last line we call `fullName` on `person`. In general, getter methods do not need to be called with a set of parentheses. Syntactically, it looks like we're accessing a property.

Now that we've gone over syntax, let's discuss some notable advantages of using getter methods:

- Getters can perform an action on the data when getting a property.
- Getters can return different values using conditionals.
- In a getter, we can access the properties of the calling object using `this`.
- The functionality of our code is easier for other developers to understand.

Another thing to keep in mind when using getter (and setter) methods is that properties cannot share the same name as the getter/setter function. If we do so, then calling the method will result in an infinite call stack error. One workaround is to add an underscore before the property name like we did in the example above.

Solution:

```
const robot = {
  _model: '1E78V2',
  _energyLevel: 100,
  // Create a getter method using get Keyword
  get energyLevel(){
  // Add if statement to check if this.energyLevel is a number & return the st
ring
    if(typeof this._energyLevel === "number"){
      return `My current energy level is ${this._energyLevel}`
    }else{
      return `System malfunction: cannot retrieve energy level`
    }
  }
};
// Log the result of calling getter method energyLevel on robot
console.log(robot.energyLevel);
```

4.6 Setters

Along with getter methods, we can also create *setter* methods which reassign values of existing properties within an object. Let's see an example of a setter method:

```
const person = {
  _age: 37,
  set age(newAge){
    if (typeof newAge === 'number'){
      this._age = newAge;
    } else {
      console.log('You must assign a number to age');
    }
  }
};
```

Notice that in the example above:

- We can perform a check for what value is being assigned to `this._age`.
- When we use the setter method, only values that are numbers will reassign `this._age`
- There are different outputs depending on what values are used to reassign `this._age`.

Then to use the setter method:

```
person.age = 40;
console.log(person._age); // Logs: 40
person.age = '40'; // Logs: You must assign a number to age
```

Setter methods like `age` do not need to be called with a set of parentheses. Syntactically, it looks like we're reassigning the value of a property.

Like getter methods, there are similar advantages to using setter methods that include checking input, performing actions on properties, and displaying a clear intention for how the object is supposed to be used. Nonetheless, even with a setter method, it is still possible to directly reassign properties. For example, in the example above, we can still set `._age` directly:

```
person._age = 'forty-five'
console.log(person._age); // Prints forty-five
```

Solution:

```
const robot = {
  _model: '1E78V2',
  _energyLevel: 100,
  _numOfSensors: 15,
  get numOfSensors(){
    if(typeof this._numOfSensors === 'number'){
      return this._numOfSensors;
    } else {
      return 'Sensors are currently down.'
    }
  },
  // Add a setter method named numOfSensors using the set keyword to
  // verify that num is a equal to number & >= before setting as name property
  set numOfSensors(num){
    if(typeof num === "number" && num >= 0){
      this._numOfSensors = num;
    }else{
      console.log(`Pass in a number that is greater than or equal to 0`);
    }
  }
};
// Use the numOfSensors setter method on robot to assign _numOfSensors to 100
robot.numOfSensors = 100;
console.log(robot.numOfSensors);
```

4.7 Factory Functions

So far we've been creating objects individually, but there are times where we want to create many instances of an object quickly. Here's where *factory functions* come in. A real world factory manufactures multiple copies of an item quickly and on a massive scale. A factory function is a function that returns an object and can be reused to make multiple object instances. Factory functions can also have parameters allowing us to customize the object that gets returned.

Let's say we wanted to create an object to represent monsters in JavaScript. There are many different types of monsters and we could go about making each monster individually but we can also use a factory function to make our lives easier. To achieve this diabolical plan of creating multiple monsters objects, we can use a factory function that has parameters:

```
const monsterFactory = (name, age, energySource, catchPhrase) => {
  return {
    name: name,
    age: age,
    energySource: energySource,
    scare() {
      console.log(catchPhrase);
    }
  }
};
```

In the monsterFactory function above, it has four parameters and returns an object that has the properties: name, age, energySource, and scare(). To make an object that represents a specific monster like a ghost, we can call monsterFactory with the necessary arguments and assign the return value to a variable:

```
const ghost = monsterFactory('Ghouly', 251, 'ectoplasm', 'BOO!');
ghost.scare(); // 'BOO!'
```

Now we have a ghost object as a result of calling monsterFactory() with the needed arguments. With monsterFactory in place, we don't have to create an object literal every time we need a new monster. Instead, we can invoke the monsterFactory function with the necessary arguments to make a monster for us!

Solution:

```
// Create a factory function that has two parameters
const robotFactory = (model, mobile) => {
  return {
    model: model,
    mobile: mobile,
    beep(){
f     console.log("Beep Boop");
  }
  }
}
// Declare variable named tinCan & assign the value of calling robotFactory wi
th two arguments
const tinCan = robotFactory("P-500", true)
tinCan.beep();
```

4.8 Property Value Shorthand

ES6 introduced some new shortcuts for assigning properties to variables known as *destructuring*.

In the previous exercise, we created a factory function that helped us create objects. We had to assign each property a key and value even though the key name was the same as the parameter name we assigned to it. To remind ourselves, here's a truncated version of the factory function:

```
const monsterFactory = (name, age) => {
  return {
    name: name,
    age: age
  }
};
```

Imagine if we had to include more properties, that process would quickly become tedious! But we can use a destructuring technique, called *property value shorthand*, to save ourselves some keystrokes. The example below works exactly like the example above:

```
const monsterFactory = (name, age) => {
 return {
  name,
  age
 }
};
```

Notice that we don't have to repeat ourselves for property assignments!

