|  |
| --- |
| INTRODUCTION TO PROGRAMMING WITH JAVASCRIPT |

|  |  |
| --- | --- |
| INTRODUCTION TO JAVASCRIPT | 2 |
| JS DEVELOPMENT SETUP | 2 |
| YOUR FIRST JS PROGRAM | 2 |
| DATA TYPES | 3 |
| COMMENTS | 4 |
| VARIABLES | 4 |
| ARITHMETIC OPERATORS | 7 |
| ASSIGNMENTS OPERATORS | 8 |
| TYPE CONVERSIONS | 9 |
| COMPARISONS OPERATORS | 12 |
| LOGICAL OPERATORS | 15 |
| CONDITIONALS | 18 |
| NESTED CONDITIONALS | 20 |
| SWITCH STATEMENTS | 24 |
| ARRAYS | 28 |
| ARRAY METHODS | 35 |
| STRINGS | 39 |
| STRING METHODS | 41 |
| FOR LOOPS | 43 |
| FOR OF | 45 |
| WHILE LOOPS | 49 |
| SETS | 51 |
| MAPS | 55 |
| OBJECTS | 58 |
| ERROR HANDLING | 67 |
| FUNCTIONS | 74 |
| EVENT & THIS | 81 |
| ARROW FUNCTIONS | 85 |
| MAP, FILTER, REDUCE | 90 |
| SCOPE | 97 |
| MUTABILITY | 105 |
| MATH | 109 |
| RANDOM | 111 |
| SORTING | 116 |

|  |
| --- |
| INTRODUCTION TO PROGRAMMING WITH JAVASCRIPT |

**INTRODUCTION TO JAVASCRIPT**

Document object model (DOM).

The tree allows us to retrieve specific elements from the html document.

**JAVASCRIPT SET UP ENVIRONMENT**

Runtime environment that allows us to run JavaScript directly from the terminal, so without the need to use a browser. JavaScript Development is the directory where we’re going to put all the code files.

To create a JavaScript code, we go to create a file.

If we want to run a test of the file, we can console.log, then write node test.js in the terminal.

Another way to do a test run is to use the browser. We start by creating a html file, we then write a JavaScript tag (<script></script>) inside the body tag and copy paste the console log code.

After this, we click on GO LIVE. The webpage will open but will stay blank. The reason for that is that any log will appear on the developer tools of the browser. We right click on the page and click inspect. Then, in the elements section we can see the code.

**YOUR FIRST JAVASCRIPT PROGRAM**

To create an easier code, we will typically create a separate JavaScript document from the html one, and copy paste the link inside the html document, like we do with CSS. To do so, we will use the tag <script></script>

<script src=”test.js”></script>

A log is information that we provide to the console. Typically, we use this as a developer to debug or test a code. To run tests with the console.log, we will use a string. A string in JavaScript is anything that’s surrounded by simple or double quotation marks. The semicolon terminates a line.

The 3 main actions we can take with JavaScript is to write general info’s. Create a warning and an error. The general info’s are going to be classified under .log, the warning under .warn, and the error under .error.

If we want to interact with elements within the html document, we can write an id reference inside the html document.

<p id=”name”>my name is Tim</p>

<script src=”test.js”></script>

We then go to the JavaScript document and type this exact thing.

Document.getElementById (“name”)

The term document stands for Document Object Model (DOM). GetElementById is what we call a method or a function that’s associated with the document. This sentence is pretty much telling the program to get into the document, to find the element that goes by the ID “Name” and do the alterations we ask it to do. There are multiple ways to alter an element. One of them is this one.

Document.getElementById (“name”).innerHTML= “ “

By adding innerHTML=””, if we have the space inside the string empty, that will mean that the document is void. What’s inside the string will be the element appearing on the page. If we add a sentence inside, we can see that the sentence now appears on the page.

Document.getElementById (“name”).innerHTML= “hello my name is Joe “

This is the basis for how we go and modify some elements inside the html document, using JavaScript.

Beyond the innerHTML, we can also add properties to our tag. Example

Document.getElementById (“name”).style = “color:red”;

Besides modifying documents, we can also read them. To do so, we will write console.log in front of our document and surround it with brackets. Example

Console.log (document.getElementById (“name).innerHTML);

**DATA TYPES**

The type of Data we have will change the way we interact with it.

Different types of Data

String: Any character inside quotations marks. Simple or double quotation.

Number: Any number we have. Can be positive, negative, decimal etc

Boolean: one of two values. true and false, written both in lower case and exactly like this. True is equivalent to 1 and false is equivalent to 0. That’s why we have 1 and 0 in binary value.

Undefined: We don’t have something. Something doesn’t exist. Used when we want to look for something, but it doesn’t exist.

Null: similar to undefined but used when we want to explicitly state that something is empty or doesn’t have a value.

Object: Anything that is inside of curly braces.

NaN: States for Not A Number. When we try to do some kind of computation/operation where the result is not a valid JavaScript number. Example, trying to divide something by 0. The result will be NaN.

