Basic JavaScript

JavaScript provides eight different data types which are **undefined, null, boolean, string, symbol, bigint, number,** and **object.**

**VAR** can be overwritten:

var myName = “James”;

var myName = “David”;

/\* Result will be variable myName will have a value of David. \*/

Use **LET** instead (when you want the variable to change):

let myName = “James”;

let myName = “David”;

/\* This will cause an error – **GOOD**, u wont override myName by accident \*/

or **CONST** (when you want the variable to remain constant):

const myName = “James”;

**Increment (zwiększenie o 1):**

i++;

to samo co:

i = i + 1;

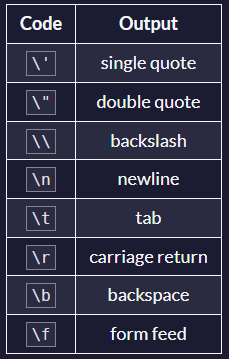
**Descrement (zmniejszenie o 1):**

i--;

to samo co:

i = i – 1;

**Addition (dodawanie): ESCAPE CHARACTERS:**

myVar = myVar +7;

to samo co:

myVar += 7;

**Subtraction (odejmowanie):**

myVar = myVar - 5;

to samo co:

myVar -= 5;

**Multiplication (mnożenie):**

myVar = myVar \* 4;

to samo co:

myVar \*= 4;

**Division (dzielenie):**

myVar = myVar / 8 ;

to samo co:

myVar /= 8;

**Single and double quotes (when it’s already inside a quote):**

Escape characters: \’ \”

const goodStr = 'Jake asks Finn, "Hey, let\'s go on an adventure?" ';

**Arrays:**

const sandwich = ["peanut butter", "jelly", "bread"];

const teams = [["Bulls", 23], ["White Sox", 45]];  *multi-dimensional array*

**PUSH (dodaj na koniec):**

const arr1 = [1, 2, 3];

arr1.push(4, 5);

*arr1 now has the value [1, 2, 3, 4, 5]*

const arr2 = ["Stimpson", "J", "cat"];

arr2.push(["happy", "joy"]);

*arr2 has the value ["Stimpson", "J", "cat", ["happy", "joy"]]*

**UNSHIFT (dodaj na początek):**

const ourArray = ["Stimpson", "J", "cat"];

ourArray.unshift("Happy");

*After the unshift, ourArray would have the value ["Happy", “Stimpson”, "J", "cat"].*

**POP (usuń ostatni element):**

const threeArr = [1, 4, 6];

const oneDown = threeArr.pop();

console.log(oneDown);

console.log(threeArr);

*The first console.log will display the value 6, and the second will display the value [1, 4].*

**SHIFT (usuń pierwszy element):**

const ourArray = ["Stimpson", "J", ["cat"]];

const removedFromOurArray = ourArray.shift();

*removedFromOurArray would have a value of the string Stimpson, and ourArray would have ["J", ["cat"]].*

**Functions:**

function functionName() {

console.log("Hello World");

}

functionName(); *invoke the function*

function testFun(param1, param2) {

console.log(param1, param2);

}

testFun("Hello", "World"); *invoke the function*

function plusThree(num) {

return num + 3;

}

const answer = plusThree(5);

*answer has the value 8.*

*plusThree takes an argument for num and returns a value equal to num + 3.*

**Comparison operators:**

*The most basic operator is the equality operator (* ***==*** *).* *The equality operator compares two values and returns true if they're equivalent or false if they are not.* ***CONVERTS***

1 == 1 *// true*

1 == 2 *// false*

1 == '1' *// true*

"3" == 3 *// true*

*Strict equality (* ***===*** *) is the counterpart to the equality operator (==). However, unlike the equality operator, which attempts to convert both values being compared to a common type, the strict equality operator does not perform a type conversion.* ***DOESN’T CONVERT***

3 === 3 *// true*

3 === '3' *// false*

typeof 3

typeof '3'

*typeof 3 returns the string ‘number’, and typeof '3' returns the string ‘string’.*

*The inequality operator (* ***!=*** *) is the opposite of the equality operator. It means not equal and returns false where equality would return true and vice versa. Like the equality operator, the inequality operator will convert data types of values while comparing.* ***CONVERTS***

1 != 2 *// true*

1 != "1" *// false*

1 != '1' *// false*

1 != true *// false*

0 != false *// false*

*The strict inequality operator (* ***!==*** *) is the logical opposite of the strict equality operator. It means "Strictly Not Equal" and returns false where strict equality would return true and vice versa. The strict inequality operator will not convert data types.*

***DOESN’T CONVERT***

3 !== 3 *// false*

3 !== '3' *// true*

4 !== 3 *// true*

*The greater than operator (* ***>*** *) compares the values of two numbers. If the number to the left is greater than the number to the right, it returns true. Otherwise, it returns false.* ***CONVERTS***

5 > 3 *// true*

7 > '3' *// true*

2 > 3 *// false*

'1' > 9 *// false*

*The greater than or equal to operator (* ***>=*** *) compares the values of two numbers. If the number to the left is greater than or equal to the number to the right, it returns true. Otherwise, it returns false.* ***CONVERTS***

6 >= 6 *// true*

7 >= '3' *// true*

2 >= 3 *// false*

'7' >= 9 *// false*

*The less than operator (* ***<*** *) compares the values of two numbers. If the number to the left is less than the number to the right, it returns true. Otherwise, it returns false.* ***CONVERTS***

2 < 5 *// true*

'3' < 7 *// true*

5 < 5 *// false*

3 < 2 *// false*

'8' < 4 *// false*

*The less than or equal to operator (* ***<=*** *) compares the values of two numbers. If the number to the left is less than or equal to the number to the right, it returns true. If the number on the left is greater than the number on the right, it returns false.* ***CONVERTS***

4 <= 5 *// true*

'7' <= 7 *// true*

5 <= 5 *// true*

3 <= 2 *// false*

'8' <= 4 *// false*

*Sometimes you will need to test more than one thing at a time. The logical and operator (* ***&&*** *) returns true if and only if the operands to the left and right of it are true.*

*The same effect could be achieved by nesting an if statement inside another if.*

if (num > 5) {

if (num < 10) {

return "Yes";

}

}

return "No";

*This code will return Yes if num is greater than 5 and less than 10. The same logic can be written with the logical and operator.*

if (num > 5 && num < 10) {

return "Yes";

}

return "No";

*The logical or operator (* ***||*** *) returns true if either of the operands is true. Otherwise, it returns false.*

*The logical or operator is composed of two pipe symbols: (* ***||*** *). This can typically be found between your Backspace and Enter keys.*

if (num > 10) {

return "No";

}

if (num < 5) {

return "No";

}

return "Yes";

*This code will return Yes if num is between 5 and 10 (5 and 10 included). The same logic can be written with the logical or operator.*

if (num > 10 || num < 5) {

return "No";

}

return "Yes";

When a condition for an **if** statement is true, the block of code following it is executed. What about when that condition is false? Normally nothing would happen. With an **else** statement, an alternate block of code can be executed.

if (num > 10) {

return "Bigger than 10";

} else {

return "10 or Less";

}

If you have multiple conditions that need to be addressed, you can chain if statements together with **else if** statements.

if (num > 15) {

return "Bigger than 15";

} else if (num < 5) {

return "Smaller than 5";

} else {

return "Between 5 and 15";

}

Use the **switch** statement to select one of many code blocks to be executed. Any valid JavaScript statements can be executed inside a case block and will run from the first matched case value until a **break** is encountered.

*In a switch statement you may not be able to specify all possible values as case statements. Instead, you can add the* ***default*** *statement which will be executed if no matching case statements are found. Think of it like the final else statement in an if/else chain.*

**A default statement should be the last case.**

switch (num) {

case value1:

statement1;

break;

case value2:

statement2;

break;

...

default:

defaultStatement;

break;

}

If the break statement is omitted from a switch statement's case, the following case statement(s) are executed until a break is encountered. If you have multiple inputs with the same output, you can represent them in a switch statement like this:

let result = "";

switch (val) {

case 1:

case 2:

case 3:

result = "1, 2, or 3";

break;

case 4:

result = "4 alone";

}

*Cases for 1, 2, and 3 will all produce the same result.*

All comparison operators return a **boolean true or false** **value**. Since **===** returns **true** or **false**, we can return the result of the comparison:

function isEqual(a, b) {

return a === b;

}

**Objects:**

Objects are similar to arrays, except that instead of using indexes to access and modify their data, you access the data in objects through what are called properties.

Objects are useful for storing data in a structured way, and can represent real world objects, like a cat. Here's a sample cat object:

const cat = {

"name": "Whiskers",

"legs": 4,

"tails": 1,

"enemies": ["Water", "Dogs"]

};

In this example, all the properties are stored as strings, such as name, legs, and tails. However, you can also use numbers as properties. You can even omit the quotes for single-word string properties, as follows:

const anotherObject = {

make: "Ford",

5: "five",

"model": "focus"

};

However, if your object has any non-string properties, JavaScript will automatically typecast them as strings.

There are two ways to access the properties of an object: dot notation ( **.** ) and bracket notation ( **[]** ), similar to an array.

Dot notation is what you use when you know the name of the property you're trying to access ahead of time. Here is a sample of using dot notation ( **.** ) to read an object's property:

const myObj = {

prop1: "val1",

prop2: "val2"

};

const prop1val = myObj.prop1;

const prop2val = myObj.prop2;

*prop1val would have a value of the string val1, and prop2val would have a value of the string val2.*

The second way to access the properties of an object is bracket notation ([]). If the property of the object you are trying to access has a space in its name, you will need to use bracket notation.

However, you can still use bracket notation on object properties without spaces.

Here is a sample of using bracket notation to read an object's property:

const myObj = {

"Space Name": "Kirk",

"More Space": "Spock",

"NoSpace": "USS Enterprise"

};

myObj["Space Name"];

myObj['More Space'];

myObj["NoSpace"];

*myObj["Space Name"] would be the string Kirk, myObj['More Space'] would be the string Spock, and myObj["NoSpace"] would be the string USS Enterprise.*

Note that **property names with spaces** in them must be in **quotes** (**single or double**).

Another use of bracket notation on objects is to access a property which is stored as the value of a variable. This can be very useful for iterating through an object's properties or when accessing a lookup table. Here is an example of using a variable to access a property:

const dogs = {

Fido: "Mutt",

Hunter: "Doberman",

Snoopie: "Beagle"

};

const myDog = "Hunter";

const myBreed = dogs[myDog];

console.log(myBreed);

*The string Doberman would be displayed in the console.*

Note that we do not use quotes around the variable name when using it to access the property because we are using the value of the variable, not the name.

After you've created a JavaScript object, you can **update its properties** at any time just like you would update any other variable. You can use either dot or bracket notation to update.

For example, let's look at ourDog:

const ourDog = {

"name": "Camper",

"legs": 4,

"tails": 1,

"friends": ["everything!"]

};

Since he's a particularly happy dog, let's change his name to the string Happy Camper.

Here's how we update his object's name property: *ourDog.name = "Happy Camper";* or *ourDog["name"] = "Happy Camper";* Now when we evaluate ourDog.name, instead of getting Camper, we'll get his new name, Happy Camper.

You can **add new properties** to existing JavaScript objects the same way you would modify them. Here's how we would add a bark property to ourDog:

*ourDog.bark = "bow-wow";*

or

*ourDog["bark"] = "bow-wow";*

Now when we evaluate ourDog.bark, we'll get his bark, bow-wow.

We can also **delete properties** from objects like this:

*delete ourDog.bark;*

Objects can be thought of as a key/value storage, like a dictionary. If you have tabular data, you can use an object to **lookup values** rather than a switch statement or an if/else chain. This is most useful when you know that your input data is limited to a certain range.

Here is an example of an article object:

const article = {

"title": "How to create objects in JavaScript",

"link": "https://www.freecodecamp.org/news/a-complete-guide-to-creating-objects-in-javascript-b0e2450655e8/",

"author": "Kaashan Hussain",

"language": "JavaScript",

"tags": "TECHNOLOGY",

"createdAt": "NOVEMBER 28, 2018"

};

const articleAuthor = article["author"];

const articleLink = article["link"];

const value = "title";

const valueLookup = article[value];

*articleAuthor is the string Kaashan Hussain, articleLink is the string* [*https://www.freecodecamp.org/news/a-complete-guide-to-creating-objects-in-javascript-b0e2450655e8/*](https://www.freecodecamp.org/news/a-complete-guide-to-creating-objects-in-javascript-b0e2450655e8/)*, and* ***valueLookup*** *is the string How to create objects in JavaScript.*

To **check if a property on a given object exists or no**t, you can use the **.hasOwnProperty()** method. someObject.hasOwnProperty(someProperty) returns true or false depending on if the property is found on the object or not. Example:

function checkForProperty(object, property) {

return object.hasOwnProperty(property);

}

checkForProperty({ top: 'hat', bottom: 'pants' }, 'top'); *// true*

checkForProperty({ top: 'hat', bottom: 'pants' }, 'middle'); *// false*

*The first checkForProperty function call returns true, while the second returns false.*

Sometimes you may want to store data in a flexible Data Structure. A JavaScript object is one way to handle flexible data. They allow for arbitrary combinations of *strings, numbers, booleans, arrays, functions, and objects.*

const ourMusic = [

{

"artist": "Daft Punk",

"title": "Homework",

"release\_year": 1997,

"formats": [

"CD",

"Cassette",

"LP"

],

"gold": true

}

];

This is an array which contains one object inside. The object has various pieces of metadata about an album. It also has a **nested** formats **array**. If you want to add more album records, you can do this by adding records to the top level array. Objects hold data in a property, which has a **key-value format.** In the example above, **"artist": "Daft Punk"** is a property that has a **key of artist** and a **value of Daft Punk.**

**Note:** You will need to place a comma after every object in the array, unless it is the last object in the array.

**Loops:**

***while***

You can run the same code multiple times by using a loop. The first type of loop we will learn is called a **while loop** because it runs while a specified condition is true and stops once that condition is no longer true.

const ourArray = [];

let i = 0;

while (i < 5) {

ourArray.push(i);

i++;

}

*In the code example above, the while loop will execute 5 times and append the numbers 0 through 4 to ourArray.*

***for***

The most common type of JavaScript loop is called a **for loop** because it runs for a specific number of times. **For loops** are declared with three optional expressions separated by semicolons:

*for (****a****;* ***b****;* ***c****),* where **a** is the initialization statement, **b** is the condition statement, and **c** is the final expression.