Solution:

```
function robotFactory(model, mobile){
  return {
    model,
    mobile,
    beep() {
      console.log('Beep Boop');
    }
  }
}

// To check that the property value shorthand technique worked:
const newRobot = robotFactory('P-501', false)
console.log(newRobot.model)
console.log(newRobot.mobile)
```

4.9 Destructured Assignment

We often want to extract key-value pairs from objects and save them as variables. Take for example the following object:

```
const vampire = {
  name: 'Dracula',
  residence: 'Transylvania',
  preferences: {
    day: 'stay inside',
    night: 'satisfy appetite'
  }
};
```

If we wanted to extract the `residence` property as a variable, we could use the following code:

```
const residence = vampire.residence;
console.log(residence); // Prints 'Transylvania'
```

However, we can also take advantage of a destructuring technique called *destructured assignment* to save ourselves some keystrokes. In destructured assignment we create a variable with the name of an object's key that is wrapped in curly braces `{ }` and assign to it the object. Take a look at the example below:

```
const { residence } = vampire;
console.log(residence); // Prints 'Transylvania'
```

Look back at the `vampire` object's properties in the first code example. Then, in the example above, we declare a new variable `residence` that extracts the value of the `residence` property of `vampire`. When we log the value of `residence` to the console, `'Transylvania'` is printed.

We can even use destructured assignment to grab nested properties of an object:

```
const { day } = vampire.preferences;
console.log(day); // Prints 'stay inside'
```

Solution:

```
const robot = {
  model: '1E78V2',
  energyLevel: 100,
  functionality: {
    beep() {
      console.log('Beep Boop');
    },
    fireLaser() {
      console.log('Pew Pew');
    },
  }
};
```

```
//If we wanted to extract the functionality property as a variable, we could u
se the following code:
//const functionality = robot.functionality;
//functionality.beep();

// Use destructured assignment to the same as above but with less code
const {functionality} = robot;
// Call .beep() method on functionality
functionality.beep() // output Beep Boop
```

4.10 Built-in Object Methods

In the previous exercises we've been creating instances of objects that have their own methods. But, we can also take advantage of built-in methods for Objects!

For example, we have access to object instance methods like: .hasOwnProperty(), .valueOf(), and many more! Practice your documentation reading skills and check out: MDN's object instance documentation. There are also useful Object class methods such as Object.assign(), Object.entries(), and Object.keys() just to name a few. For a comprehensive list, browse: MDN's object instance documentation.

Solution:

```
const robot = {
  model: 'SAL-1000',
  mobile: true,
  sentient: false,
  armor: 'Steel-plated',
  energyLevel: 75
};
// Add the object missing robot in the parentheses
const robotKeys = Object.keys(robot);
console.log(robotKeys);

// Declare robotEntries below this line:
const robotEntries = Object.entries(robot);
console.log(robotEntries);

// Declare newRobot below this line:
const newRobot = Object.assign({laserBlaster: true, voiceRecognition: true});
console.log(newRobot);
```

More solution

```
let bikes = {
  schwinn: 'blue',
  trek: 'black',
  specialized: 'red'
}
 console.log(bikes)
 console.log(Object.entries(bikes));
 console.log(Object.keys(bikes));
 console.log(Object.values(bikes));
 console.log(Object.assign(bikes));
```

Review

Congratulations on finishing Advanced Objects!

Let's review the concepts covered in this lesson:

- The object that a method belongs to is called the *calling object*.
- The `this` keyword refers to the calling object and can be used to access properties of the calling object.
- Methods do not automatically have access to other internal properties of the calling object.
- The value of `this` depends on where the `this` is being accessed from.
- We cannot use arrow functions as methods if we want to access other internal properties.
- JavaScript objects do not have built-in privacy, rather there are conventions to follow to notify other developers about the intent of the code.
- The usage of an underscore before a property name means that the original developer did not intend for that property to be directly changed.
- Setters and getter methods allow for more detailed ways of accessing and assigning properties.
- Factory functions allow us to create object instances quickly and repeatedly.
- There are different ways to use object destructuring: one way is the property value shorthand and another is destructured assignment.
- As with any concept, it is a good skill to learn how to use the documentation with objects!

Practice Project - Meal Maker

```
const menu = {
  _courses: {
    appetizers:[],
    mains:[],
    desserts:[],
},
  get appetizers(){
  return this._courses.appetizers;
},
  get mains(){
  return this._courses.mains;
},
  get desserts(){
  return this._courses.desserts;
},
set appetizers(appetizers){
  this._courses.appetizers = appetizers;
},
set appetizers(mains){
  this._courses.mains = mains;
},
set appetizers(desserts){
  this._courses.desserts = desserts;
},
  get courses(){
    return{
    appetizers: this.appetizers,
    mains: this.mains,
    desserts: this.desserts,
  };
  },
addDishToCourse(courseName, dishName, dishPrice){
  const dish = {
  name: dishName,
  price: dishPrice,
};
return this._courses[courseName].push(dish);
```

```
},
getRandomDishFromCourse(courseName){
  const dishes = this._courses[courseName];
  const randomIndex = Math.floor(Math.random() * dishes.length);
  return dishes[randomIndex];
},
generateRandomMeal(){
  const appetizer = this.getRandomDishFromCourse("appetizers")
  const main = this.getRandomDishFromCourse("mains")
  const dessert = this.getRandomDishFromCourse("desserts")
  const totalPrice = appetizer.price + main.price + dessert.price;
  return ` Your meal is ${appetizer.name}, ${main.name}, ${dessert.name} and t
he total price is: £${totalPrice}`;
},
};
menu.addDishToCourse("appetizers", "chicken strips", 10);
menu.addDishToCourse("appetizers", "nachos", 5);
menu.addDishToCourse("appetizers", "salad", 6);

menu.addDishToCourse("mains", "bariis iskukaris", 20);
menu.addDishToCourse("mains", "chicken korma", 15);
menu.addDishToCourse("mains", "pasta", 14);

menu.addDishToCourse("desserts", "chocolate", 20);
menu.addDishToCourse("desserts", "icecream", 15);
menu.addDishToCourse("desserts", "cake", 14);

const meal = menu.generateRandomMeal();
console.log(meal);
```