If we want to know what kind of value something is, we would type this code and write the element after typeof. Here we ask what 1 is.

console.log(typeof 1);

**COMMENTS**

To write a single line comment, we can start by typing //. It’s good practice to add a space right those and before the actual content. To write a multiple line comment, we can use the /\* then \*/. Anything in between will be the content of the comment.

If we are writing a code and want to have the console.log written in it, but don’t want it printed out if somebody else is reading it, we can change the console into a comment by adding // right in front of it.

To do the same for several console.log, we have a shortcut. CTRL+/ will automatically add // in front of the selected group and change it into a comment.

Another way is to add a /\* \*/ a create a multi line comment to the double // won’t be in front of every line. That will make it cleaner.

**VARIABLES**

Similar principle as variables in mathematics where x has a specific value. Container that is storing some type of value. The variable will have a name and that name will be associated with a specific value.

Symbol in coding is var.

*var* x = 1 ;

That statement means we have declared a variable. The variable is x and the value of the variable is 1.

x is storing the value 1. The container is inside the computer memory, also known as RAM. Once we have declared a variable, we can log it in, writing.

*var* x = 1 ;

console.log (x);

Different ways to define/declare variables:

Typically, we want to use a descriptive name for the variable.

Camel case: Since it’s two names, instead of adding a space in between the words, we’re going to use the camel case, meaning that we start the variable name with a lower case and every new word will start with a capital letter. Example:

*var* myNumber = 1;

Snake case: Not the convention in JavaScript. We use an underscore.

my\_number

When naming a variable, we want to be very specific. No special characters, no space between words. The only special characters working would be underscore and numbers. We can’t start a variable name by a number. Numbers need to be at the end.

Why is it important to capitalize the letters? The variable mynumber and myNumber are different and will have different values.

Use of multiple variables.

We can change the value of variables. They can change during the execution of the program.

*var* myNumber = 1;

console.log (myNumber);

myNumber = 2;

console.log (myNumber);

We can also use a variable to declare another variable.

We can print multiple variables within the same log. We just have to separate them with a comma.

*var* name1 = “tim”;

*var* name2 = “bob”;

console.log(name1, name2);

We can assign one variable to another variable.

name2 = name1;

console.log(name1, name2);

The result will be Tim for both.

**Different ways to define variables**

*var*: Var is the typical way to do it.

*constant*: Sometimes we have a constant variable or constant value that we don’t want to change. We’ll use *const* myNumber = 22;

*let:* We use this to specify that the variable can be changed.

*let* myNumber = 22;

myNumber = 3;

Best to use either *let* or *const*, instead of *var*.

When we create a variable that uses *let*, we don’t need to assign value to it.

Example:

*let* myNumber;

If we don’t assign a value to it and log it in, the result will say undefined. If we assign the value null, the result will be null.

**ARITHMETIC OPERATORS**

Used to perform computations and mathematical operations in JavaScript.

|  |  |
| --- | --- |
| Addition + | Subtraction - |
| Multiplication \* | Division / |
| Exponential \*\* | Modulus % |
| Increment ++ it adds 1 to whatever value x is | Decrement - -  It subtracts 1 to whatever value x is |

Examples of operations.

*const* x = 10;

*const* y = 20;

*const* z = x + y;

console.log (z);

x+y is what we call an expression. An expression will evaluate to a value.

Order of operations

BEDMAS: Braces, exponents, division, multiplication, addition, subtraction.

Modulus: The modulus provides the remainder after performing a division. If we have 10 % 4 the answer will be 2 because it’s the amount of times 4 fits into 10. It fits twice and equals 8. So 10-8=2.

*const* x = 10;

*const* y = 4;

*const* z = x % y;

console.log (z);

If we were to divide y by 0, we would have infinity as a result.

*const* x = 10;

*const* y = 4;

*const* z = y % 0;

console.log (z);

**ASSIGNMENTS OPERATORS**

The most basic one is =

It allows us to assign a value to x. The other operations all have = involved in them.

+= adds a value to whatever x is

*let* num = 10;

num +=5

console.log(num);

If the number is 10 and we write a variable like above, we tell the program that we want to add 5 to the value of the initial number. 5+10 = 15. This is a shortcut version of writing num = num + 5

*let* num = 10;

num -=6

console.log(num);

This would equal 4 because it’s a shortcut for num = num - 6

|  |  |
| --- | --- |
| += | -= |
| \*= | /= |
| %= | \*\*= |

**TYPE CONVERSIONS**

Arithmetic and assignment operators are useful when we use the same types. When we’re using different values, we’ll have to convert them.

*const* x = “5”;

*const* y = 10;

*const* result = x + y;

console.log (result);

The result for this will be 510 because when we’re dealing with a string, the two values are going to be squished together. If we reverse and say the equation is y + x, we’ll obtain 105.

When we add a string to a number in JavaScript, the default thing that will happen is the number will be changed to a string. It is what we call concatenation. An addition of 2 strings.

“hello” + “hello” = “hellohello”

How to convert a string into a number.