**The initialization statement** is executed one time only before the loop starts. It is typically used to define and setup your loop variable.

**The condition statement** is evaluated at the beginning of every loop iteration and will continue as long as it evaluates to true. When the condition is false at the start of the iteration, the loop will stop executing. This means if the condition starts as false, your loop will never execute.

**The final expression** is executed at the end of each loop iteration, prior to the next condition check and is usually used to increment or decrement your loop counter.

In the following example we initialize with i = 0 and iterate while our condition i < 5 is true. We'll increment i by 1 in each loop iteration with i++ as our final expression.

const ourArray = [];

for (let i = 0; i < 5; i++) {

ourArray.push(i);

}

*ourArray will now have the value [0, 1, 2, 3, 4].*

A common task in JavaScript is to **iterate through the contents of an array.** One way to do that is with a **for loop.** This code will output each element of the array arr to the console:

const arr = [10, 9, 8, 7, 6];

for (let i = 0; i < arr.length; i++) {

console.log(arr[i]);

}

*Remember that arrays have zero-based indexing, which means the last index of the array is length - 1. Our condition for this loop is i < arr.length, which stops the loop when i is equal to length. In this case the last iteration is i === 4 i.e. when i becomes equal to arr.length - 1 and outputs 6 to the console. Then i increases to 5, and the loop terminates because i < arr.length is false.*

If you have a **multi-dimensional array**, you can use the same logic as the prior waypoint to **loop through both the array and any sub-arrays**. Here is an example:

const arr = [

[1, 2], [3, 4], [5, 6]

];

for (let i = 0; i < arr.length; i++) {

for (let j = 0; j < arr[i].length; j++) {

console.log(arr[i][j]);

}

}

*This outputs each sub-element in arr one at a time. Note that for the inner loop, we are checking the .length of arr[i], since arr[i] is itself an array.*

***do … while***

The next type of loop you will learn is called **a do...while loop**. It is called a **do...while** loop because **it will first do one pass of the code inside the loop no matter what**, and then continue to run the loop while the specified condition evaluates to true.

const ourArray = [];

let i = 0;

do {

ourArray.push(i);

i++;

} while (i < 5);

*The example above behaves similar to other types of loops, and the resulting array will look like [0, 1, 2, 3, 4]. However, what makes the do...while different from other loops is how it behaves when the condition fails on the first check. Let's see this in action. Here is a regular while loop that will run the code in the loop as long as i < 5:*

const ourArray = [];

let i = 5;

while (i < 5) {

ourArray.push(i);

i++;

}

In the example ABOVE, we initialize the value of ourArray to an empty array and the value of i to 5. When we execute the while loop, the condition evaluates to false because i is not less than 5, so we do not execute the code inside the loop. The result is that ourArray will end up with no values added to it, and it will still look like [] when all of the code in the example above has completed running.

Now, take a look at a do...while loop:

const ourArray = [];

let i = 5;

do {

ourArray.push(i);

i++;

} while (i < 5);

*In this case, we initialize the value of i to 5, just like we did with the while loop. When we get to the next line, there is no condition to evaluate, so we go to the code inside the curly braces and execute it. We will add a single element to the array and then increment i before we get to the condition check. When we finally evaluate the condition i < 5 on the last line, we see that i is now 6, which fails the conditional check, so we exit the loop and are done. At the end of the above example, the value of ourArray is [5].* ***Essentially, a do...while loop ensures that the code inside the loop will run at least once.***

**Recursion** is the concept that a **function can be expressed in terms of itself.** To help understand this, start by thinking about the following task: multiply the first n elements of an array to create the product of those elements. Using a for loop, you could do this:

function multiply(arr, n) {

let product = 1;

for (let i = 0; i < n; i++) {

product \*= arr[i];

}

return product;

}

*However, notice that multiply(arr, n) == multiply(arr, n - 1) \* arr[n - 1]. That means you can rewrite multiply in terms of itself and never need to use a loop.*

function multiply(arr, n) {

if (n <= 0) {

return 1;

} else {

return multiply(arr, n - 1) \* arr[n - 1];

}

}

*The recursive version of multiply breaks down like this. In the base case, where n <= 0, it returns 1. For larger values of n, it calls itself, but with n - 1. That function call is evaluated in the same way, calling multiply again until n <= 0. At this point, all the functions can return and the original multiply returns the answer.*

**Note:** Recursive functions must have a base case when they return without calling the function again (in this example, when n <= 0), otherwise they can never finish executing.

**Generate Random:**

Random numbers are useful for creating random behavior.

JavaScript has a **Math.random()** function that generates a random decimal number ***between 0 (inclusive) and 1 (exclusive)***. **Thus Math.random() can return a 0 but never return a 1.**

**Note:** Like Storing Values with the Assignment Operator, all function calls will be resolved before the return executes, so we can return the value of the Math.random() function.

You can generate random decimal numbers with Math.random(), but sometimes you need to **generate random whole numbers.** The following process will give you a random whole number less than 20:

1. Use Math.random() to generate a random decimal number.
2. Multiply that random decimal number by 20.
3. Use Math.floor() to round this number down to its nearest whole number.

Remember that Math.random() can never quite return a 1, so it's impossible to actually get 20 since you are rounding down with Math.floor(). This process will give you a random whole number in the range from 0 to 19.

Putting everything together, this is what your code looks like:

Math.floor(Math.random() \* 20);

*You are calling Math.random(), multiplying the result by 20, then passing the value to Math.floor() to round the value down to the nearest whole number.*

You can **generate a random whole number in the range from zero to a given number.** You can also pick a **different lower number for this range.** You'll call your minimum number ***min*** and your maximum number ***max.***

This formula gives a random whole number in the range from min to max. Take a moment to read it and try to understand what this code is doing:

Math.floor(Math.random() \* (max - min + 1)) + min

**parseInt():**

The **parseInt()** function parses a **string** and returns an **integer.** *(* ***liczba całkowita*** *)* Here's an example:

const a = parseInt("007");

*The above function converts the string 007 to* ***the integer 7.*** *If the first character in the string can't be converted into a number, then it returns NaN.*

The **parseInt()** function parses a string and returns an integer. **It takes a second argument for the radix**, which specifies the base of the number in the string. The radix can be an integer between 2 and 36.

The function call looks like:

parseInt(string, radix);

And here's an example:

const a = parseInt("11", 2);

*The radix variable says that 11 is in the* ***binary system, or base 2.*** *This example converts the* ***string 11 to an integer 3.***

The ***conditional operator, also called the ternary operator,*** can be used as a **one line if-else expression.**

The syntax is **a ? b : c**, where **a** is the condition, **b** is the code to run when the condition returns true, and **c** is the code to run when the condition returns false.

The following function uses an if/else statement to check a condition:

function findGreater(a, b) {

if(a > b) {

return "a is greater";

}

else {

return "b is greater or equal";

}

}

*This can be re-written using the conditional operator:*

function findGreater(a, b) {

return a > b ? "a is greater" : "b is greater or equal";

}

In the previous challenge, you used a single conditional operator. You can also **chain them together to check for multiple conditions.**

The following function uses if, else if, and else statements to check multiple conditions:

function findGreaterOrEqual(a, b) {

if (a === b) {

return "a and b are equal";

}

else if (a > b) {

return "a is greater";

}

else {

return "b is greater";

}

}

*The above function can be re-written using multiple conditional operators:*

function findGreaterOrEqual(a, b) {

return (a === b) ? "a and b are equal"

: (a > b) ? "a is greater"

: "b is greater";

}

It is considered best practice to format multiple conditional operators such that **each condition is on a separate line**, as shown above. Using multiple conditional operators without proper indentation may make your code hard to read.

**Recursions:**

**First recursions a little bit higher in the doc.**

In a previous challenge, you learned how to use recursion to replace a for loop. Now, let's look at a more complex function that returns an array of consecutive integers starting with 1 through the number passed to the function.

As mentioned in the previous challenge, there will be a base case. The base case tells the recursive function when it no longer needs to call itself. It is a simple case where the return value is already known. There will also be a recursive call which executes the original function with different arguments. If the function is written correctly, eventually the base case will be reached.

For example, say you want to write a recursive function that returns an array containing the numbers 1 through n. This function will need to accept an argument, n, representing the final number. Then it will need to call itself with progressively smaller values of n until it reaches 1. You could write the function as follows:

function countup(n) {

if (n < 1) {

return [];

} else {

const countArray = countup(n - 1);

countArray.push(n);

return countArray;

}

}

console.log(countup(5));

*The value [1, 2, 3, 4, 5] will be displayed in the console.*

At first, this seems counterintuitive since the value of n decreases, but the values in the final array are increasing. **This happens because the push happens last, after the recursive call has returned.** At the point where n is pushed into the array, countup(n - 1) has already been evaluated and returned [1, 2, ..., n - 1].

**ES6**

**Compare Scopes of the *var* and *let* keywords**

When you declare a variable with the **var** **keyword**, it is **declared globally, or locally if declared inside a function.**

The **let** keyword behaves similarly, but with some extra features. When you declare a variable with the **let keyword inside a block, statement, or expression, its scope is limited to that block, statement, or expression.** For example:

var numArray = [];

for (var i = 0; i < 3; i++) {

numArray.push(i);

}

console.log(numArray);

console.log(i);

*Here the console will display the values [0, 1, 2] and 3.*

With the var keyword, i is declared globally. So when i++ is executed, it updates the global variable. This code is similar to the following:

var numArray = [];

var i;

for (i = 0; i < 3; i++) {

numArray.push(i);

}

console.log(numArray);

console.log(i);

*Here the console will display the values [0, 1, 2] and 3.*

This behavior will cause problems if you were to create a function and store it for later use inside a for loop that uses the i variable. This is because the stored function will always refer to the value of the updated global i variable.

var printNumTwo;

for (var i = 0; i < 3; i++) {

if (i === 2) {

printNumTwo = function() {

return i;

};

}

}

console.log(printNumTwo());

*Here the console will display the value 3.*

As you can see, printNumTwo() prints 3 and not 2. This is because the value assigned to i was updated and the printNumTwo() returns the global i and not the value i had when the function was created in the for loop. The let keyword does not follow this behavior:

let printNumTwo;

for (let i = 0; i < 3; i++) {

if (i === 2) {

printNumTwo = function() {

return i;

};

}

}

console.log(printNumTwo());

console.log(i);

*Here the console will display the value 2, and an error that i is not defined.*

i is not defined because it was not declared in the global scope. It is only declared within the for loop statement. printNumTwo() returned the correct value because three different i variables with unique values (0, 1, and 2) were created by the let keyword within the loop statement.

**Mutate an Array Declared with const**

The const declaration has many use cases in modern JavaScript.

Some developers prefer to assign all their variables using const by default, unless they know they will need to reassign the value. Only in that case, they use let.

However, it is important to understand that **objects (including arrays and functions) assigned to a variable using const are still mutable**. Using the const declaration only prevents reassignment of the variable identifier.

const s = [5, 6, 7];

s = [1, 2, 3];

s[2] = 45;

console.log(s);

*s = [1, 2, 3] will result in an error. After commenting out that line, the console.log will display the value [5, 6, 45].*

As you can see, you can mutate the object [5, 6, 7] itself and the variable s will still point to the altered array [5, 6, 45]. Like all arrays, the array elements in s are mutable, but because const was used, you cannot use the variable identifier s to point to a different array using the assignment operator.