Practice Project - Team Stats

```
const team = {
  _players: [
 {
    firstName: "Eddie",
    lastName: "Driver",
    age: 18
 },
 {
    firstName: "Derrick",
    lastName: "Jenkins",
    age: 21
 },
 {
    firstName: "Simon",
    lastName: "Fern",
    age: 19
 },
  ],
  _games: [
 {
  opponent: 'Falcons',
  teamPoints: 3,
  opponentPoints: 27
},
{
  opponent: 'Saints',
  teamPoints: 32,
  opponentPoints: 23
},
{
  opponent: 'Giants',
  teamPoints: 22,
  opponentPoints: 29
},
  ],
```

```
get players(){
  return this._players;
},
get games(){
  return this._games;
},
addPlayer(firstName, lastName, age){
  const player = {
    firstName: firstName,
    lastName: lastName,
    age: age
  };
  this.players.push(player);
},
addGame(opp, myPoints, oppPoints){
  const game = {
    opponent: opp,
    teamPoints: myPoints,
    opponentPoints: oppPoints
  };
  this.games.push(game);
}
}
team.addPlayer("Steph", "Curry", 28);
team.addPlayer("Lisa", "Leslie", 44);
team.addPlayer("Bugs", "Bunny", 76);

team.addGame("Seahawks", 34, 27);
team.addGame("Bengals", 29, 33);
team.addGame("Rams", 21, 41);

//console.log(team.players);

console.log(team.games);
```

5.0 Higher-order functions

5.1 Introduction

We are often unaware of the number of assumptions we make when we communicate with other people in our native languages. If we told you to "count to three," we would expect you to say or think the numbers one, two and three. We assumed you would know to start with "one" and end with "three". With programming, we're faced with needing to be more explicit with our directions to the computer. Here's how we might tell the computer to "count to three":

```
for (let i = 1; i<=3; i++) {
  console.log(i)
}
```

When we speak to other humans, we share a vocabulary that gives us quick ways to communicate complicated concepts. When we say "bake", it calls to mind a familiar subroutine— preheating an oven, putting something into an oven for a set amount of time, and finally removing it. This allows us to *abstract* away a lot of the details and communicate key concepts more concisely. Instead of listing all those details, we can say, "We baked a cake," and still impart all that meaning to you.

In programming, we can accomplish "abstraction" by writing functions. In addition to allowing us to reuse our code, functions help to make clear, readable programs. If you encountered `countToThree()` in a program, you might be able to quickly guess what the function did without having to stop and read the function's body.

We're also going to learn about another way to add a level of abstraction to our programming: *higher-order functions*. *Higher-order functions* are functions that accept other functions as arguments and/or return functions as output. This enables us to build abstractions on other abstractions, just like "We hosted a birthday party" is an abstraction that may build on the abstraction "We made a cake."

In summary, using more abstraction in our code allows us to write more modular code which is easier to read and debug.

5.2 Functions as Data

JavaScript functions behave like any other data type in the language; we can assign functions to variables, and we can reassign them to new variables. Below, we have an annoyingly long function name that hurts the readability of any code in which it's used. Let's pretend this function does important work and needs to be called repeatedly!

```
const announceThatIAmDoingImportantWork = () => {
    console.log("I'm doing very important work!");
};
```

What if we wanted to rename this function without sacrificing the source code? We can re-assign the function to a variable with a suitably short name:

```
const busy = announceThatIAmDoingImportantWork;

busy(); // This function call barely takes any space!
```

`busy` is a variable that holds a *reference* to our original function. If we could look up the address in memory of `busy` and the address in memory of `announceThatIAmDoingImportantWork` they would point to the same place. Our new `busy()` function can be invoked with parentheses as if that was the name we originally gave our function.

Notice how we assign `announceThatIAmDoingImportantWork` without parentheses as the value to the `busy` variable. We want to assign the value of the function itself, not the value it returns when invoked.

In JavaScript, functions are *first class objects*. This means that, like other objects you've encountered, JavaScript functions can have properties and methods.

Since functions are a type of object, they have properties such as `.length` and `.name` and methods such as `.toString()`. You can see more about the methods and properties of functions [in the documentation].

Functions are special because we can invoke them, but we can still treat them like any other type of data. Let's get some practice doing that!

Solution:

```
// Create a shorter variable is2p2
const is2p2 = checkThatTwoPlusTwoEqualsFourAMillionTimes;
// Invoke your is2p2() function
is2p2()
// Find the name of the original function
console.log(is2p2.name);
```

5.3 Functions as Parameters

Since functions can behave like any other type of data in JavaScript, it might not surprise you to learn that we can also pass functions (into other functions) as parameters. A *higher-order function* is a function that either accepts functions as parameters, returns a function, or both! We call the functions that get passed in as parameters and invoked *callback functions* because they get called during the execution of the higher-order function.