***Number*(x)**

Use the function named *Number.* Example. The answer here is 15

*const* x = “5”;

*const* y = 10;

*const* result = y + *Number* (x);

console.log (result);

If we have a string that contains letters and numbers, the answer here will be NaN. Not a Number. The reason is because hello is not a number. When we have an addition of NaN and a Number, the result will be NaN.

*const* x = “hello5”;

*const* y = 10;

*const* result = *Number* (y) + *Number* (x);

console.log (result);

**parseInt(x)**

parseInt specifically converts a string into an integer. An integer is a number that doesn’t contain decimals. If we add two numbers together and one of them has decimals, the decimal won’t be taken into consideration.

*const* x = “5.6”;

*const* y = 10;

*const* result = y + parseInt(x);

console.log (result);

**parsefloat(x)**

parsefloat will include the decimal. parsefloat and parseInt only work on strings. With this equation, we obtain 10. False equates to 0. If the statement were true, the result would be 11 since true equates to 1.

*const* x = false;

*const* y = 10;

*const* result = y + *Number* (x);

console.log (result);

How to convert other values to strings:

*const* name1 = “tim”;

*const* age = 23;

*const* sentence = “hi my name is” + name1 + “and I am” + age + ”years old”;

console.log (sentence);

This is a multiple string concatenation. The result will be:

hi my name is Tim and I am 23 years old

When we add multiple strings together, if we don’t want the characters to be squished together, we need to add spaces between the different elements of the equation. In this case 23 was converted automatically to a string since everything is considered a string when another string is already present in the equation. If we want to specifically convert 23, we will use age.toString()

Example of operation:

*const* name1 = “tim”;

*const* age = 23;

console.log (typeof age.toString());

Another way to do it is to write this.

*const* name1 = “tim”;

*const* age = 23;

console.log (*String*(age), age);

The result here will be 23 23. The first one is a string, the second one is a number and that’s why they have different colors.

How to convert an entire expression to a string.

*const* name1 = “tim”;

*const* age = 23;

console.log (2 + 3).toString());

The result here will be 5. If we take out the parenthesis isolating 2 + 3, we will obtain an error because we can’t have a number directly followed by .toString. If we have this, we’ll obtain 23 because 3 will be part of an expression. That’s when the concatenation will happen.

*const* name1 = “tim”;

*const* age = 23;

console.log (2 + (3)).toString());

How to add a value to a number.

If we add + in the log, it will automatically convert the string 1997 to a number.

*const* year = “1997”;

*const* age = 23;

console.log (+year);

If we add another +, 1997 and 23 will be added together and we’ll obtain 2020. This is known as the unary operator. The unary operator is the possibility to put a + in front of a string and automatically convert it to a number.

**COMPARISON OPERATORS**

Operators used to compare different values. Typically, we use them for numbers. We can also use strings but it’s less used.

|  |  |
| --- | --- |
| == | Equal to |
| === | Strict equal to |
| != | Unequal to |
| !== | Strictly not equal to |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |

**==**

A condition is anything that’s going to evaluate to true of false. This example will evaluate to a Boolean, true or false. In this case, it’s true because 5 = 5

*const* value = 5 = = 5;

console.log (value);

The double = sign is essentially looking at the value on the left-hand side and the value on the right-hand side and evaluating if left equals right.

When we have this scenario, the result will also come up as true. Why? Because when we’re using this operator, we are performing a loose equivalent.

If we had 0 == false, the result would be true because 0 is equivalent to false. They’re not the same type but their value is equivalent. That’s the reason why we use the triple = sign with this operator when we want to have a value to is strictly equal. We reinforce that they’re identical and the same type.

**===**

The value AND the type are equal. If we use two strings, for them to be equal, we also must take the spaces into consideration. “Hello” is not the same as “Hello “ or “HellO”. It would be the same for the == with these strings because they don’t have the same value.

If we were to use:

undefined == null, the result would be true. If we were to use undefined === null the result would be false

**!=**

works the opposite as ==

**!==**

works the opposite as !==

The results will then be the opposite as the examples cited above.

**<** and **>**

work the same as in math for numbers. For strings, if we have

“a” > “b”;

The result will be false because b comes after a in the alphabet.

If we have this:

*const* value = “D”> “b”;

“A” = 65

“a” = 97

console.log (value);

Capital D is smaller than b because the ascii value of D is smaller than b. Each letter on the keyboard has a different value. A for example is 65 and a is 97. We then know that a is bigger than A. Usually when we deal with these, we only deal with the regular letters and the capital letters, not the special characters.

a = 97, b=98, c=99 etc

A=65, B=66, C=67

If we have these strings with these letters, the first letter from the first string will be compared to the first letter from the second string. We then move on to the second letter and so on.

How to see what the ascii character code of a specific character:

*const* str = “hello”

console.log (str.charCodeAt())

console.log (value);

Inside the non bold brackets, we’ll add an index. An index is a number assigned to each letter of a character. Example, we have the string “hello”, inside the string is the character/word hello. For each letter of the word, a number is assigned, and it always starts with the first letter being equal to 0.

|  |  |
| --- | --- |
| h | 0 |
| e | 1 |
| l | 2 |
| l | 3 |
| o | 4 |

Now if we want to know the value of h, we’ll add the index 0 inside the non bold bracket. And the result will be 104.