As seen in the previous challenge, const declaration alone doesn't really protect your data from mutation. To ensure your data doesn't change, JavaScript provides a function **Object.freeze** to prevent data mutation. Any attempt at changing the object will be rejected, with an error thrown if the script is running in strict mode.

let obj = {

name:"FreeCodeCamp",

review:"Awesome"

};

Object.freeze(obj);

obj.review = "bad";

obj.newProp = "Test";

console.log(obj);

*The obj.review and obj.newProp assignments will result in errors, because our editor runs in strict mode by default, and the console will display the value { name: "FreeCodeCamp", review: "Awesome" }.*

**Arrow Functions**

In JavaScript, we often don't need to name our functions, especially when passing a function as an argument to another function. Instead, we create inline functions. We don't need to name these functions because we do not reuse them anywhere else. To achieve this, we often use the following syntax:

const myFunc = function() {

const myVar = "value";

return myVar;

}

ES6 provides us with the syntactic sugar to not have to write anonymous functions this way. Instead, you can use **arrow function syntax:**

const myFunc = () => {

const myVar = "value";

return myVar;

}

*When there is no function body, and only a return value, arrow function syntax allows you to omit the keyword return as well as the brackets surrounding the code. This helps simplify smaller functions into one-line statements:*

const myFunc = () => "value";

*This code will still return the string value by default.*

Just like a regular function, you can pass arguments into an arrow function.

const doubler = (item) => item \* 2;

doubler(4);

*doubler(4) would return the value 8.*

If an arrow function has a single parameter, the parentheses enclosing the parameter may be omitted.

const doubler = item => item \* 2;

It is possible to pass more than one argument into an arrow function.

const multiplier = (item, multi) => item \* multi;

multiplier(4, 2);

*multiplier(4, 2) would return the value 8.*

In order to help us create more flexible functions, ES6 introduces **default parameters for functions.** Check out this code:

const greeting = (name = "Anonymous") => "Hello " + name;

console.log(greeting("John"));

console.log(greeting());

*The console will display the strings Hello John and Hello Anonymous.*

The **default parameter kicks in when the argument is not specified (it is undefined)**. As you can see in the example above, the parameter name will receive its default value Anonymous when you do not provide a value for the parameter. **You can add default values for as many parameters as you want.**

In order to help us create more flexible functions, ES6 introduces the **rest parameter** for function parameters. With the **rest** parameter, you can create functions that take a **variable number of arguments. These arguments are stored in an array** that can be accessed later from inside the function. Check out this code:

function howMany(...args) {

return "You have passed " + args.length + " arguments.";

}

console.log(howMany(0, 1, 2));

console.log(howMany("string", null, [1, 2, 3], { }));

*The console would display the strings You have passed 3 arguments. and You have passed 4 arguments..*

The rest parameter eliminates the need to use the arguments object and allows us to use array methods on the array of parameters passed to the function howMany.

ES6 introduces the **spread operator,** which allows us to expand arrays and other expressions in places where multiple parameters or elements are expected. The ES5 code below uses apply() to compute the maximum value in an array:

var arr = [6, 89, 3, 45];

var maximus = Math.max.apply(null, arr);

*maximus would have a value of 89.*

We had to use Math.max.apply(null, arr) because Math.max(arr) returns NaN. Math.max() expects comma-separated arguments, but not an array. The spread operator makes this syntax much better to read and maintain.

const arr = [6, 89, 3, 45];

const maximus = Math.max(...arr);

*maximus would have a value of 89.*

...arr returns an unpacked array. In other words, it spreads the array. However, the spread operator only works in-place, like in an argument to a function or in an array literal. For example:

const spreaded = [...arr];

However, the following code will not work:

const spreaded = ...arr;

**Destructuring Assignment:**

Destructuring assignment is special syntax introduced in ES6, for neatly assigning values taken directly from an object. Consider the following ES5 code:

const user = { name: 'John Doe', age: 34 };

const name = user.name;

const age = user.age;

*name would have a value of the string John Doe, and age would have the number 34.*

Here's an equivalent assignment statement using the ES6 destructuring syntax:

const { name, age } = user;

*Again, name would have a value of the string John Doe, and age would have the number 34.*

Here, the name and age variables will be created and assigned the values of their respective values from the user object. You can see how much cleaner this is. You can extract as many or few values from the object as you want.

Restructuring allows you to **assign a new variable name when extracting values.** You can do this by putting the new name after a colon when assigning the value. Using the same object from the last example:

const user = { name: 'John Doe', age: 34 };

Here's how you can give new variable names in the assignment:

const { name: userName, age: userAge } = user;

*You may read it as "get the value of user.name and assign it to a new variable named userName" and so on. The value of userName would be the string John Doe, and the value of userAge would be the number 34.*

You can use the same principles from the previous two lessons to **destructure values from nested objects.** Using an object similar to previous examples:

const user = {

johnDoe: {

age: 34,

email: 'johnDoe@freeCodeCamp.com'

}

};

Here's how to extract the values of object properties and assign them to variables with the same name:

const { johnDoe: { age, email }} = user;

And here's how you can assign an object properties' values to variables with different names:

const { johnDoe: { age: userAge, email: userEmail }} = user;

ES6 makes **destructuring arrays** as easy as destructuring objects.

One key difference between the spread operator and array destructuring is that the spread operator unpacks all contents of an array into a comma-separated list. Consequently, you cannot pick or choose which elements you want to assign to variables.

Destructuring an array lets us do exactly that:

const [a, b] = [1, 2, 3, 4, 5, 6];

console.log(a, b);

*The console will display the values of a and b as 1, 2.*

The variable a is assigned the first value of the array, and b is assigned the second value of the array.

We can also access the value at any index in an array with destructuring by using commas to reach the desired index:

const [a, b,,, c] = [1, 2, 3, 4, 5, 6];

console.log(a, b, c);

*The console will display the values of a, b, and c as 1, 2, 5.*

In some situations involving array destructuring, we might want to **collect the rest of the elements into a separate array.** The result is similar to Array.prototype.slice(), as shown below:

const [a, b, ...arr] = [1, 2, 3, 4, 5, 7];

console.log(a, b);

console.log(arr);

*The console would display the values 1, 2 and [3, 4, 5, 7].*

Variables a and b take the first and second values from the array. After that, because of the rest syntax presence, arr gets the rest of the values in the form of an array. The rest element only works correctly as the last variable in the list. As in, you cannot use the rest syntax to catch a subarray that leaves out the last element of the original array.

In some cases, you can **destructure the object in a function argument itself.** Consider the code below:

const profileUpdate = (profileData) => {

const { name, age, nationality, location } = profileData;

}

*This effectively destructures the object sent into the function.*

This can also be done in-place:

const profileUpdate = ({ name, age, nationality, location }) => {

}

*When profileData is passed to the above function, the values are destructured from the function parameter for use within the function.*

**Template Literals:**

A new feature of ES6 is **the template literal.** This is a special type of string that makes creating complex strings easier.

**Template literals** allow you to create multi-line strings and to use string interpolation features to create strings.Consider the code below:

const person = {

name: "Zodiac Hasbro",

age: 56

};

const greeting = `Hello, my name is ${person.name}!

I am ${person.age} years old.`;

console.log(greeting);

*The console will display the strings Hello, my name is Zodiac Hasbro! and I am 56 years old. .*

A lot of things happened there. Firstly, the example uses backticks (`), not quotes (' or "), to wrap the string. Secondly, notice that the string is multi-line, both in the code and the output. This saves inserting \n within strings. The ${variable} syntax used above is a placeholder. Basically, you won't have to use concatenation with the + operator anymore. To add variables to strings, you just drop the variable in a template string and wrap it with ${ and }. Similarly, you can include other expressions in your string literal, for example ${a + b}. This new way of creating strings gives you more flexibility to create robust strings.

ES6 adds some nice support for easily **defining object literals.** Consider the following code:

const getMousePosition = (x, y) => ({

x: x,

y: y

});

*getMousePosition is a simple function that returns an object containing two properties.*

ES6 provides the syntactic sugar to eliminate the redundancy of having to write x: x. You can simply write x once, and it will be converted tox: x (or something equivalent) under the hood. Here is the same function from above rewritten to use this new syntax:

const getMousePosition = (x, y) => ({ x, y });

When **defining functions within objects in ES5**, we have to use the keyword function as follows:

const person = {

name: "Taylor",

sayHello: function() {

return `Hello! My name is ${this.name}.`;

}

};

**With ES6, you can remove the function keyword and colon altogether when defining functions in objects.** Here's an example of this syntax:

const person = {

name: "Taylor",

sayHello() {

return `Hello! My name is ${this.name}.`;

}

};

**Class:**

ES6 provides a new syntax to **create objects,** usingthe **class** keyword.

In ES5, an object can be created by defining a constructor function and using the new keyword to instantiate the object.

In ES6, a class declaration has a constructor method that is invoked with the new keyword. If the constructor method is not explicitly defined, then it is implicitly defined with no arguments.

// Explicit constructor

class SpaceShuttle {

constructor(targetPlanet) {

this.targetPlanet = targetPlanet;

}

takeOff() {

console.log("To " + this.targetPlanet + "!");

}

}

// Implicit constructor

class Rocket {

launch() {

console.log("To the moon!");

}

}

const zeus = new SpaceShuttle('Jupiter');

// prints To Jupiter! in console

zeus.takeOff();

const atlas = new Rocket();

// prints To the moon! in console

atlas.launch();

It should be noted that the class keyword declares a new function, to which a constructor is added. This constructor is invoked when new is called to create a new object.

**Note:** UpperCamelCase should be used by convention for ES6 class names, as in SpaceShuttle used above.

You can obtain values from an object and set the value of a property within an object.

These are classically called **getters** and **setters.**

**Getter functions** are meant to simply return (get) the value of an object's private variable to the user without the user directly accessing the private variable.

**Setter functions** are meant to modify (set) the value of an object's private variable based on the value passed into the setter function. This change could involve calculations, or even overwriting the previous value completely.

class Book {

constructor(author) {

this.\_author = author;

}

// getter

get writer() {

return this.\_author;

}

// setter

set writer(updatedAuthor) {

this.\_author = updatedAuthor;

}

}

const novel = new Book('anonymous');

console.log(novel.writer);

novel.writer = 'newAuthor';

console.log(novel.writer);

*The console would display the strings anonymous and newAuthor.*

Notice the syntax used to invoke the getter and setter. They do not even look like functions. Getters and setters are important because they hide internal implementation details.

**Note:** It is convention to precede the name of a private variable with an underscore (\_). However, the practice itself does not make a variable private.

**Module script:**

JavaScript started with a small role to play on an otherwise mostly HTML web. Today, it’s huge, and some websites are built almost entirely with JavaScript. In order to make JavaScript more modular, clean, and maintainable; ES6 introduced a way to **easily share code among JavaScript files.** This involves exporting parts of a file for use in one or more other files, and importing the parts you need, where you need them. In order to take advantage of this functionality, you need to create a script in your HTML document with a type of module. Here’s an example:

<script type="module" src="filename.js"></script>

A script that uses this module type can now use the import and export features you will learn about in the upcoming challenges.

Imagine a file called math\_functions.js that contains several functions related to mathematical operations. One of them is stored in a variable, add, that takes in two numbers and returns their sum. **You want to use this function in several different JavaScript files. In order to share it with these other files, you first need to export it.**

export const add = (x, y) => {

return x + y;

}

*The above is a common way to export a single function, but you can achieve the same thing like this:*

const add = (x, y) => {

return x + y;

}

export { add };

When you export a variable or function, you can import it in another file and use it without having to rewrite the code. You can export multiple things by repeating the first example for each thing you want to export, or by placing them all in the export statement of the second example, like this:

export { add, subtract };

**Import** allows you to choose which parts of a file or module to load. In the previous lesson, the examples exported add from the math\_functions.js file. Here's how you can import it to use in another file:

import { add } from './math\_functions.js';

*Here, import will find add in math\_functions.js, import just that function for you to use, and ignore the rest. The ./ tells the import to look for the math\_functions.js file in the same folder as the current file. The relative file path (./) and file extension (.js) are required when using import in this way.*

You can import more than one item from the file by adding them in the import statement like this:

import { add, subtract } from './math\_functions.js';

Suppose you have a file and you wish to **import all of its contents into the current file.** This can be done with the import \* as syntax. Here's an example where the contents of a file named math\_functions.js are imported into a file in the same directory:

import \* as myMathModule from "./math\_functions.js";

*The above import statement will create an object called myMathModule. This is just a variable name, you can name it anything. The object will contain all of the exports from math\_functions.js in it, so you can access the functions like you would any other object property.*

Here's how you can use the add and subtract functions that were imported:

myMathModule.add(2,3);

myMathModule.subtract(5,3);

In the export lesson, you learned about the syntax referred to as a named export. This allowed you to make multiple functions and variables available for use in other files.

There is another export syntax you need to know, known as **export default.** Usually you will use this syntax if only one value is being exported from a file. It is also used to create a fallback value for a file or module.

Below are examples using export default:

export default function add(x, y) {

return x + y;

}

export default function(x, y) {

return x + y;

}

*The first is a named function, and the second is an anonymous function.*

Since export default is used to declare a fallback value for a module or file, you can only have one value be a default export in each module or file. Additionally, you cannot use export default with var, let, or const

In the last challenge, you learned about export default and its uses. To **import a default export**, you need to use a different import syntax. In the following example, add is the default export of the math\_functions.js file. Here is how to import it:

import add from "./math\_functions.js";

*The syntax differs in one key place. The imported value, add, is not surrounded by curly braces ( {} ). add here is simply a variable name for whatever the default export of the math\_functions.js file is. You can use any name here when importing a default.*

**Promise with resolve and reject:**

**A promise** in JavaScript is exactly what it sounds like - **you use it to make a promise to do something**, usually asynchronously. When the task completes, you either fulfill your promise or fail to do so. Promise is a constructor function, so you need to use the new keyword to create one. It takes a function, as its argument, with two parameters - resolve and reject. These are methods used to determine the outcome of the promise.