When we pass a function in as an argument to another function, we don't invoke it. Invoking the function would evaluate to the return value of that function call. With callbacks, we pass in the function itself by typing the function name *without* the parentheses (that would evaluate to the result of calling the function):

```
const timeFuncRuntime = funcParameter => {
   let t1 = Date.now();
   funcParameter();
   let t2 = Date.now();
   return t2 - t1;
}

const addOneToOne = () => 1 + 1;

timeFuncRuntime(addOneToOne);
```

We wrote a higher-order function, `timeFuncRuntime()`. It takes in a function as an argument, saves a starting time, invokes the callback function, records the time after the function was called, and returns the time the function took to run by subtracting the starting time from the ending time.

This higher-order function could be used with any callback function which makes it a potentially powerful piece of code.

We then invoked `timeFuncRuntime()` first with the `addOneToOne()` function - note how we passed in `addOneToOne` and did not invoke it.

```
timeFuncRuntime(() => {
  for (let i = 10; i>0; i--){
    console.log(i);
  }
});
```

In this example, we invoked `timeFuncRuntime()` with an anonymous function that counts backwards from 10. Anonymous functions can be arguments too!

Let's get some practice using functions and writing higher-order functions.

Solution:

```
//Save a variable, time2p2 & assign the result of calling the function with th
e other function
```

```
const time2p2 = timeFuncRuntime(checkThatTwoPlusTwoEqualsFourAMillionTimes);
// Write higher-order function
const checkConsistentOutput = function(fn, val){
  const firstAttempt = fn(val);
  const secondAttempt = fn(val);
  if(firstAttempt === secondAttempt){
    return firstAttempt;
  }else{
    return `This function returned inconsistent results`
  }
};
// Invoke the function with addTwo function & number as arguments
checkConsistentOutput(addTwo, 4)
```

Review

Great job! By thinking about functions as data and learning about higher-order functions, you've taken important steps in being able to write clean, modular code and take advantage of JavaScript's flexibility.

Let's review what we learned in this lesson:

- Abstraction allows us to write complicated code in a way that's easy to reuse, debug, and understand for human readers
- We can work with functions the same way we would any other type of data including reassigning them to new variables
- JavaScript functions are first-class objects, so they have properties and methods like any object
- Functions can be passed into other functions as parameters
- A higher-order function is a function that either accepts functions as parameters, returns a function, or both

6.0 Iterators

6.1 Introduction to Iterators

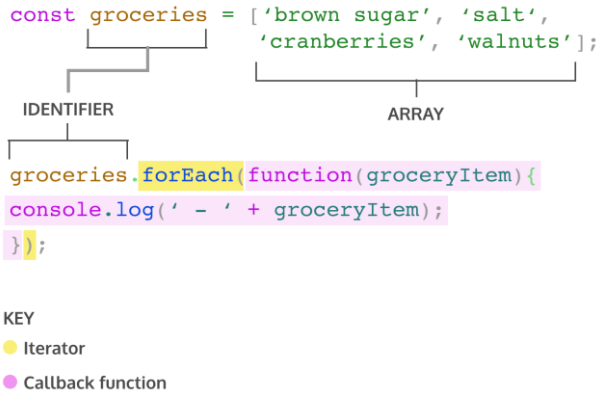
Imagine you had a grocery list and you wanted to know what each item on the list was. You'd have to scan through each row and check for the item. This common task is similar to what we have to do when we want to iterate over, or loop through, an array. One tool at our disposal is the `for` loop. However, we also have access to built-in array methods which make looping easier.

The built-in JavaScript array methods that help us iterate are called *iteration methods*, at times referred to as *iterators*. Iterators are methods called on arrays to manipulate elements and return values. Notice below the different methods being called on the arrays:

```
forEach()
.map()
.filter()

const artists = ['Picasso', 'Kahlo', 'Matisse', 'Utamaro'];

artists.forEach(artist => {
  console.log(artist + ' is one of my favorite artists.');
});

const numbers = [1, 2, 3, 4, 5];

const squareNumbers = numbers.map(number => {
  return number * number;
});
console.log(squareNumbers);

const things = ['desk', 'chair', 5, 'backpack', 3.14, 100];

const onlyNumbers = things.filter(thing => {
  return typeof thing === 'number';
});
console.log(onlyNumbers);
```

6.2 The .forEach() Method

The first iteration method that we're going to learn is .forEach(). Aptly named, .forEach() will execute the same code for each element of an array.



The code on the right will log a nicely formatted list of the groceries to the console. Let's explore the syntax of invoking .forEach().

- groceries.forEach() calls the forEach method on the groceries array.
- .forEach() takes an argument of callback function. Remember, a callback function is a function passed as an argument into another function.
- .forEach() loops through the array and executes the callback function for each element. During each execution, the current element is passed as an argument to the callback function.
- The return value for .forEach() will always be undefined.

Another way to pass a callback for .forEach() is to use arrow function syntax.

```
groceries.forEach(groceryItem => console.log(groceryItem));
```

We can also define a function beforehand to be used as the callback function.

```
function printGrocery(element){
  console.log(element);
}

groceries.forEach(printGrocery);
```

The above example uses a function declaration but you can also use a function expression or arrow function as well.

All three code snippets do the same thing. In each array iteration method, we can use any of the three examples to supply a callback function as an argument to the iterator. It's good to be aware of the different ways to pass in callback functions as arguments in iterators because developers have different stylistic preferences. Nonetheless, due to the strong adoption of ES6, we will be using arrow function syntax in the later exercises.