**<=**

means less than OR equal to. Same principle as in mathematics when we deal with numbers. When we deal with strings, it’s different. If we have 6>= “6” JavaScript will attempt to convert the string into a number. If the string is a valid number, it will convert it, if it’s not numeric or if it’s an empty string, it will convert into NaN. So, by default this expression will have the result false.

6>= “hello”

If we compare two numeric strings, it will compare two ascii codes.

Example

*const* value = 10 > = “2”;

console.log (value);

We compare 1 to 2 because the string starts with the character 1. If we had “20” >= “2” the statement would be true because we compare 2 to 2. If we want to compare those 2 strings as numbers, we need to convert the strings into numbers and then compare them. Example

*const* value = *Number* (“20”) > = “2”;

console.log (value);

Here we have the string “20” being converted into a number. The equation will then automatically consider 2 as a number as well.

Note : Ascii =American standard code for information exchange.

**LOGICAL OPERATORS**

Allow us to combine multiple conditions together to create more complex expressions.

|  |  |
| --- | --- |
| && | End |
| || | Or |
| ! | Exclamation point |

&& will evaluate if both the left and right end side are true

|| will evaluate if either left or right end side are true

! will either reverse or negate the current Boolean is

**&&**

Example for &&

*const* and = true && false;

console.log (and);

This statement is false because we don’t have true on both sides. && will evaluate if both left and right end sides are true. If we had true && true the statement would be true. If we had false && false the statement would be false. If we had this example

*const* and = x && y;

console.log (and);

We would have different types of combinations. Since x and y can each be either true or false, we could have.

False && False 0 0

False && true 0 1

True && False 1 0

True && true 1 1

|  |  |
| --- | --- |
| /\*  x y  0 0  0 1  1 0  1 1  \*/ | This is what we call a truth table. Only one of these statements would be correct. The truth table is useful when we have a lot more variables included. |

If we added a variable:

*const* and = x && y && z

We would list out all of the possible values. The easiest way to do so is to start with the last column and alternate between 0 and 1 until you get to 2 to the exponent of the number of variables values. That’s how you get the amount of possible combinations, 2 raised to the exponent of the number of variables. In this case we have 2 raised to the exponent 3 since we have 3 variables ( x y z )

/\*

x y z

0 0 0

0 0 1

0 1 0

0 1 1

1 0 0

1 0 1

1 1 0

1 1 1

\*/

We start with z and we alternate every line of the column with 0 1

The next column over were going to alternate every other line with 0 1

The next column over we alternate every four lines of the column with 0 1

The only correct combination is going to be the last one 1 1 1 since they’re all true

**||**

*const* and = true || true;

console.log (and);

We’ll get true as a result because the left or the right end side is true. If we had true && false the result would also be true. If we had false && false, the result would be false. With || we have 3 different possibilities to have a true statement.

**!**

*const* and = !false;

console.log (and);

In this operation, we have a reverse of whatever statement is written so !false would be true and !true would be false. If we use this type of operation for a different kind of statement.

*const* and = ! (5 < 4);

console.log (and);

We write the not in front of whatever statement. Here we have the statement in brackets being false. With the ! we have !false so the result is true.

*const* example = ! (5 % 2 = = 1 && true == 1 && undefined != null || 5 – 2 ==1);

console.log (example);

For a statement like this where we don’t have a lot of parenthesis and aren’t sure where to start.

Order

!

&&

||

**CONDITIONALS**

If, else, else if

We can use it to control the flow of our program. We can check if our condition is true and if it’s true, we can enter a specific block of code. If it’s not true we can enter a different block of code.

**If**

Example

Inside the parenthesis we put anything that evaluates to true or false.

*const* age = 21;

if (age >= 16) {

console.log (“You can drive”);

}

Anything inside of the curly braces is going to happen if the statement inside the parenthesis is true. As soon as we get outside of the curly braces the rest of the code will happen no matter what.

*const* age = 15;

if (age >= 16) {

console.log (“You can drive”);

}

console.log (“Thanks for playing”);

If we change the constant to 15, the statement will be false. Because you have to be 16 to be able to drive so being 15, you cannot drive. Only thanks for playing will be displayed since it’s separate from the previous statements inside the curly braces.

**Else**

Optional and comes after an if statement. If the if statement is true, it will run the following code. If it’s not true, we can use the else statement. Example

*const* age = 81;

if (age >= 16) {

console.log (“You can drive”);

} else {

console.log (“You can NOT drive);

}

console.log (“Thanks for playing”);

If the age is 81, the first statement is true, then the code will be run. If we change the age to 15, the result will be you can NOT drive because it will run the else code.

*const* age = 15;

if (age >= 16) {

console.log (“You can drive”);

} else {

console.log (“You can NOT drive);

}

console.log (“Thanks for playing”);

**Else if**

Used if the if statement is false. We then will go to the next line and see if that statement is true. If it’s true we’ll run the code. If it’s false we’ll go to the next line, the else statement. We can have multiple else if statements. If none of the statements were true, we would get no result, and nothing would run.