The syntax looks like this:

const myPromise = new Promise((resolve, reject) => {

});

**A promise** has three states: **pending, fulfilled, and rejected.** The promise you created in the last challenge is forever stuck in the pending state because you did not add a way to complete the promise. The resolve and reject parameters given to the promise argument are used to do this. resolve is used when you want your promise to succeed, and reject is used when you want it to fail. These are methods that take an argument, as seen below.

const myPromise = new Promise((resolve, reject) => {

if(condition here) {

resolve("Promise was fulfilled");

} else {

reject("Promise was rejected");

}

});

*The example above uses strings for the argument of these functions, but it can really be anything. Often, it might be an object, that you would use data from, to put on your website or elsewhere.*

**Promises are most useful when you have a process that takes an unknown amount of time in your code (i.e. something asynchronous),** **often a server request.** When you make a server request it takes some amount of time, and after it completes you usually want to do something with the response from the server. This can be achieved by using the then method. The **then** method is executed immediately after your promise **is fulfilled with resolve.** Here’s an example:

myPromise.then(result => {

});

*result comes from the argument given to the resolve method.*

**catch** is the method used when your promise **has been rejected.** It is executed immediately after a **promise's reject method is called.** Here’s the syntax:

myPromise.catch(error => {

});

*error is the argument passed in to the reject method.*

Regular Expressions:

**Test() method:**

Regular expressions are used in programming languages to match parts of strings. You create patterns to help you do that matching.

If you want to find the word the in the string The dog chased the cat, you could use the following regular expression: /the/. Notice that quote marks are not required within the regular expression.

JavaScript has multiple ways to use regexes. One way to test a regex is using the **.test()** method. **The .test() method takes the regex, applies it to a string (which is placed inside the parentheses), and returns true or false if your pattern finds something or not.**

let testStr = "freeCodeCamp";

let testRegex = /Code/;

testRegex.test(testStr);

*The test method here returns true.*

In the last challenge, you searched for the word Hello using the regular expression /Hello/. That regex searched for a **literal match** of the string Hello. Here's another example searching for a literal match of the string Kevin:

let testStr = "Hello, my name is Kevin.";

let testRegex = /Kevin/;

testRegex.test(testStr);

*This test call will return true.*

Any other forms of Kevin will not match. For example, the regex /Kevin/ will not match kevin or KEVIN.

let wrongRegex = /kevin/;

wrongRegex.test(testStr);

*This test call will return false.*

A future challenge will show how to match those other forms as well.

Using regexes like /coding/, you can look for the pattern coding in another string.

This is powerful to search single strings, but it's limited to only one pattern. **You can search for multiple patterns using the alternation or OR operator: |.**

This operator matches patterns either before or after it. For example, if you wanted to match the strings yes or no, the regex you want is /yes|no/.

You can also search for more than just two patterns. You can do this by adding more patterns with more OR operators separating them, like /yes|no|maybe/.

Up until now, you've looked at regexes to do literal matches of strings. But sometimes, you might want to also **match case differences.**

Case (or sometimes letter case) is the difference between uppercase letters and lowercase letters. Examples of uppercase are A, B, and C. Examples of lowercase are a, b, and c.

You can match both cases using what is called a **flag.** There are other flags but here you'll focus on the flag that ignores case - **the i flag.** You can use it by appending it to the regex. An example of using this flag is **/ignorecase/i.** This regex can match the strings ignorecase, igNoreCase, and IgnoreCase.

**Match() method:**

So far, you have only been checking if a pattern exists or not within a string. You can also extract the actual matches you found with the **.match()** method.

To use the **.match()** method, apply the method on a string and pass in the regex inside the parentheses. Here's an example:

"Hello, World!".match(/Hello/);

let ourStr = "Regular expressions";

let ourRegex = /expressions/;

ourStr.match(ourRegex);

*Here the first match would return ["Hello"] and the second would return ["expressions"].*

**Note** that the .match syntax is the "opposite" of the .test method you have been using thus far:

'string'.match(/regex/);

/regex/.test('string');

So far, you have only been able to extract or search a pattern once.

let testStr = "Repeat, Repeat, Repeat";

let ourRegex = /Repeat/;

testStr.match(ourRegex);

*Here match would return ["Repeat"].*

**To search or extract a pattern more than once,** you can use the global search flag: **g**.

let repeatRegex = /Repeat/g;

testStr.match(repeatRegex);

*And here match returns the value ["Repeat", "Repeat", "Repeat"]*

**Note:** You can have multiple flags on your regex like /search/**gi**

Sometimes you won't (or don't need to) know the exact characters in your patterns. Thinking of all words that match, say, a misspelling would take a long time. Luckily, you can save time using the **wildcard character**: **.**

The wildcard character **.** will match any one character. The wildcard is also called **dot** and **period**. You can use the wildcard character just like any other character in the regex. For example, if you wanted to match hug, huh, hut, and hum, you can use the regex /hu./ to match all four words.

let humStr = "I'll hum a song";

let hugStr = "Bear hug";

let huRegex = /hu./;

huRegex.test(humStr);

huRegex.test(hugStr);

*Both of these test calls would return true.*

You learned how to match literal patterns (/literal/) and wildcard character (/./). Those are the extremes of regular expressions, where **one finds exact matches and the other matches everything.** **There are options that are a balance between the two extremes.**

You can search for a literal pattern with some flexibility with character classes. Character classes allow you to define a group of characters you wish to match by placing them inside square ([ and ]) brackets.

For example, you want to match bag, big, and bug but not bog. You can create the regex /b[aiu]g/ to do this. The [aiu] is the character class that will only match the characters a, i, or u.

let bigStr = "big";

let bagStr = "bag";

let bugStr = "bug";

let bogStr = "bog";

let bgRegex = /b[aiu]g/;

bigStr.match(bgRegex);

bagStr.match(bgRegex);

bugStr.match(bgRegex);

bogStr.match(bgRegex);

*In order, the four match calls would return the values ["big"], ["bag"], ["bug"], and null.*

You saw how you can use character sets to specify a group of characters to match, but that's a lot of typing when you need to match a large range of characters (for example, every letter in the alphabet). Fortunately, there is a built-in feature that makes this short and simple.

Inside a character set, **you can define a range of characters to match using a hyphen character**: **-**.

For example, to match lowercase letters a through e you would use [a-e].

let catStr = "cat";

let batStr = "bat";

let matStr = "mat";

let bgRegex = /[a-e]at/;

catStr.match(bgRegex);

batStr.match(bgRegex);

matStr.match(bgRegex);

*In order, the three match calls would return the values ["cat"], ["bat"], and null.*

Using the hyphen **(-)** to match a range of characters is not limited to letters. **It also works to match a range of numbers.**

For example, /[0-5]/ matches any number between 0 and 5, including the 0 and 5.

Also, it is possible to combine a range of letters and numbers in a single character set.

let jennyStr = "Jenny8675309";

let myRegex = /[a-z0-9]/ig;

jennyStr.match(myRegex);

So far, you have created a set of characters that you want to match, but you could also create a set of characters that **you do not want to match.** These types of character sets are called **negated character sets.**

To create a negated character set, you place a caret character ( **^** ) after the opening bracket and before the characters you do not want to match.

For example, /[^aeiou]/gi matches all characters that are not a vowel. Note that characters like ., !, [, @, / and white space are matched - the negated vowel character set only excludes the vowel characters.

Sometimes, you need to match a character (or group of characters) that **appears one or more times in a row.** This means it occurs at least once, and may be repeated.

You can use the **+** character to check if that is the case. **Remember, the character or pattern has to be present consecutively.** That is, the character has to repeat one after the other.

For example, /a+/g would find one match in abc and return ["a"]. Because of the +, it would also find a single match in aabc and return ["aa"].

If it were instead checking the string abab, it would find two matches and return ["a", "a"] because the a characters are not in a row - there is a b between them. Finally, since there is no a in the string bcd, it wouldn't find a match.

The last challenge used the plus + sign to look for characters that occur one or more times. There's also an option that matches characters **that occur zero or more times.**

The character to do this is the asterisk or star: **\***.

let soccerWord = "gooooooooal!";

let gPhrase = "gut feeling";

let oPhrase = "over the moon";

let goRegex = /go\*/;

soccerWord.match(goRegex);

gPhrase.match(goRegex);

oPhrase.match(goRegex);

*In order, the three match calls would return the values ["goooooooo"], ["g"], and null.*

In regular expressions, a **greedy match** finds the longest possible part of a string that fits the regex pattern and returns it as a match. The alternative is called a **lazy match**, which finds the smallest possible part of the string that satisfies the regex pattern.

You can apply the regex /t[a-z]\*i/ to the string "titanic". This regex is basically a pattern that starts with t, ends with i, and has some letters in between.

Regular expressions are by default **greedy**, so the match would return ["titani"]. It finds the largest sub-string possible to fit the pattern.

However, you can use the **?** character to change it to **lazy** matching. "titanic" matched against the adjusted regex of /t[a-z]\*?i/ returns ["ti"].

**Note:** Parsing HTML with regular expressions should be avoided, but pattern matching an HTML string with regular expressions is completely fine.

Prior challenges showed that regular expressions can be used to look for a number of matches. They are also used to search for **patterns in specific positions in strings.**

In an earlier challenge, you used the caret character ( **^** ) **inside a character set** to create a negated character set in the form [^thingsThatWillNotBeMatched]. **Outside of a character set**, the caret is used to search for patterns at the beginning of strings.

let firstString = "Ricky is first and can be found.";

let firstRegex = /^Ricky/;

firstRegex.test(firstString);

let notFirst = "You can't find Ricky now.";

firstRegex.test(notFirst);

*The first test call would return true, while the second would return false.*

In the last challenge, you learned to use the caret character to search for patterns at the beginning of strings. There is also a way to **search for patterns at the end of strings.**

You can search the end of strings using the dollar sign character **$ at the end of the regex.**

let theEnding = "This is a never ending story";

let storyRegex = /story$/;

storyRegex.test(theEnding);

let noEnding = "Sometimes a story will have to end";

storyRegex.test(noEnding);

*The first test call would return true, while the second would return false.*

Using character classes, you were able to search for all letters of the alphabet with [a-z]. This kind of character class is common enough that there is a shortcut for it, although it includes a few extra characters as well.

The closest character class in JavaScript to match the alphabet is **\w**. This shortcut is equal to **[A-Za-z0-9\_]**. This character class matches upper and lowercase letters plus numbers. Note, this character class also includes the underscore character ( \_ ).

let longHand = /[A-Za-z0-9\_]+/;

let shortHand = /\w+/;

let numbers = "42";

let varNames = "important\_var";

longHand.test(numbers);

shortHand.test(numbers);

longHand.test(varNames);

shortHand.test(varNames);

*All four of these test calls would return true.*

These shortcut character classes are also known as shorthand character classes.

You've learned that you can use a shortcut to match alphanumerics [A-Za-z0-9\_] using \w. A natural pattern you might want to search for is the **opposite of alphanumerics.**

You can search for the opposite of the \w with **\W**. Note, the opposite pattern uses a capital letter. This shortcut is the same as **[^A-Za-z0-9\_]**.

let shortHand = /\W/;

let numbers = "42%";

let sentence = "Coding!";

numbers.match(shortHand);

sentence.match(shortHand);

*The first match call would return the value ["%"] and the second would return ["!"].*

You've learned shortcuts for common string patterns like alphanumerics. Another common pattern is looking for just digits or numbers.

The shortcut to **look** **for** **digit** **characters** is **\d**, with a lowercase d. This is equal to the character class **[0-9]**, which looks for **a single character of any number between zero and nine**.

The last challenge showed how to search for digits using the shortcut \d with a lowercase d. You can also **search for non-digits** using a similar shortcut that uses an uppercase D instead.

The shortcut to look for non-digit characters is **\D**. This is equal to the character class **[^0-9]**, which **looks for a single character that is not a number between zero and nine.**

The challenges so far have covered matching letters of the alphabet and numbers. You can also **match the whitespace or spaces between letters.**

You can **search for whitespace** using **\s**, which is a lowercase s. This pattern not only matches whitespace, but also carriage return, tab, form feed, and new line characters. You can think of it as similar to the character class [ \r\t\f\n\v].

let whiteSpace = "Whitespace. Whitespace everywhere!"

let spaceRegex = /\s/g;

whiteSpace.match(spaceRegex);

*This match call would return [" ", " "].*

You learned about searching for whitespace using \s, with a lowercase s. You can also search for **everything except whitespace.**

**Search for non-whitespace** using **\S**, which is an uppercase s. This pattern will not match whitespace, carriage return, tab, form feed, and new line characters. You can think of it being similar to the character class [^ \r\t\f\n\v].

let whiteSpace = "Whitespace. Whitespace everywhere!"

let nonSpaceRegex = /\S/g;

whiteSpace.match(nonSpaceRegex).length;

*The value returned by the .length method would be 32.*

Recall that you use the plus sign + to look for one or more characters and the asterisk \* to look for zero or more characters. These are convenient but sometimes you want to **match a certain range of patterns.**

You can specify the lower and upper number of patterns with **quantity specifiers**. Quantity specifiers are used with **curly brackets** **({ and }).** You put two numbers between the curly brackets - for the lower and upper number of patterns.

