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;

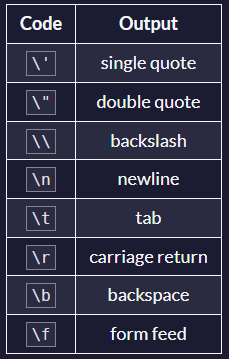
**Descrement (zmniejszenie o 1):**

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.

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);