*const* age = 15;

if (age >= 16) {

console.log (“You can drive”);

} else if (age === 15) {

console.log (“Only one year away”);

} else {

console.log (“You can NOT drive);

}

console.log (“Thanks for playing”);

**NESTED CONDITIONALS**

Concept that we can nest codes inside other codes, for example the if/else statements previously seen. We can do this process until infinity, but it’s not recommended. It is however useful to do a couple nested conditionals for some specific situations.

Example

If we want to do a poll where we ask questions to the user and we want to make sure to hit a key demographic, we’ll ask previous questions before the actual survey. Example

We want to make sure the user is from Canada. We need them to enter the country they live in. If they do, we’ll ask them another question.

We have 2 possibilities of answer. Capital two letters for the symbol of the country or entire country spelled. If we want them to be living in the us and they answer united states, we can add a nested conditional and write down “thanks for participating” aka we don’t need anything else from them.

If we need more info, we can also ask them their age. That will be nested after the question about the country.

*const* country = prompt (“what country do you live in?”);

if (country === “CA” || country === “canada”) {

*const* age = prompt (“Please enter your age”);

alert (age);

} else {

Alert (“thanks for participating, we do not need any more info)

}

In this case, we ask what country they live in. We get the answer Canada, so we add a nested conditional asking about their age. As a result on the webpage, it will show up as a first question, then once answered, if the answer is Canada, the user will get a follow up question about their age. If the answer is united states, the user will get a sentence saying thanks for participating.

If we want to add another question that is going to be linked to some text, we can go into the html parent file and write down a small paragraph with the text we’ll need to link it to the JavaScript file, to style it if necessary. The paragraph needs to be written inside the body tag of the html document but cannot be written underneath the script tag or it will show up as an error.

*const* country = prompt (“what country do you live in?”);

if (country === “CA” || country === “canada”) {

*const* age = prompt (“Please enter your age”);

if (age >= 18) {

*const* color = prompt (“Please enter your fav color”);

alert (color);

} else {

alert (“thanks for participating, we do not need any more info)

}

} else {

alert (“thanks for participating, we do not need any more info)

}

If we want to add a statement after the user has given a specific answer, we can write this. In this case, the statement follows if the answer is blue.

*const* country = prompt (“what country do you live in?”);

if (country === “CA” || country === “canada”) {

*const* age = prompt (“Please enter your age”);

if (age >= 18) {

*const* color = prompt (“Please enter your fav color”);

if (color === blue) {

alert (“This is my favorite color as well”);

} else {

alert (“thanks for participating, we do not need any more info)

}

} else {

alert (“thanks for participating, we do not need any more info)

}

Inside the html document we’re going to add the paragraph tag with an ID, inside the body tag.

<body>

<p id = “color” ></p>

<script src = “test.js” ></script>

</body>

Inside the JavaScript document, instead of having the if statement of “that is my favorite color as well”, we’ll write the link to the html document.

*const* country = prompt (“what country do you live in?”);

if (country === “CA” || country === “canada”) {

*const* age = prompt (“Please enter your age”);

if (age >= 18) {

*const* color = prompt (“Please enter your fav color”);

*const* text = document.getElementById (“color”);

text.innerHTML = color;

text.style = “color:” + color;

} else {

alert (“thanks for participating, we do not need any more info)

}

} else {

alert (“thanks for participating, we do not need any more info)

}

In the JavaScript document we have the prompt question, then underneath it, we inform that there will be a text as an answer and that text will be referenced by an ID. The text will be located inside of a HTML document and that’s why we have inner.HTML. Then we style the text with the string so the answer will be written in text but also in color according to the answer the use gives.

**SWITCH STATEMENTS**

They don’t exist in every program but do in JavaScript. They are similar to the else/if statements because they allow you to control the flow of your program and execute different blocks of codes depending on specific cases.

If we wanted to reply to specific answers to different user answers, we could use the else/if/else if statements and write this. Depending on the color the user is writing he’ll get a specific answer.

*const* color = prompt (“Enter your fav color: “);

If (color === “red”) {

} else if (color === “blue”) {

} else if (color === “orange”) {

}

There is a better way to write this. Instead, we can write a switch statement. The switch statement will be written like this,. Switch, parenthesis, and curly braces.

*const* color = prompt (“Enter your fav color: “);

switch () {

}

Inside the parenthesis we write the value that we want to switch on. In this case, color

*const* color = prompt (“Enter your fav color: “);

switch (color) {

case “red”:

alert (“red”)

break;

case “blue”:

alert (“blue”)

break;

}

Underneath the switch we will write the different cases depending on the colors. If the answer is red, the text statement will be whatever is inside the string in the alert. In this case the word red.

If the answer is blue, same principle, but the text statement will be blue.

Attention: We have to write the break underneath each case to notify that the case is over and that we’ll move on to the next case underneath.