For example, to match only the letter a appearing between 3 and 5 times in the string ah, your regex would be /a{3,5}h/.

let A4 = "aaaah";

let A2 = "aah";

let multipleA = /a{3,5}h/;

multipleA.test(A4);

multipleA.test(A2);

*The first test call would return true, while the second would return false.*

You can specify the lower and upper number of patterns with quantity specifiers using curly brackets. Sometimes you **only** want to **specify the lower number of patterns with no upper limit.**

To only specify the lower number of patterns, **keep the first number followed by a comma.**

For example, to match only the string hah with the letter a appearing at least 3 times, your regex would be /ha{3,}h/.

let A4 = "haaaah";

let A2 = "haah";

let A100 = "h" + "a".repeat(100) + "h";

let multipleA = /ha{3,}h/;

multipleA.test(A4);

multipleA.test(A2);

multipleA.test(A100);

*In order, the three test calls would return true, false, and true.*

You can specify the lower and upper number of patterns with quantity specifiers using curly brackets. **Sometimes you only want a specific number of matches.**

To specify a certain number of patterns, **just have that one number between the curly brackets.**

For example, to match only the word hah with the letter a 3 times, your regex would be /ha{3}h/.

let A4 = "haaaah";

let A3 = "haaah";

let A100 = "h" + "a".repeat(100) + "h";

let multipleHA = /ha{3}h/;

multipleHA.test(A4);

multipleHA.test(A3);

multipleHA.test(A100);

*In order, the three test calls would return false, true, and false.*

Sometimes the patterns you want to **search for may have parts of it that may or may not exist.** However, it may be important to check for them nonetheless.

You can specify the possible existence of an element with a **question mark, ?.** This checks for zero or one of the **preceding element.** You can think of this symbol as saying the previous element is optional.

For example, there are slight differences in American and British English and you can use the question mark to match both spellings.

let american = "color";

let british = "colour";

let rainbowRegex= /colou?r/;

rainbowRegex.test(american);

rainbowRegex.test(british);

*Both uses of the test method would return true.*

**Lookaheads** are patterns that tell JavaScript to look-ahead in your string to check for patterns further along. This can be useful when you want to search for multiple patterns over the same string.

There are two kinds of lookaheads: **positive lookahead** and **negative lookahead**.

A **positive lookahead** will look to make sure the element in the search pattern is there, but won't actually match it. A positive lookahead is used as **(?=...)** where the ... is the required part that is not matched.

On the other hand, a **negative lookahead** will look to make sure the element in the search pattern is not there. A negative lookahead is used as **(?!...)** where the ... is the pattern that you do not want to be there. The rest of the pattern is returned if the negative lookahead part is not present.

Lookaheads are a bit confusing but some examples will help.

let quit = "qu";

let noquit = "qt";

let quRegex= /q(?=u)/;

let qRegex = /q(?!u)/;

quit.match(quRegex);

noquit.match(qRegex);

*Both of these match calls would return ["q"].*

**A more practical use** **of lookaheads** is to check two or more patterns in one string. Here is a (naively) simple password checker that looks for between 3 and 6 characters and at least one number:

let password = "abc123";

let checkPass = /(?=\w{3,6})(?=\D\*\d)/;

checkPass.test(password);

Sometimes we want to check for groups of characters using a Regular Expression and to achieve that we use parentheses **()**.

If you want to find either Penguin or Pumpkin in a string, you can use the following Regular Expression: /P(engu|umpk)in/g

Then check whether the desired string groups are in the test string by using the test() method.

let testStr = "Pumpkin";

let testRegex = /P(engu|umpk)in/;

testRegex.test(testStr);

*The test method here would return true.*

Say you want to **match a word that occurs** **multiple times** like below.

let repeatStr = "row row row your boat";

You could use /row row row/, but what if you don't know the specific word repeated?

**Capture groups can be used to find repeated substrings.**

**Capture groups** are constructed by enclosing the regex pattern to be captured in parentheses. In this case, the goal is to capture a word consisting of alphanumeric characters so the capture group will be \w+ enclosed by parentheses: /(\w+)/.

The substring matched by the group is saved to a temporary "variable", which can be accessed within the same regex using a backslash and the number of the capture group (e.g. \1). Capture groups are automatically numbered by the position of their opening parentheses (left to right), starting at 1.

The example below matches a word that occurs thrice separated by spaces:

let repeatRegex = /(\w+) \1 \1/;

repeatRegex.test(repeatStr); // Returns true

repeatStr.match(repeatRegex); // Returns ["row row row", "row"]

*Using the .match() method on a string will return an array with the matched substring, along with its captured groups.*

Searching is useful. However, you can make searching even more powerful when it also **changes (or replaces) the text you match.**

You can **search and replace text** in a string using **.replace()** on a string. The inputs for **.replace()** is first the regex pattern you want to search for. The second parameter is the string to replace the match or a function to do something.

let wrongText = "The sky is silver.";

let silverRegex = /silver/;

wrongText.replace(silverRegex, "blue");

*The replace call would return the string The sky is blue. .*

You can also access capture groups in the replacement string with dollar signs ($).

"Code Camp".replace(/(\w+)\s(\w+)/, '$2 $1');

*The replace call would return the string Camp Code.*

**Debugging:**

The **console.log()** method, which "prints" the output of what's within its parentheses to the console, will likely be the most helpful debugging tool. Placing it at strategic points in your code can show you the intermediate values of variables. It's good practice to have an idea of what the output should be before looking at what it is. Having check points to see the status of your calculations throughout your code will help narrow down where the problem is.

Here's an example to print the string Hello world! to the console:

console.log('Hello world!');

There are many methods to use with console to output messages. log, warn, and clear to name a few.

**console.clear();**

You can use **typeof** to check the data structure, or type, of a variable. This is useful in debugging when working with multiple data types. If you think you're adding two numbers, but one is actually a string, the results can be unexpected. Type errors can lurk in calculations or function calls. Be careful especially when you're accessing and working with external data in the form of a JavaScript Object Notation (JSON) object.

Here are some examples using typeof:

console.log(typeof "");

console.log(typeof 0);

console.log(typeof []);

console.log(typeof {});

*In order, the console will display the strings string, number, object, and object.*

JavaScript recognizes seven primitive (immutable) data types: Boolean, Null, Undefined, Number, String, Symbol (new with ES6), and BigInt (new with ES2020), and one type for mutable items: Object. Note that in JavaScript, arrays are technically a type of object.

The console.log() and typeof methods are the two primary ways to check intermediate values and types of program output. Now it's time to get into the common forms that bugs take. One syntax-level issue that fast typers can commiserate with is **the humble spelling error.**

Transposed, missing, or miscapitalized characters in a variable or function name will have the browser looking for an object that doesn't exist - and complain in the form of a **reference error**. **JavaScript variable and function names are case-sensitive.**

Branching programs, i.e. ones that do different things if certain conditions are met, rely on if, else if, and else statements in JavaScript. The condition sometimes takes the form of testing whether a result is equal to a value.

This logic is spoken (in English, at least) as "if x equals y, then ..." which can literally translate into code using the =, or assignment operator. This leads to unexpected control flow in your program.

As covered in previous challenges, the assignment operator (=) in JavaScript assigns a value to a variable name. And the == and === operators check for equality (the triple === tests for strict equality, meaning both value and type are the same).

The code below assigns x to be 2, which evaluates as true. Almost every value on its own in JavaScript evaluates to true, except what are known as the "falsy" values: false, 0, "" (an empty string), NaN, undefined, and null.

let x = 1;

let y = 2;

if (x = y) {

} else {

}

*In this example, the code block within the if statement will run for any value of y, unless y is falsy. The else block, which we expect to run here, will not actually run.*

**Basic Data Structures:**

An array's length, like the data types it can contain, is not fixed. Arrays can be defined with a length of any number of elements, and elements can be added or removed over time; in other words, arrays are mutable. In this challenge, we will look at two methods with which we can programmatically modify an array: **Array.push()** and **Array.unshift()**.

Both methods take one or more elements as parameters and add those elements to the array the method is being called on; the push() method adds elements to the end of an array, and unshift() adds elements to the beginning. Consider the following:

let twentyThree = 'XXIII';

let romanNumerals = ['XXI', 'XXII'];

romanNumerals.unshift('XIX', 'XX');

*romanNumerals would have the value ['XIX', 'XX', 'XXI', 'XXII'].*

romanNumerals.push(twentyThree);

*romanNumerals would have the value ['XIX', 'XX', 'XXI', 'XXII', 'XXIII']. Notice that we can also pass variables, which allows us even greater flexibility in dynamically modifying our array's data.*

Both push() and unshift() have corresponding methods that are nearly functional opposites: **pop()** and **shift()**. As you may have guessed by now, instead of adding, pop() removes an element from the end of an array, while shift() removes an element from the beginning. The key difference between pop() and shift() and their cousins push() and unshift(), is that neither method takes parameters, and each only allows an array to be modified by a single element at a time.Let's take a look:

let greetings = ['whats up?', 'hello', 'see ya!'];

greetings.pop();

*greetings would have the value ['whats up?', 'hello'].*

greetings.shift();

*greetings would have the value ['hello'].*

We can also return the value of the removed element with either method like this:

let popped = greetings.pop();

*greetings would have the value [], and popped would have the value hello.*

***Splice()***

Ok, so we've learned how to remove elements from the beginning and end of arrays using shift() and pop(), but what if we want to remove an element from somewhere in the middle? Or remove more than one element at once? Well, that's where **splice()** comes in. **splice()** allows us to do just that: **remove any number of consecutive elements from anywhere in an array.**

**splice()** can take up to 3 parameters, but for now, we'll focus on just the first 2. The first two parameters of splice() are integers which represent indexes, or positions, of items in the array that splice() is being called upon. And remember, arrays are zero-indexed, so to indicate the first element of an array, we would use 0. splice()'s first parameter represents the index on the array from which to begin removing elements, while the second parameter indicates the number of elements to delete. For example:

let array = ['today', 'was', 'not', 'so', 'great'];

array.splice(2, 2);

*Here we remove 2 elements, beginning with the third element (at index 2). array would have the value ['today', 'was', 'great'].*

splice() not only modifies the array it's being called on, but it also returns a new array containing the value of the removed elements:

let array = ['I', 'am', 'feeling', 'really', 'happy'];

let newArray = array.splice(3, 2);

*newArray has the value ['really', 'happy'].*

Remember in the last challenge we mentioned that splice() can take up to three parameters? Well, **you can use the third parameter, comprised of one or more element(s), to add to the array.** This can be incredibly useful for quickly switching out an element, or a set of elements, for another.

const numbers = [10, 11, 12, 12, 15];

const startIndex = 3;

const amountToDelete = 1;

numbers.splice(startIndex, amountToDelete, 13, 14);

console.log(numbers);

*The second occurrence of 12 is removed, and we add 13 and 14 at the same index. The numbers array would now be [ 10, 11, 12, 13, 14, 15 ].*

Here, we begin with an array of numbers. Then, we pass the following to splice(): The index at which to begin deleting elements (3), the number of elements to be deleted (1), and the remaining arguments (13, 14) will be inserted starting at that same index. Note that there can be any number of elements (separated by commas) following amountToDelete, each of which gets inserted.

***Slice()***

The next method we will cover **is slice()**. Rather than modifying an array, **slice() copies or extracts a given number of elements to a new array, leaving the array it is called upon untouched**. slice() takes only 2 parameters — the first is the index at which to begin extraction, and the second is the index at which to stop extraction (extraction will occur up to, but not including the element at this index). Consider this:

let weatherConditions = ['rain', 'snow', 'sleet', 'hail', 'clear'];

let todaysWeather = weatherConditions.slice(1, 3);

*todaysWeather would have the value ['snow', 'sleet'], while weatherConditions would still have ['rain', 'snow', 'sleet', 'hail', 'clear'].*

In effect, we have created a new array by extracting elements from an existing array.

While slice() allows us to be selective about what elements of an array to copy, among several other useful tasks, **ES6's new spread operator** allows us to easily copy all of an array's elements, in order, with a simple and highly readable syntax. The spread syntax simply looks like this: **...**

In practice, we can use the spread operator to copy an array like so:

let thisArray = [true, true, undefined, false, null];

let thatArray = [...thisArray];

*thatArray equals [true, true, undefined, false, null]. thisArray remains unchanged and thatArray contains the same elements as thisArray.*

Another huge advantage of the spread operator is the ability to **combine arrays, or to insert all the elements of one array into another, at any index.** With more traditional syntaxes, we can concatenate arrays, but this only allows us to combine arrays at the end of one, and at the start of another. Spread syntax makes the following operation extremely simple:

let thisArray = ['sage', 'rosemary', 'parsley', 'thyme'];

let thatArray = ['basil', 'cilantro', ...thisArray, 'coriander'];

*thatArray would have the value ['basil', 'cilantro', 'sage', 'rosemary', 'parsley', 'thyme', 'coriander'].*

Using spread syntax, we have just achieved an operation that would have been more complex and more verbose had we used traditional methods.