Solution:

```
const fruits = ['mango', 'papaya', 'pineapple', 'apple'];

// Iterate over fruits below
```

```
fruits.forEach(fruit => {
  console.log(`I want to eat a ${fruit}`)
});
```

6.3 The .map() Method

The second iterator we're going to cover is .map(). When .map() is called on an array, it takes an argument of a callback function and returns a new array! Take a look at an example of calling .map():

```
const numbers = [1, 2, 3, 4, 5];

const bigNumbers = numbers.map(number => {
  return number * 10;
});
```

.map() works in a similar manner to .forEach()— the major difference is that .map() returns a new array.

In the example above:

- numbers is an array of numbers.
- bigNumbers will store the return value of calling .map() on numbers.
- numbers.map will iterate through each element in the numbers array and pass the element into the callback function.
- return number * 10 is the code we wish to execute upon each element in the array. This will save each value from the numbers array, multiplied by 10, to a new array.

If we take a look at numbers and bigNumbers:

```
console.log(numbers); // Output: [1, 2, 3, 4, 5]
console.log(bigNumbers); // Output: [10, 20, 30, 40, 50]
```

Notice that the elements in numbers were not altered and bigNumbers is a new array.

Solution:

```
const animals = ['Hen', 'elephant', 'llama', 'leopard', 'ostrich', 'Whale', 'octopus', 'rabbit', 'lion', 'dog'];

// Create the secretMessage array below
const secretMessage = animals.map(firstCharacter => {
  return `${firstCharacter[0]}`
});
// cleaner solution in one line:
//const secretMessage = animals.map(animal => animal[0])
console.log(secretMessage.join(''));

const bigNumbers = [100, 200, 300, 400, 500];

// Create the smallNumbers array below

const smallNumbers = bigNumbers.map(num => {
  return num / 100;
});
// const smallNumbers = bigNumbers.map(num => num / 100);
console.log(smallNumbers)
```

6.4 The .filter() Method

Another useful iterator method is `.filter()`. Like `.map()`, `.filter()` returns a new array. However, `.filter()` returns an array of elements after filtering out certain elements from the original array. The callback function for the `.filter()` method should return `true` or `false` depending on the element that is passed to it. The elements that cause the callback function to return `true` are added to the new array. Take a look at the following example:

```
const words = ['chair', 'music', 'pillow', 'brick', 'pen', 'door'];

const shortWords = words.filter(word => {
  return word.length < 6;
});
```

- `words` is an array that contains string elements.
- `const shortWords =` declares a new variable that will store the returned array from invoking `.filter()`.

- The callback function is an arrow function that has a single parameter, `word`. Each element in the `words` array will be passed to this function as an argument.
- `word.length < 6;` is the condition in the callback function. Any `word` from the `words` array that has fewer than `6` characters will be added to the `shortWords` array.

Let's also check the values of `words` and `shortWords`:

```
console.log(words); // Output: ['chair', 'music', 'pillow', 'brick', 'pen',
'door'];
console.log(shortWords); // Output: ['chair', 'music', 'brick', 'pen', 'door']
```

Observe how `words` was not mutated, i.e. changed, and `shortWords` is a new array.

Solution:

```
const randomNumbers = [375, 200, 3.14, 7, 13, 852];

// Call .filter() on randomNumbers below
const smallNumbers = randomNumbers.filter(small => {
  return small < 250;
});
const favoriteWords = ['nostalgia', 'hyperbole', 'fervent', 'esoteric', 'seren
e'];


// Call .filter() on favoriteWords below
const longFavoriteWords = favoriteWords.filter(longWords => longWords.length >
 7);

console.log(longFavoriteWords)
```

6.5 The .findIndex() Method

We sometimes want to find the location of an element in an array. That's where the .findIndex() method comes in! Calling .findIndex() on an array will return the index of the first element that evaluates to true in the callback function.

```
const jumbledNums = [123, 25, 78, 5, 9];

const lessThanTen = jumbledNums.findIndex(num => {
  return num < 10;
});
```

- jumbledNums is an array that contains elements that are numbers.
- const lessThanTen = declares a new variable that stores the returned index number from invoking .findIndex().
- The callback function is an arrow function that has a single parameter, num. Each element in the jumbledNums array will be passed to this function as an argument.
- num < 10; is the condition that elements are checked against. .findIndex() will return the index of the first element which evaluates to true for that condition.

Let's take a look at what lessThanTen evaluates to:

```
console.log(lessThanTen); // Output: 3
```

If we check what element has index of 3:

```
console.log(jumbledNums[3]); // Output: 5
```

Great, the element in index 3 is the number 5. This makes sense since 5 is the first element that is less than 10.