The switch statement is useful when we have a lot of different values we want to change. If the statement only has two different answers, we’ll probably use else/if since it’s only 2 values.

The default case:

What happens if we have a switch statement that is the equivalent of the else statement? The one that happens if none of the cases were the case, in this example if none of the colors in the cases matched the original one.

We write this:

*const* color = prompt (“Enter your fav color: “);

switch (color) {

case “red”:

alert (“red is pretty cool”)

break;

case “blue”:

alert (“I hate the color”)

break;

case “orange”:

alert (“orange is my fav”)

break;

default:

alert (“I don’t know that color”);

break;

}

If we were to not include the break inside the switch statement, we could get what we call a fall through. A fall through could be voluntary sometimes, but it typically isn’t.

The switch statements are flexible, meaning we CAN have else/if statements inside of the cases

Other Example:

We want to ask the user their age. If we type this, nothing will happen. Why? Because the prompt is reading as a string. And the switch with the different cases is reading as a triple equal sign === aka the strict equality operator. We have to convert our age into a number to be able to use it in the switch statement properly. What we’ll do to fix the issue is wrap the prompt inside a number and

*const* age = *Number* (prompt (“Enter your age: “));

switch (age) {

case “10”:

alert (“you are 10”)

break;

case “20”:

alert (“you are 20”)

break;

case “30”:

alert (“you are 30”)

break;

}

**ARRAYS**

A way to store multiple elements in an ordered collection. We want to store them in only one object or one variable.

Example

We write the constant array like this *const* arr. We then place inside the square brackets all the elements we want included inside the array.

*const* arr = [10, 20, 30];

We can mix different elements inside the array. Numbers, strings, Boolean etc. We can also put an array inside another array.

To access an element, we will reference its index. In programming we always start counting at zero. So every single element in our array can be accessed by an individual integer starting at 0. In this case if we want to access 10, we’ll refer to it as 0 since it’s the first element. If we want to refer to 20, we will use 1, if we want to refer to “hello”, it will be 2.

*const* arr = [10, 20, “hello];

console.log (arr [0])

If we try to access an index that doesn’t exist, we’ll get undefined.

With the array we can also assign values to an index

Example

If we want to change the value of 10 to 2, we will write this.

*const* arr = [10, 20, “hello];

arr [0] = 2 ;

console.log (arr);

How to access the last element of the array without knowing how many elements are in it. We know that the value of the last element is going to be the total amount of elements minus 1, since we start counting at 0. We would write this. The arr.length is going to give us the length of the array.

*const* arr = [10, 20, “hello];

arr [arr.length – 1 ] = 2 ;

console.log (arr);

If we want to change the last element of the array, we’ll write for example 100.

So in the result, the last element will be 100

*const* arr = [10, 20, “hello];

arr [arr.length – 1 ] = 100 ;

console.log (arr);

result : [ 10, 20, 100]

Other ways to initialize or create arrays

*const* arr = new *Array* (20, 20);

console.log (arr)

If we have a few elements, it will work perfectly fine. If we have only 1 element, it will start acting strange. For example here, we only have element 20 and the result will show [<20 empty items>]

*const* arr = new *Array* (20);

console.log (arr)

When we have only 1 element that is a number, we initialize the array with that number of values. Example, we have 20 different values inside this one because we wrote 20. If we had written 30, we would have 30 different values. If we have only 1 element that is a string, the result will work fine.

What’s the real positive aspect to using an array?

It is not the place different element inside. We use the array to grow or change the size if the array based on some dynamic input. Example

To do list created by user. We don’t know what the amount of elements are going to be added by the user so we’ll add to the array every time a new element is added to the list.

**Different types of array**

If we want to ask the user to enter information 3 times

We can enter the 3 questions and then write the array like this

*const* name1 = prompt (“Enter your name”)

*const* age = prompt (“Enter your age”)

*const* hairColor = prompt ("Enter your hair color”)

*const* arr = [name1, age, hairColor]

Push function

If the array already exists and we want to add to it. We use the function known as push. It will push the elements inside of the pre-existing array. (note: If the array already exists it is located above the new information in the code). With this method, the elements will be added at the end of the array. If the array is just empty braces, the element will become the first element of the array. If we have elements already inside the array, it will be the 5th element. In this case, we write the push function for each of the elements we want added then we alert that we want that push action.

*const* arr = [];

*const* name1 = prompt (“Enter your name”)

*const* age = prompt (“Enter your age”)

*const* hairColor = prompt ("Enter your hair color”)

*const* arr = [name1, age, hairColor]

arr.push (name1);

arr.push (age);

arr.push (hairColor);

alert (arr);

If we wanted to add these elements to the array that weren’t empty strings, we can write an if statement. In this case, if the name is not equal to an empty string, we will push the element inside the array

*const* arr = [];

*const* name1 = prompt (“Enter your name”)

*const* age = prompt (“Enter your age”)

*const* hairColor = prompt ("Enter your hair color”)

*const* arr = [name1, age, hairColor]

if (name1 !== “ “ ) arr.push (name1);

arr.push (age);

arr.push (hairColor);

alert (arr);

Pop function

The pop function allows us to remove an element from the end of the array. In this case, we ask the user if they want to remove the last element. If the answer is yes, then we’ll proceed and delete it by typing the push the arr.push in the same line. We then add an alert.