***indexOf()***

Since arrays can be changed, or mutated, at any time, there's no guarantee about where a particular piece of data will be on a given array, or if that element even still exists. Luckily, JavaScript provides us with another built-in method, **indexOf(),** that allows us to quickly and easily check for the presence of an element on an array. **indexOf()** takes an element as a parameter, and when called, **it returns the position, or index, of that element, or -1 if the element does not exist on the array.**

For example:

let fruits = ['apples', 'pears', 'oranges', 'peaches', 'pears'];

fruits.indexOf('dates');

fruits.indexOf('oranges');

fruits.indexOf('pears');

*indexOf('dates') returns -1, indexOf('oranges') returns 2, and indexOf('pears') returns 1 (the first index at which each element exists).*

Sometimes when working with arrays, it is very handy to be able to **iterate through each item** to find one or more elements that we might need, or to manipulate an array based on which data items meet a certain set of criteria. JavaScript offers several built in methods that each iterate over arrays in slightly different ways to achieve different results (such as every(), forEach(), map(), etc.), however the technique which is most flexible and offers us the greatest amount of control is a simple for loop.

Consider the following:

function greaterThanTen(arr) {

let newArr = [];

for (let i = 0; i < arr.length; i++) {

if (arr[i] > 10) {

newArr.push(arr[i]);

}

}

return newArr;

}

greaterThanTen([2, 12, 8, 14, 80, 0, 1]);

*Using a for loop, this function iterates through and accesses each element of the array, and subjects it to a simple test that we have created. In this way, we have easily and programmatically determined which data items are greater than 10, and returned a new array, [12, 14, 80], containing those items.*

One of the most powerful features when thinking of arrays as data structures, is **that arrays can contain, or even be completely made up of other arrays**. We have seen arrays that contain arrays in previous challenges, but fairly simple ones. However, arrays can contain an infinite depth of arrays that can contain other arrays, each with their own arbitrary levels of depth, and so on. In this way, an array can very quickly become a very complex data structure, known as a **multi-dimensiona**l, or **nested array**. Consider the following example:

let nestedArray = [

['deep'],

[

['deeper'], ['deeper']

],

[

[

['deepest'], ['deepest']

],

[

[

['deepest-est?']

]

]

]

];

*The deep array is nested 2 levels deep. The deeper arrays are 3 levels deep. The deepest arrays are 4 levels, and the deepest-est? is 5.*

While this example may seem convoluted, this level of complexity is not unheard of, or even unusual, when dealing with large amounts of data. However, we can still very easily access the deepest levels of an array this complex with bracket notation:

console.log(nestedArray[2][1][0][0][0]);

*This logs the string deepest-est?. And now that we know where that piece of data is, we can reset it if we need to:*

nestedArray[2][1][0][0][0] = 'deeper still';

console.log(nestedArray[2][1][0][0][0]);

*Now it logs deeper still.*

***Objects:***

At their most basic, objects are just collections of key-value pairs. In other words, they are pieces of data (values) mapped to unique identifiers called properties (keys). Take a look at an example:

const tekkenCharacter = {

player: 'Hwoarang',

fightingStyle: 'Tae Kwon Doe',

human: true

};

*The above code defines a Tekken video game character object called tekkenCharacter. It has three properties, each of which map to a specific value.*

If you want to add an additional property, such as "origin", it can be done by assigning origin to the object:

tekkenCharacter.origin = 'South Korea';

*This uses dot notation.* *If you were to observe the tekkenCharacter object, it will now include the origin property.*

Hwoarang also had distinct orange hair. You can add this property with bracket notation by doing:

tekkenCharacter['hair color'] = 'dyed orange';

Bracket notation is required if your property has a space in it or if you want to use a variable to name the property. In the above case, the property is enclosed in quotes to denote it as a string and will be added exactly as shown. Without quotes, it will be evaluated as a variable and the name of the property will be whatever value the variable is.

Here's an example with a variable:

const eyes = 'eye color';

tekkenCharacter[eyes] = 'brown';

*After adding all the examples, the object will look like this:*

{

player: 'Hwoarang',

fightingStyle: 'Tae Kwon Doe',

human: true,

origin: 'South Korea',

'hair color': 'dyed orange',

'eye color': 'brown'

};

Now let's take a look at a slightly more complex object. Object properties can be nested to an arbitrary depth, and their values can be any type of data supported by JavaScript, including arrays and even other objects. Consider the following:

let nestedObject = {

id: 28802695164,

date: 'December 31, 2016',

data: {

totalUsers: 99,

online: 80,

onlineStatus: {

active: 67,

away: 13,

busy: 8

}

}

};

*nestedObject has three properties: id (value is a number), date (value is a string), and data (value is an object with its nested structure).* While structures can quickly become complex, we can still use the same notations to access the information we need. To assign the value 10 to the busy property of the nested onlineStatus object, we use dot notation to reference the property:

nestedObject.data.onlineStatus.busy = 10;

In the first object challenge we mentioned the use of bracket notation as a way to access property values using the evaluation of a variable. For instance, imagine that our foods object is being used in a program for a supermarket cash register. We have some function that sets the selectedFood and we want to check our foods object for the presence of that food. This might look like:

let selectedFood = getCurrentFood(scannedItem);

let inventory = foods[selectedFood];

*This code will evaluate the value stored in the selectedFood variable and return the value of that key in the foods object, or undefined if it is not present.*

Bracket notation is very useful because sometimes object properties are not known before runtime or we need to access them in a more dynamic way.

Now you know what objects are and their basic features and advantages. In short, they are key-value stores which provide a flexible, intuitive way to structure data, and, they provide very fast lookup time. Throughout the rest of these challenges, we will describe several common operations you can perform on objects so you can become comfortable applying these useful data structures in your programs.

In earlier challenges, we have both added to and modified an object's key-value pairs. Here we will see how we can **remove a key-value pair from an object.**

Let's revisit our foods object example one last time. If we wanted to remove the apples key, we can remove it by using the **delete** keyword like this:

delete foods.apples;

Now we can add, modify, and remove keys from objects. But what if we just wanted to know **if an object has a specific property?** JavaScript provides us with two different ways to do this. One uses the **hasOwnProperty()** method and the other uses the **in** keyword. If we have an object users with a property of Alan, we could **check for its presence in either of the following ways:**

users.hasOwnProperty('Alan');

'Alan' in users;

*Both of these would return true.*

**Sometimes you need to iterate through all the keys within an object.** You can use a **for...in loop** to do this. The **for...in loop** looks like:

const refrigerator = {

'milk': 1,

'eggs': 12,

};

for (const food in refrigerator) {

console.log(food, refrigerator[food]);

}

*This code logs milk 1 and eggs 12, with each key-value pair on its own line.*

We defined the variable food in the loop head and this variable was set to each of the object's keys on each iteration, resulting in each food's name being printed to the console.

**NOTE:** Objects do not maintain an ordering to stored keys like arrays do; thus a key's position on an object, or the relative order in which it appears, is irrelevant when referencing or accessing that key.

We can **also generate an array which contains all the keys stored in an object** with the **Object.keys()** method. This method takes an object as the argument and returns an array of strings representing each property in the object. Again, there will be no specific order to the entries in the array.

**Object Oriented Programming:**

Objects can have a special type of property, called a **method.**

Methods are **properties that are functions**. This adds different behavior to an object. Here is the duck example with a method:

let duck = {

name: "Aflac",

numLegs: 2,

sayName: function() {return "The name of this duck is " + duck.name + ".";}

};

duck.sayName();

*The example adds the sayName method, which is a function that returns a sentence giving the name of the duck. Notice that the method accessed the name property in the return statement using duck.name.* The next challenge will cover another way to do this.

The last challenge introduced a method to the duck object. It used duck.name dot notation to access the value for the name property within the return statement:

sayName: function() {return "The name of this duck is " + duck.name + ".";}

While this is a valid way to access the object's property, there is a pitfall here. If the variable name changes, any code referencing the original name would need to be updated as well. In a short object definition, it isn't a problem, but if an object has many references to its properties there is a greater chance for error.

A way to avoid these issues is with the **this** keyword:

let duck = {

name: "Aflac",

numLegs: 2,

sayName: function() {return "The name of this duck is " + this.name + ".";}

};

***this*** *is a deep topic, and the above example is only one way to use it. In the current context, this refers to the object that the method is associated with: duck. If the object's name is changed to mallard, it is not necessary to find all the references to duck in the code. It makes the code reusable and easier to read.*

**Constructors are functions that create new objects.** They define properties and behaviors that will belong to the new object. Think of them as a blueprint for the creation of new objects.

Here is an example of a constructor:

function Bird() {

this.name = "Albert";

this.color = "blue";

this.numLegs = 2;

}

*This constructor defines a Bird object with properties name, color, and numLegs set to Albert, blue, and 2, respectively.*

**Constructors follow a few conventions:**

* Constructors are defined with a capitalized name to distinguish them from other functions that are not constructors.
* Constructors use the keyword this to set properties of the object they will create. Inside the constructor, this refers to the new object it will create.
* Constructors define properties and behaviors instead of returning a value as other functions might.

Here's the Bird constructor from the previous challenge:

function Bird() {

this.name = "Albert";

this.color = "blue";

this.numLegs = 2;

}

let blueBird = new Bird();

**NOTE:** **this** inside the constructor always refers to the object being created.

Notice that the **new** **operator** is used when calling a constructor. This tells JavaScript to create a new instance of Bird called blueBird. Without the **new operator**, this inside the constructor would not point to the newly created object, giving unexpected results. Now blueBird has all the properties defined inside the Bird constructor:

blueBird.name;

blueBird.color;

blueBird.numLegs;

Just like any other object, its properties can be accessed and modified:

blueBird.name = 'Elvira';

blueBird.name;

The Bird and Dog constructors from the last challenge worked well. However, notice that all Birds that are created with the Bird constructor are automatically named Albert, are blue in color, and have two legs. What if you want birds with different values for name and color? It's possible to change the properties of each bird manually but that would be a lot of work:

let swan = new Bird();

swan.name = "Carlos";

swan.color = "white";

Suppose you were writing a program to keep track of hundreds or even thousands of different birds in an aviary. It would take a lot of time to create all the birds, then change the properties to different values for every one. To more easily create different Bird objects, **you can design your** Bird **constructor to accept parameters:**

function Bird(name, color) {

this.name = name;

this.color = color;

this.numLegs = 2;

}

Then pass in the values as arguments to define each unique bird into the Bird constructor: let cardinal = new Bird("Bruce", "red"); This gives a new instance of Bird with name and color properties set to Bruce and red, respectively. The numLegs property is still set to 2. The cardinal has these properties:

cardinal.name

cardinal.color

cardinal.numLegs

**The constructor is more flexible. It's now possible to define the properties for each Bird at the time it is created, which is one way that JavaScript constructors are so useful.** They group objects together based on shared characteristics and behavior and define a blueprint that automates their creation.

Anytime a constructor function creates a new object, that object is said to be an instance of its constructor. JavaScript gives a convenient way to verify this with the **instanceof** operator. **instanceof** allows you to compare an object to a constructor, returning true or false based on whether or not that object was created with the constructor. Here's an example:

let Bird = function(name, color) {

this.name = name;

this.color = color;

this.numLegs = 2;

}

let crow = new Bird("Alexis", "black");

crow instanceof Bird;

*This instanceof method would return true.*

If an object is created without using a constructor, instanceof will verify that it is not an instance of that constructor:

let canary = {

name: "Mildred",

color: "Yellow",

numLegs: 2

};

canary instanceof Bird;

*This instanceof method would return false.*

In the following example, the Bird constructor defines two properties: name and numLegs:

function Bird(name) {

this.name = name;

this.numLegs = 2;

}

let duck = new Bird("Donald");

let canary = new Bird("Tweety");

name and numLegs are called **own properties, because they are defined directly on the instance object.** That means that duck and canary each has its own separate copy of these properties. In fact every instance of Bird will have its own copy of these properties. The following code adds all of the own properties of duck to the array ownProps:

let ownProps = [];

for (let property in duck) {

if(duck.hasOwnProperty(property)) {

ownProps.push(property);

}

}

console.log(ownProps);

*The console would display the value ["name", "numLegs"].*

Since numLegs will probably have the same value for all instances of Bird, you essentially have a duplicated variable numLegs inside each Bird instance.

This may not be an issue when there are only two instances, but imagine if there are millions of instances. That would be a lot of duplicated variables.

A better way is to use the **prototype** of Bird. Properties in the **prototype** are shared among ALL instances of Bird. Here's how to add numLegs to the Bird **prototype**:

Bird.prototype.numLegs = 2;

*Now all instances of Bird have the numLegs property.*

console.log(duck.numLegs);

console.log(canary.numLegs);

Since all instances automatically have the properties on the prototype, **think of a prototype as a "recipe" for creating objects.** Note that the prototype for duck and canary is part of the Bird constructor as Bird.prototype.