If there isn't a single element in the array that satisfies the condition in the callback, then .findIndex() will return -1.

```
const greaterThan1000 = jumbledNums.findIndex(num => {
  return num > 1000;
});
console.log(greaterThan1000); // Output: -1
```

Solution:

```
const animals = ['hippo', 'tiger', 'lion', 'seal', 'cheetah', 'monkey', 'salam
ander', 'elephant'];


// Call .findIndex() on animals below
const foundAnimal = animals.findIndex(animal => animal === "elephant");
```

```
console.log(foundAnimal);

// Call .findIndex() on animals below
const startsWithS = animals.findIndex(animalS => animalS[0] === "s");

console.log(startsWithS);
```

6.6 The .reduce() Method

Another widely used iteration method is .reduce(). The .reduce() method returns a single value after iterating through the elements of an array, thereby *reducing* the array. Take a look at the example below:

```
const numbers = [1, 2, 4, 10];

const summedNums = numbers.reduce((accumulator, currentValue) => {
  return accumulator + currentValue
})

console.log(summedNums) // Output: 17
```

Here are the values of accumulator and currentValue as we iterate through the numbers array:

Iteration	accumulator	currentValue	return value
First	1	2	3
Second	3	4	7
Third	7	10	17

Now let's go over the use of .reduce() from the example above:

- numbers is an array that contains numbers.
- summedNums is a variable that stores the returned value of invoking .reduce() on numbers.
- numbers.reduce() calls the .reduce() method on the numbers array and takes in a callback function as argument.
- The callback function has two parameters, accumulator and currentValue. The value of accumulator starts off as the value of the first element in the array and the currentValue starts as the second element. To see the value of accumulator and currentValue change, review the chart above.
- As .reduce() iterates through the array, the return value of the callback function becomes the accumulator value for the next iteration, currentValue takes on the value of the current element in the looping process.

The .reduce() method can also take an optional second parameter to set an initial value for accumulator (remember, the first argument is the callback function!). For instance:

```
const numbers = [1, 2, 4, 10];

const summedNums = numbers.reduce((accumulator, currentValue) => {
  return accumulator + currentValue
}, 100)  // <- Second argument for .reduce()

console.log(summedNums); // Output: 117
```

Here's an updated chart that accounts for the second argument of 100:

Iteration #	accumulator	currentValue	return value
First	100	1	101
Second	101	2	103
Third	103	4	107
Fourth	107	10	117

Solution:

```
const newNumbers = [1, 3, 5, 7];

// Call .reduce() on newNumbers below
const newSum = newNumbers.reduce((accumulator, currentValue) => {
  console.log('The value of accumulator: ', accumulator);
  console.log('The value of currentValue: ', currentValue);
  return accumulator + currentValue;
}, 10); // added a second argument of 10

console.log(newSum); // output:26
```

6.7 Iterator Documentation

There are many additional built-in array methods, a complete list of which is on the MDN's Array iteration methods page. Below are some examples such as every(), some() and filter()

Solution:

```
const words = ['unique', 'uncanny', 'pique', 'oxymoron', 'guise'];

// Something is missing in the method call below - some()
console.log(words.some((word) => {
  return word.length < 6;
}));

// Use filter to create a new array
const interestingWords = words.filter(word => word.length > 5);

console.log(interestingWords)

// Make sure to uncomment the code below and fix the incorrect code before run
ning it
// every() checks if every element meets specified test
console.log(interestingWords.every(word => {word.length > 5}));
```

6.8 Choose the Right Iterator

There are many iteration methods you can choose. In addition to learning the correct syntax for the use of iteration methods, it is also important to learn how to choose the correct method for different scenarios.

```
const cities = ['Orlando', 'Dubai', 'Edinburgh', 'Chennai', 'Accra', 'Denver',
 'Eskisehir', 'Medellin', 'Yokohama'];

const nums = [1, 50, 75, 200, 350, 525, 1000];

//  Choose a method that will return undefined
cities.forEach(city => console.log('Have you visited ' + city + '?'));
```

```
// Choose a method that will return a new array
const longCities = cities.filter(city => city.length > 7);

// Choose a method that will return a single value
const word = cities.reduce((acc, currVal) => {
  return acc + currVal[0]
}, "C");

console.log(word)

// Choose a method that will return a new array
const smallerNums = nums.map(num => num - 5);

// Choose a method that will return a boolean value
nums.every(num => num < 0);
```

Review

Awesome job on clearing the iterators lesson! You have learned a number of useful methods in this lesson as well as how to use the JavaScript documentation from the Mozilla Developer Network to discover and understand additional methods. Let's review!

- `.forEach()` is used to execute the same code on every element in an array but does not change the array and returns `undefined`.
- `.map()` executes the same code on every element in an array and returns a new array with the updated elements.
- `.filter()` checks every element in an array to see if it meets certain criteria and returns a new array with the elements that return truthy for the criteria.
- `.findIndex()` returns the index of the first element of an array that satisfies a condition in the callback function. It returns `-1` if none of the elements in the array satisfies the condition.
- `.reduce()` iterates through an array and takes the values of the elements and returns a single value.
- All iterator methods take a callback function, which can be a pre-defined function, a function expression, or an arrow function.
- You can visit the Mozilla Developer Network to learn more about iterator methods (and all other parts of JavaScript!).