*const* arr = [];

*const* name1 = prompt (“Enter your name”)

*const* age = prompt (“Enter your age”)

*const* hairColor = prompt ("Enter your hair color”)

if (name1 !== “ “ ) arr.push (name1);

if (age !== “ “) arr.push (age);

if (hairColor !== “ “) arr.push (hairColor);

*const* del = prompt (“would you like to delete the last element?”)

if (del === “yes”) arr .pop ();

alert (arr);

Nested arrays

In this example we have 3 arrays. We want to nest arr2 and arr3 inside of arr. We then write the push function.

*const* arr = [];

*const* arr2 = [1,2,3];

*const* arr = [3,4,5];

arr.push (arr2);

arr.push (arr3);

console.log (arr);

Result: [ [ 1,2,3], [3,4,5] ]

How to access elements inside of a nested array:

If we were to type index 0, we would get the first array that is nested inside the main array. Why? Because this whole array is the first element so it’s at index 0.

*const* arr = [];

*const* arr2 = [1,2,3];

*const* arr = [3,4,5];

arr.push (arr2);

arr.push (arr3);

console.log (arr [0]);

Result: [ 1,2,3]

If we wanted to access an element that’s inside of an array nested in the main array, we would add another square bracket right after the first bracket. Example:

Inside the log we write [1] for the second array and we write [0] for the first element of that same array. In this case we would access 3 that is located inside of [3,4,5]

*const* arr = [];

*const* arr2 = [1,2,3];

*const* arr = [3,4,5];

arr.push (arr2);

arr.push (arr3);

console.log (arr [1][0]);

Another example if we wanted to access “hello”, we would have this

*const* arr = [];

*const* arr2 = [1,2,3];

*const* arr = [3,4,5, [“hello”]];

arr.push (arr2);

arr.push (arr3);

console.log (arr [1][3][0]);

What is dimensional arrays? It’s an array nested inside of another array. So if we have two dimensional arrays, we have this [ [], [], ]. It’s two levels of arrays, not 2 arrays. If we want 3 dimensional arrays we will have [ [ [], [], ] ]

**ARRAY METHODS**

A method is anything that follows the dot notation. Beyond push and pop we have other types of methods.

.toString()

This following method prints out the string representation of the array.

*const* arr = [1,2,3];

*const* stringArr = arr.to String ();

console.log (stringArr);

The original array includes the spaces and the brackets. The string does not. It just gives us the value of what’s inside the array, separated by commas.

Different methods that allow to modify what’s inside of the array:

.shift()

The shift method removes the first element from the array and returns it. If the array is empty, undefined is returned and the array is not modified. The result will be [2, 3] 1

*const* arr = [1,2,3];

*const* removed = arr.shift ();

console.log (arr, removed);

The first element is removed from the array and returned outside of it and after it. If we just do arr.shift() and don’t include the variable *const* removed, the first variable will simply be removed. The result will be [2, 3]

*const* arr = [1,2,3];

arr.shift ();

console.log (arr);

.unshift()

It will insert new elements at the beginning of the array and return the new length of the array. The result will be [-1, 1, 2, 3]

*const* arr = [1,2,3];

arr.unshift (-1);

console.log (arr);

.sort()

This one organizes the array but the result we’re getting is not in ascending number. Why? It’s using the string representation of the array. Every element will be organized according to its ascii value. -8 will be considered by the dash. The dash has the lowest value of all. Then for 23 we will consider 2 because that’s the beginning of the string value. then 45 will have 4. Then 7 will just be 7.

*const* arr = [45,23,45, -8, 7];

arr.sort ();

console.log (arr);

To be able to sort it according to the actual number, we can use this method. We’re going to create a function and basically tell the program that that’s how we want the organization to be done. We’ll use this equation (can be confusing but that’s the way to do it). If we were to do the opposite and have b-a e would have the descending order of 45, 45, 23, 7, -8

*const* arr = [45,23,45, -8, 7];

arr.sort ((*a, b*) => *a* - *b*);

console.log (arr);

.concat()

This method will combine different arrays in a concatenation. So, we’re adding arr2 to the end of arr. This method does not modify the pre-existing arrays. It returns with a new array that is a concatenation of the others. The result will be [45, 23, 45, -8, 7, ‘hello’, ‘world’]. Written in different colors because different types.

*const* arr = [45,23,45,-8,7];

*const* arr2 = [“hello”, “world”];

*const* arr3 = arr.concat (arr2);

Console.log (arr3);

.splice()

This method removes an element from the array. if we wanted to remove an element that was in the middle of an array. This method takes two arguments. The first argument is the index in the array to start removing elements from. The second argument is the delete count, the number of elements that we should remove from the array. in this case if we wanted to delete 45 (the one in the middle, we would write this. The 2 stands for the index order, and 1 stands for how many elements we want removed. If we wanted to remove 2 elements starting at middle 45, we would type arr.splice (2,2) and we would remove 43 and -8 because it’s the following element.