You have now seen two kinds of properties: **own properties** and **prototype properties**. Own properties are defined directly on the object instance itself. And prototype properties are defined on the prototype.

function Bird(name) {

this.name = name; *//own property*

}

Bird.prototype.numLegs = 2; *// prototype property*

let duck = new Bird("Donald");

**Here is how you add** duck's **own properties** **to the array ownProps** **and prototype properties to the array prototypeProps:**

let ownProps = [];

let prototypeProps = [];

for (let property in duck) {

if(duck.hasOwnProperty(property)) {

ownProps.push(property);

} else {

prototypeProps.push(property);

}

}

console.log(ownProps);

console.log(prototypeProps);

*console.log(ownProps) would display ["name"] in the console, and console.log(prototypeProps) would display ["numLegs"].*

There is a special **constructor** property located on the object instances duck and beagle that were created in the previous challenges:

let duck = new Bird();

let beagle = new Dog();

console.log(duck.constructor === Bird);

console.log(beagle.constructor === Dog);

*Both of these console.log calls would display true in the console.*

Note that the constructor property is a reference to the constructor function that created the instance. The advantage of the constructor property is that it's possible to check for this property to find out what kind of object it is. Here's an example of how this could be used:

function joinBirdFraternity(candidate) {

if (candidate.constructor === Bird) {

return true;

} else {

return false;

}

}

**Note:** Since the constructor property can be overwritten (which will be covered in the next two challenges) **it’s generally better to use the *instanceof* method** to check the type of an object.

Up until now you have been adding properties to the prototype individually:

Bird.prototype.numLegs = 2;

*This becomes tedious after more than a few properties.*

Bird.prototype.eat = function() {

console.log("nom nom nom");

}

Bird.prototype.describe = function() {

console.log("My name is " + this.name);

}

A more efficient way is to **set the prototype to a new object that already contains the properties.**

This way, **the properties are added all at once:**

Bird.prototype = {

numLegs: 2,

eat: function() {

console.log("nom nom nom");

},

describe: function() {

console.log("My name is " + this.name);

}

};

There is one crucial side effect of manually setting the prototype to a new object. **It erases the constructor property!** This property can be used to check which constructor function created the instance, but since the property has been overwritten, it now gives false results:

duck.constructor === Bird;

duck.constructor === Object;

duck instanceof Bird;

*In order, these expressions would evaluate to false, true, and true.*

To fix this, whenever a prototype is manually set to a new object, **remember to define the constructor property:**

Bird.prototype = {

constructor: Bird,

numLegs: 2,

eat: function() {

console.log("nom nom nom");

},

describe: function() {

console.log("My name is " + this.name);

}

};

Just like people inherit genes from their parents, **an object inherits its prototype directly from the constructor function that created it.** For example, here the Bird constructor creates the duck object:

function Bird(name) {

this.name = name;

}

let duck = new Bird("Donald");

duck inherits its prototype from the Bird constructor function. You can show this relationship with the isPrototypeOf method:

Bird.prototype.isPrototypeOf(duck);

*This would return true.*

**All objects in JavaScript (with a few exceptions) have a prototype. Also, an object’s prototype itself is an object.**

function Bird(name) {

this.name = name;

}

typeof Bird.prototype;

Because a prototype is an object, a prototype can have its own prototype! In this case, the prototype of Bird.prototype is Object.prototype:

Object.prototype.isPrototypeOf(Bird.prototype);

How is this useful? You may recall the hasOwnProperty method from a previous challenge:

let duck = new Bird("Donald");

duck.hasOwnProperty("name");

The hasOwnProperty method is defined in Object.prototype, which can be accessed by Bird.prototype, which can then be accessed by duck. This is an example of the prototype chain. In this prototype chain, Bird is the supertype for duck, while duck is the subtype. Object is a supertype for both Bird and duck. Object is a supertype for all objects in JavaScript. **Therefore, any object can use the hasOwnProperty method.**

There's a principle in programming called **Don't Repeat Yourself (DRY).** The reason repeated code is a problem is because any change requires fixing code in multiple places. This usually means more work for programmers and more room for errors.

Notice in the example below that the describe method is shared by Bird and Dog:

Bird.prototype = {

constructor: Bird,

describe: function() {

console.log("My name is " + this.name);

}

};

Dog.prototype = {

constructor: Dog,

describe: function() {

console.log("My name is " + this.name);

}

};

The describe method is repeated in two places. The code can be edited to follow the DRY principle by creating a supertype (or parent) called Animal:

function Animal() { };

Animal.prototype = {

constructor: Animal,

describe: function() {

console.log("My name is " + this.name);

}

};

*Since Animal includes the describe method, you can remove it from Bird and Dog:*

Bird.prototype = {

constructor: Bird

};

Dog.prototype = {

constructor: Dog

};

In the previous challenge, you created a supertype called Animal that defined behaviors shared by all animals:

function Animal() { }

Animal.prototype.eat = function() {

console.log("nom nom nom");

};

This and the next challenge will cover **how to reuse the methods of Animal inside Bird and Dog without defining them again.** It uses a technique called inheritance. This challenge covers the first step: make an instance of the supertype (or parent). You already know one way to create an instance of Animal using the new operator:

let animal = new Animal();

There are some disadvantages when using this syntax for inheritance, which are too complex for the scope of this challenge. Instead, here's an alternative approach without those disadvantages:

let animal = Object.create(Animal.prototype); *// this line is better*

Object.create(obj) creates a new object, and sets obj as the new object's prototype. Recall that the prototype is like the "recipe" for creating an object. By setting the prototype of animal to be the prototype of Animal, you are effectively giving the animal instance the same "recipe" as any other instance of Animal.

animal.eat();

animal instanceof Animal;

*The instanceof method here would return true.*

In the previous challenge you saw the first step for inheriting behavior from the supertype (or parent) Animal: making a new instance of Animal.

This challenge covers **the next step: set the prototype of the subtype (or child)—in this case, Bird—to be an instance of Animal.**

Bird.prototype = Object.create(Animal.prototype);

Remember that the prototype is like the "recipe" for creating an object. In a way, the recipe for Bird now includes all the key "ingredients" from Animal.

let duck = new Bird("Donald");

duck.eat();

*duck inherits all of Animal's properties, including the eat method.*

**When an object inherits its prototype from another object, it also inherits the supertype's constructor property.** Here's an example:

function Bird() { }

Bird.prototype = Object.create(Animal.prototype);

let duck = new Bird();

duck.constructor

But duck and all instances of Bird should show that they were constructed by Bird and not Animal. To do so, **you can manually set the constructor property of Bird to the Bird object:**

Bird.prototype.constructor = Bird;

duck.constructor

**A constructor function that inherits its prototype object from a supertype constructor function can still have its own methods in addition to inherited methods.** For example, Bird is a constructor that inherits its prototype from Animal:

function Animal() { }

Animal.prototype.eat = function() {

console.log("nom nom nom");

};

function Bird() { }

Bird.prototype = Object.create(Animal.prototype);

Bird.prototype.constructor = Bird;

In addition to what is inherited from Animal, you want to add behavior that is unique to Bird objects. Here, Bird will get a fly() function. Functions are added to Bird's prototype the same way as any constructor function:

Bird.prototype.fly = function() {

console.log("I'm flying!");

};

*Now instances of Bird will have both eat() and fly() methods:*

let duck = new Bird();

duck.eat();

duck.fly();

*duck.eat() would display the string nom nom nom in the console, and duck.fly() would display the string I'm flying!.*

**In previous lessons, you learned that an object can inherit its behavior (methods) from another object by referencing its prototype object:**

ChildObject.prototype = Object.create(ParentObject.prototype);

Then the ChildObject received its own methods by chaining them onto its prototype:

ChildObject.prototype.methodName = function() {...};

It's possible to override an inherited method. It's done the same way - by adding a method to ChildObject.prototype using the same method name as the one to override. Here's an example of Bird overriding the eat() method inherited from Animal:

function Animal() { }

Animal.prototype.eat = function() {

return "nom nom nom";

};

function Bird() { }

Bird.prototype = Object.create(Animal.prototype);

Bird.prototype.eat = function() {

return "peck peck peck";

};

If you have an instance let duck = new Bird(); and you call duck.eat(), this is how JavaScript looks for the method on the prototype chain of duck:

*duck => Is eat() defined here? No.*

*Bird => Is eat() defined here? => Yes. Execute it and stop searching.*

*Animal => eat() is also defined, but JavaScript stopped searching before reaching this level.*

*Object => JavaScript stopped searching before reaching this level.*

As you have seen, behavior is shared through inheritance. **However, there are cases when inheritance is not the best solution.** Inheritance does not work well for unrelated objects like Bird and Airplane. They can both fly, but a Bird is not a type of Airplane and vice versa.

**For unrelated objects, it's better to use mixins**. A **mixin** allows other objects to use a collection of functions.

let flyMixin = function(obj) {

obj.fly = function() {

console.log("Flying, wooosh!");

}

};

*The flyMixin takes any object and gives it the fly method.*

let bird = {

name: "Donald",

numLegs: 2

};

let plane = {

model: "777",

numPassengers: 524

};

flyMixin(bird);

flyMixin(plane);

*Here bird and plane are passed into flyMixin, which then assigns the fly function to each object. Now bird and plane can both fly:*

bird.fly();

plane.fly();

*The console would display the string Flying, wooosh! twice, once for each .fly() call.*

**Note how the mixin allows for the same fly method to be reused by unrelated objects bird and plane.**

In the previous challenge, bird had a public property name. It is considered public because it can be accessed and changed outside of bird's definition.

bird.name = "Duffy";

Therefore, any part of your code can easily change the name of bird to any value. Think about things like passwords and bank accounts being easily changeable by any part of your codebase. That could cause a lot of issues.

**The simplest way to make this public property private is by creating a variable within the constructor function.** This changes the scope of that variable to be within the constructor function versus available globally. This way, the variable can only be accessed and changed by methods also within the constructor function.

function Bird() {

let hatchedEgg = 10;

this.getHatchedEggCount = function() {

return hatchedEgg;

};

}

let ducky = new Bird();

ducky.getHatchedEggCount();

Here getHatchedEggCount is a privileged method, because it has access to the private variable hatchedEgg. This is possible because hatchedEgg is declared in the same context as getHatchedEggCount. In JavaScript, a function always has access to the context in which it was created. This is called **closure.**

A common pattern in JavaScript is to execute a function as soon as it is declared:

(function () {

console.log("Chirp, chirp!");

})();

*This is an anonymous function expression that executes right away, and outputs Chirp, chirp! immediately.*

**Note** that the function has no name and is not stored in a variable. The two parentheses () at the end of the function expression cause it to be immediately executed or invoked. This pattern is known as an **immediately invoked function expression** or **IIFE**.

An immediately invoked function expression **(IIFE)** is often used to group related functionality into a single object or module. For example, an earlier challenge defined two mixins:

function glideMixin(obj) {

obj.glide = function() {

console.log("Gliding on the water");

};

}

function flyMixin(obj) {

obj.fly = function() {

console.log("Flying, wooosh!");

};

}

**We can group these mixins into a module as follows:**

let motionModule = (function () {

return {

glideMixin: function(obj) {

obj.glide = function() {

console.log("Gliding on the water");

};

},

flyMixin: function(obj) {

obj.fly = function() {

console.log("Flying, wooosh!");

};

}

}

})();

Note that you have an immediately invoked function expression **(IIFE)** that returns an object motionModule. This returned object contains all of the mixin behaviors as properties of the object. The advantage of the module pattern is that all of the motion behaviors can be packaged into a single object that can then be used by other parts of your code. Here is an example using it:

motionModule.glideMixin(duck);

duck.glide();

**Functional Programming:**

Functional programming is a style of programming **where solutions are simple, isolated functions**, without any side effects outside of the function scope: **INPUT -> PROCESS -> OUTPUT**

Functional programming is about:

* Isolated functions - there is no dependence on the state of the program, which includes global variables that are subject to change
* Pure functions - the same input always gives the same output
* Functions with limited side effects - any changes, or mutations, to the state of the program outside the function are carefully controlled

The members of freeCodeCamp happen to love tea.

In the code editor, the prepareTea and getTea functions are already defined for you. Call the getTea function to get 40 cups of tea for the team, and store them in the tea4TeamFCC variable.

// Function that returns a string representing a cup of green tea

const prepareTea = () => 'greenTea';

/\*

Given a function (representing the tea type) and number of cups needed, the

following function returns an array of strings (each representing a cup of

a specific type of tea).

\*/

const getTea = (numOfCups) => {

const teaCups = [];

for(let cups = 1; cups <= numOfCups; cups += 1) {

const teaCup = prepareTea();

teaCups.push(teaCup);

}

return teaCups;

};

// Only change code below this line

const tea4TeamFCC = getTea(40);

// Only change code above this line

The FCC Team now wants two types of tea: green tea and black tea. With that information, we'll need to revisit the getTea function from last challenge to handle various tea requests. We can modify getTea to accept a function as a parameter to be able to change the type of tea it prepares. This makes getTea more flexible, and gives the programmer more control when client requests change. But first, let's cover some functional terminology:

**Callbacks** are the functions that are slipped or passed into another function to decide the invocation of that function. You may have seen them passed to other methods, for example in filter, the callback function tells JavaScript the criteria for how to filter an array.

