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;

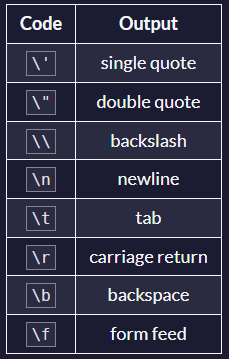
**Descrement (zmniejszenie o 1):**

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].