Practice Project: Mini linter

```
// Split the string into individual words
const storyWords = story.split(" ")

// Check how many words in the array
storyWords.length;
// Iterate over the array & filter() method to not include unnecessary words
const betterWords = storyWords.filter(word => !unnecessaryWords.includes(word)
);
// console.log(betterWords);

// Check how many times the words are used in the overusedWords array

let reallyCount = 0;
let basicallyCount = 0;
let veryCount = 0;

storyWords.forEach(word => {
  if(word === "really"){
    reallyCount += 1;
  } else if(word === "very"){
    veryCount += 1;
  } else if(word === "basically"){
    basicallyCount += 1;
  }
});

// You can also use for of loop:
// for (const word of storyWords){
//   if( word === "really"){
//     reallyCount += 1;
// etc....

// Check how many sentences in the array
let sentences = 0;

for (word of storyWords){
    if(word[word.length-1] === "." || word[word.length-1] === "!"){
```

```
      sentences+= 1;
 }
 }

// Log the word count, sentence count and overused words

console.log(`word count: ${storyWords.length}, sentences count: ${sentences},
"really" count: ${reallyCount}, "basically" count: ${basicallyCount}, "very" c
ount: ${veryCount}`)

// Log the array to the console as a single string
console.log(betterWords.join(" "))
```

```
const storyWords = story.split(" ")
// console.log(storyWords.length);

const betterWords = storyWords.filter(word => !unnecessaryWords.includes(word)
);
// console.log(betterWords);

let reallyCount = 0;
let basicallyCount = 0;
let veryCount = 0;

for (const word of storyWords){
  if( word === "really"){
    reallyCount += 1;
  }else if(word === "very"){
    veryCount += 1;
  }else if(word === "basically"){
    basicallyCount += 1;
}
}
console.log(`How many times these words are used? really count: ${reallyCount}
, basically count: ${basicallyCount}, and very count: ${veryCount}`)
```

```
let sentences = 0;

for (word of storyWords){
    if(word[word.length-1] === "." || word[word.length-1] === "!"){
      sentences+= 1;
 }
 }
// console.log(`sentences: ${sentences}`);

// console.log(`word count: ${storyWords.length}, sentences count: ${sentences
}, "really" count: ${reallyCount}, "basically" count: ${basicallyCount}, "very
" count: ${veryCount}`)

// console.log(betterWords.join(" "))
```

7.0 Errors and debugging

7.1 Error Stack Traces

A piece of software, called a *compiler*, is trying to translate your code so that your computer can understand and run it. However, the compiler is coming across a piece of code that it can't interpret. As a result, it throws an error back to you to let you know that it has to stop and why.

This information is logged as an **error stack trace** — a printed message containing information about where the error occurred, what type of error was thrown, and a description of the error.

Reading Error Stack Traces

Now that we know what information we can get from an error stack trace, let's take a look at an example.

```
/home/ccuser/workspace/learn-javascript-debugging-code/app.js:1
myVariable;
^

ReferenceError: myVariable is not defined
...
```

Using this stack trace, let's answer three questions you should ask yourself every time you want to debug an error:

1. **In what line did the error occur?** You can almost always find this information on the first line of the stack trace in the following format `<file path>/<file name>:<line number>`. In this example, the location is `app.js:1`. This means that the error was thrown in the file named `app.js` on line `1`.
2. **What type of error was thrown?** The first word on the fifth line of the stack trace is the type of error that was thrown. In this case, the type of error was `ReferenceError`. We will discuss this error type in the next exercise.
3. **What is the error message?** The rest of the fifth line after the error type provides an error message, describing what went wrong. In this case, the description is `myVariable is not defined`.

7.2 JavaScript Error Types

Now that you can identify the type of error from an error stack trace, you might be wondering what the different types of errors mean.

Here are three common error types:

- **SyntaxError**: This error will be thrown when a typo creates invalid code — code that cannot be interpreted by the compiler. When this error is thrown, scan your code to make sure you properly opened and closed all brackets, braces, and parentheses and that you didn't include any invalid semicolons.
- **ReferenceError**: This error will be thrown if you try to use a variable that does not exist. When this error is thrown, make sure all variables are properly declared.
- **TypeError**: This error will be thrown if you attempt to perform an operation on a value of the wrong type. For example, if we tried to use a string method on a number, it would throw a TypeError.

Solution:

```
myVariable++;
// 1 - What type of error will be thrown on the line above: ReferenceError

const myString = 42;
myString.substring(0);
// 2 - What type of error will be thrown on the line above: TypeError
const myRandomNumber; = Math.random();
// 3 - What type of error will be thrown on the line above: SyntaxError
```

7.3 Debugging Errors

Here's a process for efficiently working through your code's errors one by one:

1. Run your code. Using the first error's stack trace, identify the error's type, description, and location.
2. Go to the file name and line number indicated by the error stack trace. Using the error type and description, identify the bug in your code.
3. Fix the bug and re-run your code.
4. Repeat steps 1-3 until your code no longer throws any errors.

Solution:

```
function isSumBigger(number1, number2, total) {
  const sum = number1 + number2;

  if (sum > total) {
    return true;
  } else {
    return false;
  }
}

// Should return true
console.log(`isSumBigger(1, 3, 2) returns: ${isSumBigger(1, 3, 2)}`);

// Should return false
console.log(`isSumBigger(1, 3, 5) returns: ${isSumBigger(1, 3, 5)}`);
```

7.4 Locating Silent Bugs

Errors thrown by the computer are really useful because they identify the bug type and location for you right away. However, even if your code runs error-free, it is not necessarily bug-free.

You may find that your code is consistently returning incorrect values without throwing any errors. A lack of thrown errors does not mean your code logic is completely correct.

An incredibly powerful tool for locating bugs is a method you likely learned very early on in your JavaScript journey: `console.log()`!

By adding print statements to our code, we can identify where things have gone wrong.

Solution:

```
function capitalizeASingleWord(word) {
  //console.log(word)
  if (word.match(' ')) {
    return null;
  }

  let firstLetter = word.charAt(0);
  const restOfWord = word.slice(1);

  firstLetter = firstLetter.toUpperCase();

  return firstLetter + restOfWord;
}

// Should return "Hey"
console.log(`capitalizeASingleWord('hey') returns: ${capitalizeASingleWord('he
y')}`);

// Should return null
console.log(`capitalizeASingleWord('hey ho') returns: ${capitalizeASingleWord(
'hey ho')}`);
```

7.5 Debugging with console.log()

Let's synthesize our workflow from the previous exercise into a reusable set of debugging steps.