Functions that can be assigned to a variable, passed into another function, or returned from another function just like any other normal value, are called **first class functions. In JavaScript, all functions are first class functions.**

The functions that take a function as an argument, or return a function as a return value, are called **higher order functions.**

When functions are passed in to or returned from another function, then those functions which were passed in or returned can be called a **lambda.**

// Function that returns a string representing a cup of green tea

const prepareGreenTea = () => 'greenTea';

// Function that returns a string representing a cup of black tea

const prepareBlackTea = () => 'blackTea';

/\*

Given a function (representing the tea type) and number of cups needed, the

following function returns an array of strings (each representing a cup of

a specific type of tea).

\*/

const getTea = (prepareTea, numOfCups) => {

const teaCups = [];

for(let cups = 1; cups <= numOfCups; cups += 1) {

const teaCup = prepareTea();

teaCups.push(teaCup);

}

return teaCups;

};

// Only change code below this line

const tea4GreenTeamFCC = getTea(prepareGreenTea, 27);

const tea4BlackTeamFCC = getTea(prepareBlackTea, 13);

// Only change code above this line

console.log(tea4GreenTeamFCC, tea4BlackTeamFCC);

Functional programming is a good habit. It keeps your code easy to manage, and saves you from sneaky bugs. But before we get there, let's look at an imperative approach to programming to highlight where you may have issues.

In English (and many other languages), the imperative tense is used to give commands. Similarly, an imperative style in programming is one that gives the computer a set of statements to perform a task.

Often the statements change the state of the program, like updating global variables. A classic example is writing a for loop that gives exact directions to iterate over the indices of an array.

In contrast, functional programming is a form of declarative programming. You tell the computer what you want done by calling a method or function.

JavaScript offers many predefined methods that handle common tasks so you don't need to write out how the computer should perform them. For example, instead of using the for loop mentioned above, you could call the map method which handles the details of iterating over an array. This helps to avoid semantic errors, like the "Off By One Errors" that were covered in the Debugging section.

Consider the scenario: you are browsing the web in your browser, and want to track the tabs you have opened. Let's try to model this using some simple object-oriented code.

A Window object is made up of tabs, and you usually have more than one Window open. The titles of each open site in each Window object is held in an array. After working in the browser (opening new tabs, merging windows, and closing tabs), you want to print the tabs that are still open. Closed tabs are removed from the array and new tabs (for simplicity) get added to the end of it.

The code editor shows an implementation of this functionality with functions for tabOpen(), tabClose(), and join(). The array tabs is part of the Window object that stores the name of the open pages.

// tabs is an array of titles of each site open within the window

const Window = function(tabs) {

this.tabs = tabs; // We keep a record of the array inside the object

};

// When you join two windows into one window

Window.prototype.join = function(otherWindow) {

this.tabs = this.tabs.concat(otherWindow.tabs);

return this;

};

// When you open a new tab at the end

Window.prototype.tabOpen = function(tab) {

this.tabs.push('new tab'); // Let's open a new tab for now

return this;

};

// When you close a tab

Window.prototype.tabClose = function(index) {

// Only change code below this line

const tabsBeforeIndex = this.tabs.slice(0, index); // Get the tabs before the tab

const tabsAfterIndex = this.tabs.slice(index + 1); // Get the tabs after the tab

this.tabs = tabsBeforeIndex.concat(tabsAfterIndex); // Join them together

// Only change code above this line

return this;

};

// Let's create three browser windows

const workWindow = new Window(['GMail', 'Inbox', 'Work mail', 'Docs', 'freeCodeCamp']); // Your mailbox, drive, and other work sites

const socialWindow = new Window(['FB', 'Gitter', 'Reddit', 'Twitter', 'Medium']); // Social sites

const videoWindow = new Window(['Netflix', 'YouTube', 'Vimeo', 'Vine']); // Entertainment sites

// Now perform the tab opening, closing, and other operations

const finalTabs = socialWindow

.tabOpen() // Open a new tab for cat memes

.join(videoWindow.tabClose(2)) // Close third tab in video window, and join

.join(workWindow.tabClose(1).tabOpen());

console.log(finalTabs.tabs);

If you haven't already figured it out, the issue in the previous challenge was with the splice call in the tabClose() function. Unfortunately, splice changes the original array it is called on, so the second call to it used a modified array, and gave unexpected results.

This is a small example of a much larger pattern - you call a function on a variable, array, or an object, and the function changes the variable or something in the object.

**One of the core principles of functional programming is to not change things.** Changes lead to bugs. It's easier to prevent bugs knowing that your functions don't change anything, including the function arguments or any global variable.

The previous example didn't have any complicated operations but the splice method changed the original array, and resulted in a bug.

**Recall that in functional programming, changing or altering things is called mutation, and the outcome is called a side effect.** **A function, ideally, should be a pure function, meaning that it does not cause any side effects.**

// The global variable

let fixedValue = 4;

// Only change code below this line

function incrementer) {

return fixedValue + 1;

// Only change code above this line

}

The last challenge was a step closer to functional programming principles, but there is still something missing.

We didn't alter the global variable value, but the function incrementer would not work without the global variable fixedValue being there.

**Another principle of functional programming is to always declare your dependencies explicitly.** **This means if a function depends on a variable or object being present, then pass that variable or object directly into the function as an argument.**

There are several good consequences from this principle. The function is easier to test, you know exactly what input it takes, and it won't depend on anything else in your program.

This can give you more confidence when you alter, remove, or add new code. You would know what you can or cannot change and you can see where the potential traps are.

Finally, the function would always produce the same output for the same set of inputs, no matter what part of the code executes it.

// The global variable

let fixedValue = 4;

// Only change code below this line

function incrementer(valueToIncrement) {

return valueToIncrement += 1;

// Only change code above this line

}

So far, we have seen two distinct principles for functional programming:

1. Don't alter a variable or object - create new variables and objects and return them if need be from a function. Hint: using something like const newArr = arrVar, where arrVar is an array will simply create a reference to the existing variable and not a copy. So changing a value in newArr would change the value in arrVar.
2. Declare function parameters - any computation inside a function depends only on the arguments passed to the function, and not on any global object or variable.

Adding one to a number is not very exciting, but we can apply these principles when working with arrays or more complex objects.

Rewrite the code so the global array bookList is not changed inside either function. The add function should add the given bookName to the end of the array passed to it and return a new array (list). The remove function should remove the given bookName from the array passed to it.

// The global variable

const bookList = ["The Hound of the Baskervilles", "On The Electrodynamics of Moving Bodies", "Philosophiæ Naturalis Principia Mathematica", "Disquisitiones Arithmeticae"];

// Change code below this line

function add(listName, bookName) {

const list = [...listName];

list.push(bookName);

return list;

// Change code above this line

}

// Change code below this line

function remove(listName, bookName) {

const list = [...listName];

const book\_index = list.indexOf(bookName);

if (book\_index >= 0) {

list.splice(book\_index, 1);

return list;

// Change code above this line

}

}

So far we have learned to use pure functions to avoid side effects in a program. Also, we have seen the value in having a function only depend on its input arguments.

This is only the beginning. As its name suggests, functional programming is centered around a theory of functions.

It would make sense to be able to pass them as arguments to other functions, and return a function from another function. Functions are considered first class objects in JavaScript, which means they can be used like any other object. They can be saved in variables, stored in an object, or passed as function arguments.

Let's start with some simple array functions, which are methods on the array object prototype. In this exercise we are looking at **Array.prototype.map()**, or more simply **map**.

The **map** method iterates over each item in an array and returns a new array containing the results of calling the callback function on each element. It does this without mutating the original array.

When the callback is used, it is passed three arguments. The first argument is the current element being processed. The second is the index of that element and the third is the array upon which the **map** method was called.

See below for an example using the **map** method on the users array to return a new array containing only the names of the users as elements. For simplicity, the example only uses the first argument of the callback.

const users = [

{ name: 'John', age: 34 },

{ name: 'Amy', age: 20 },

{ name: 'camperCat', age: 10 }

];

const names = users.map(user => user.name);

console.log(names);

*The console would display the value [ 'John', 'Amy', 'camperCat' ].*

As you have seen from applying Array.prototype.map(), or simply map() earlier, the map method returns an array of the same length as the one it was called on. It also doesn't alter the original array, as long as its callback function doesn't.

In other words, map is a pure function, and its output depends solely on its inputs. Plus, it takes another function as its argument.

You might learn a lot about the map method if you implement your own version of it. It is recommended you use a for loop or Array.prototype.forEach().

Write your own Array.prototype.myMap(), which should behave exactly like Array.prototype.map(). You should not use the built-in map method. The Array instance can be accessed in the myMap method using this.

Array.prototype.myMap = function(callback) {

const newArray = [];

// Only change code below this line

for (let i = 0; i < this.length; i++) {

/\*

this[i] - current element being processed

i - index of that element

this - array upon which the map method was called

\*/

newArray.push(callback(this[i], i, this));

}

// Only change code above this line

return newArray;

};

Another useful array function is Array.prototype.filter(), or simply **filter().**

**filter** calls a function on each element of an array and returns a new array containing only the elements for which that function returns a truthy value - that is, a value which returns true if passed to the Boolean() constructor. In other words, it filters the array, based on the function passed to it. Like map, it does this without needing to modify the original array.

The callback function accepts three arguments. The first argument is the current element being processed. The second is the index of that element and the third is the array upon which the filter method was called.

See below for an example using the filter method on the users array to return a new array containing only the users under the age of 30. For simplicity, the example only uses the first argument of the callback.

const users = [

{ name: 'John', age: 34 },

{ name: 'Amy', age: 20 },

{ name: 'camperCat', age: 10 }

];

const usersUnder30 = users.filter(user => user.age < 30);

console.log(usersUnder30);

*The console would display the value [ { name: 'Amy', age: 20 }, { name: 'camperCat', age: 10 } ].*

You might learn a lot about the filter method if you implement your own version of it. It is recommended you use a for loop or Array.prototype.forEach().

Write your own Array.prototype.myFilter(), which should behave exactly like Array.prototype.filter(). You should not use the built-in filter method. The Array instance can be accessed in the myFilter method using this.

Array.prototype.myFilter = function(callback) {

const newArray = [];

// Only change code below this line

for (let i = 0; i < this.length; i++) {

if (callback(this[i], i, this) == true) {

newArray.push(this[i]);

}

}

// Only change code above this line

return newArray;

};

The **slice** method returns a copy of certain elements of an array. It can take two arguments, the first gives the index of where to begin the slice, the second is the index for where to end the slice (and it's non-inclusive). If the arguments are not provided, the default is to start at the beginning of the array through the end, which is an easy way to make a copy of the entire array. The slice method does not mutate the original array, but returns a new one.

Here's an example:

const arr = ["Cat", "Dog", "Tiger", "Zebra"];

const newArray = arr.slice(1, 3);

*newArray would have the value ["Dog", "Tiger"].*

Use the slice method in the sliceArray function to return part of the anim array given the provided beginSlice and endSlice indices. The function should return an array.

function sliceArray(anim, beginSlice, endSlice) {

// Only change code below this line

return anim.slice(beginSlice, endSlice);

// Only change code above this line

}

const inputAnim = ["Cat", "Dog", "Tiger", "Zebra", "Ant"];

sliceArray(inputAnim, 1, 3);

A common pattern while working with arrays is when you want to remove items and keep the rest of the array. JavaScript offers the **splice** method for this, which takes arguments for the index of where to start removing items, then the number of items to remove. If the second argument is not provided, the default is to remove items through the end. However, the splice method mutates the original array it is called on. Here's an example:

const cities = ["Chicago", "Delhi", "Islamabad", "London", "Berlin"];

cities.splice(3, 1);

*Here splice returns the string London and deletes it from the cities array. cities will have the value ["Chicago", "Delhi", "Islamabad", "Berlin"].*

As we saw in the last challenge, the slice method does not mutate the original array, but returns a new one which can be saved into a variable. Recall that the **slice** method takes two arguments for the indices to begin and end the slice (the end is non-inclusive), and returns those items in a new array. Using the **slice** method instead of splice helps to avoid any array-mutating side effects.

Rewrite the function nonMutatingSplice by using slice instead of splice. It should limit the provided cities array to a length of 3, and return a new array with only the first three items.

Do not mutate the original array provided to the function.

function nonMutatingSplice(cities) {

return cities.slice(0,3);

}

**Concatenation means to join items end to end.** JavaScript offers the **concat** method for both strings and arrays that work in the same way. For arrays, the method is called on one, then another array is provided as the argument to **concat**, which is added to the end of the first array. It returns a new array and does not mutate either of the original arrays:

[1, 2, 3].concat([4, 5, 6]);

*The returned array would be [1, 2, 3, 4, 5, 6].*

Functional programming is all about creating and using non-mutating functions.

The last challenge introduced the **concat** method as a way to merge arrays into a new array without mutating the original arrays. Compare concat to the push method. push adds items to the end of the same array it is called on, which mutates that array. Here's an example:

const arr = [1, 2, 3];

arr.push(4, 5, 6);

*arr would have a modified value of [1, 2, 3, 4, 5, 6], which is not the functional programming way.*

**concat** offers a way to merge new items to the end of an array without any mutating side effects.