1. Go to the beginning of the malfunctioning code. Print out all starting variables, existing values, and arguments using `console.log()`. If the values are what you expect, move on to the next piece of logic in the code. If not, you have identified a bug and should skip to step 3.
2. After the next piece of logic in your code, add `console.log()` statements to ensure updated variables have the values that you now expect and that the block of code is being executed. If that logic is executing properly, continue repeating this step until you find a line not working as expected, then move to step 3.
3. Fix the identified bug and run your code again. If it now works as expected, you've finished debugging! If not, continue stepping through your code using step 2 until it does.

This might seem like a lot of printing, but once you get into the routine of it, it will be far faster than trying to stare at your code until you find your bugs. Let's try this debugging process again in practice.

```
// Returns the string whose first letter is later in the alphabet. If both fir
st letters are equal, returns null.
function getLaterFirstLetter(string1, string2) {
  const firstLetter1 = string1.charAt(0);
  const firstLetter2 = string2.charAt(0);
  //console.log(firstLetter1, firstLetter2) console log to find the bug

  if (firstLetter1 === firstLetter2) {
    return null;
  } else if (firstLetter1 > firstLetter2) {
    return string1;
  } else {
    return string2;
    //console.log(firstLetter1, firstLetter2) // console log to find the bug
  }
}

// Should return "blueberry"
```

```
console.log(`getLaterFirstLetter('avocado', 'blueberry') returns: ${getLaterFirstLetter('avocado', 'blueberry')}`);

// Should return "zebra"
console.log(`getLaterFirstLetter('zebra', 'aardvark') returns : ${getLaterFirstLetter('zebra', 'aardvark')}`);

// Should return null
console.log(`getLaterFirstLetter('astro', 'afar') returns: ${getLaterFirstLetter('astro', 'afar')}`);
```

7.6 Finding Documentation

Sometimes once you've tracked down a bug, you might still be confused on how to fix it! Whenever you want to know more about how JavaScript works and what it can do, the best place to go is **documentation**. You can find the JavaScript documentation at the MDN JavaScript web docs.

The MDN JavaScript web docs are a powerful resource, but they can be overwhelming because they cover so much information. We encourage you to explore the docs, but often the fastest way to access a specific part of the docs you're interested in is to Google it.

For example, if we wanted more information on the `Number` object's `.isNan()` method, we could Google "MDN isNan" and then click the link to the MDN page. If we were looking to see a list of all of the `String` built-in methods, we might Google "MDN String", click the link to MDN, and then scroll down to the "Methods" section of the documentation.

There are many ways to get to the documentation you are looking for. Find the one that works best for your workflow.

Solution:

```
// Link to String.repeat() documentation: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/repeat
function doubleString(string) {
  return string.repeat(2);
}

// Should return 'echoecho'
console.log("doubleString('echo') returns: " + doubleString('echo'));
```

7.7 Stack Overflow

At this point, you might be thinking to yourself, documentation is good and all, but there's no way it will solve all of my issues! And we totally agree. All programming languages have difficult problems and strange edge cases that appear unexpectedly and are sometimes impossible to solve alone.

If you are ever stuck trying to solve a coding problem, we strongly encourage you to Google for a solution. One of the best sites you will see appear in the search results is Stack Overflow.

Stack Overflow is a question and answer forum where frustrated programmers post issues and other programmers discuss and vote for solutions. Almost always if you have an issue, Stack Overflow has an answer.

For example, say you are stumped trying to write an alphabetize function. If you search "alphabetize string JavaScript" on Google, this Stack Overflow search result will appear. You can quickly scan through the answers on it to see which ones work for you.

Solution:

```
// Returns whether or not the provided string contains a substring of "cake" i
n it.
function containsCake(string) {
    return string.includes("cake")
}

// Should return true
console.log("containsCake('I think cake is my soul mate.') returns: " + contai
nsCake('I think cake is my soul mate.'));

// Should return false
console.log("containsCake('Pie is certainly the coolest dessert.') returns: "
+ containsCake('Pie is certainly the coolest dessert.'));
```

7.8 Debugging Review

You just learned a lot of techniques for helping you get unstuck in all debugging situations. Congratulations! Let's synthesize everything you learned into one debugging process.

1. **Is your code throwing errors?** If so, read the error stack trace for the type, description, and location of the error. Go to the error's location and try to fix.
2. **Is your code broken but not throwing errors?** Walk through your code using `console.log()` statements. When unexpected results occur, isolate the bug and try to fix it.
3. **Did you locate the bug using steps 1 and 2, but can't fix the bug?** Consult documentation to make sure you are using all JavaScript functionality properly. If you are still stuck, Google your issue and consult Stack Overflow for help. Read solutions or post your own Stack Overflow question if none exist on the topic.

Solutions:

```
function isStringPerfectLength(string, minLength, maxLength) {
  const stringLength = string.length;

  if (stringLength < minLength) {
    return false;
  } else if (stringLength > maxLength) {
    return false;
  } else {
    return true;
  }
}

// Should return true
console.log("isStringPerfectLength('Dog', 2, 4) returns: " + isStringPerfectLe
ngth('Dog', 2, 4));

// Should return false
console.log("isStringPerfectLength('Mouse', 2, 4) returns: " + isStringPerfect
Length('Mouse', 2, 4));

// Should return false
console.log("isStringPerfectLength('Cat', 4, 9) returns: " + isStringPerfectLe
ngth('Cat', 4, 9));
```