*const* arr = [45,23,45, -8, 7];

arr.splice(2, 1);

console.log (arr3, arr, arr2);

If we add the *const* removed in front of, it will return the elements we removed and put it inside a new one. Result [45, 23], [45, -8, 7]

*const* arr = [45,23,45, -8, 7];

*const* removed = arr.splice(2, 3);

console.log (arr, removed);

.join(“,”)

Joins all of the elements inside of the array and creates a string out of them. The result is 45, 23, 45, -8, 7

The comma inside of the bracket is known as the delimiter or the separator for this kind of joined string. We can use other types like the pipe| and the result will show 45|23|45|-8|7, or we can use the space etc.

*const* arr = [45,23,45, -8, 7];

*const* str = arr.join(“, “);

console.log (str);

45, 23, 45, -8, 7

.slice()

Returns a portion of the array, also known as a subarray which is a subsection of the array, meaning some of the elements. To indicate what we want we have a start value and an end value. inside the bracket we indicate the index we want to start at and the index we want to end at. If we only add one number inside the bracket, by default it’s reading that we only write a start value and will. In this example we start at index 1 and grab all of the remaining elements after it

*const* arr = [45,23,45, -8, 7];

*const* sl = arr.slice(1);

console.log (sl);

*const* sl is a new array so if we log this in, we’ll have this result. [45,23,45, -8, 7] [23,45, -8, 7]

here we console.log both the original array (arr) and the new one (sl)

*const* arr = [45,23,45, -8, 7];

*const* sl = arr.slice(1);

console.log (arr, sl);

The end index will not be included inside the result. The result is [23, 45]

*const* arr = [45,23,45, -8, 7];

*const* sl = arr.slice(1, 3);

console.log (sl);

If we wanted to remove 23, 45, -8 and 7, we could write *const* sl = arr.slice(1, 5). 5 is not a valid index number but will then include the last element.

**STRINGS**

The symbols for the string are of two types. The single and the double quotation mark. When we have a string and want to include some quote inside of it, we can then use the other type. For example, we use the double quote for the string, then we’ll use the single quote mark inside of it. An issue arises though with only having two types of representation of a string. If we wanted to add both a single and then a double quote inside of a string, the code would read it wrong, and some characters would have an error to it. “” or ‘ ‘

One solution is to use an escape character. In this case, we include the backslash inside of the string and place it in front of the character we don’t want to be read as the closing of the string but just another character that is part of the string.

**‘ “ \’ ‘**

What happens if I wanted to include a single backslash inside my string without it having anything to do with the string. We would have to write it double. //

console.log (“ \\”);

Result: \

Another type of special character is \n

We use that character inside a string if we want to go to the next line between different elements of the string. Example we have the string (“hello world”). One way to have hello and world on two different lines is to write two different strings but the easiest is to just place \n in between them. That character will be invisible as well.

console.log (“hello \n world”);

Result :

hello

world

Another special character is the \t

This is the backslash tab and creates a tab in front of the word it precedes.

console.log (“hello \n\t world”);

result:

hello

world

Back ticks

Another way to create a string character in JavaScript.

We can use the back tick. The back ticks are used when we want to format a string by passing in some kind of variable or expression. Top left-hand side of the keyboard. We have special characters inside of the string, but the back tick allows us to read them as part of the string. The result is hello 13 world

*const* str = ‘hello ${6 + 7} world’;

console.log (str);

This method is useful when we want to use different types of variables inside a string without having to use concatenation. The result will be hello tim world

*const* name1 = “tim”;

*const* str = ‘hello ${name1} world’;

console.log (str);

Other properties of strings

We can access different indices of the string.

*const* name1 = “tim”;

*const* str = ‘hello ${name1} world’;

console.log (str [0]);

Here we want to know what the first indices of the string is, the result is h because we typed in index 0.

We can also look at the length of the string. The result will be 15

*const* name1 = “tim”;

*const* str = ‘hello ${name1} world’;

console.log (str.length);

If we tried to change one of the indices of the string, nothing would happen. Strings are known as immutable data types whereas arrays are mutable.

**STRING METHODS**

All these methods don’t modify the original string. They create a new one independent from the rest.

str.slice()

works the exact same way as array. In this example, we have the index 6 and 8. So we start on the space before w and we stop on the letter o. result we get is wo. We get a new string though; the original is not being modified.

*const* str = “hello world”;

*const* str2 = str.slice (6, 8);

console.log (str2);

str.replace()

In the first string, we tell what element we want to replace, in the 2nd string we write what we want it to replace.

*const* str = “hello world”;

*const* str2 = str.replace (“l”, “!”);

console.log (str2);

If we wanted to replace all of the instances of the letter L, we would write str.replaceAll and all the “l” from hello world would be replaced. Result would be he!!o wor!d

str.toUpperCase() OR st.toLowerCase()

it will change the characters into capital letters. Useful when you check user input. Where no matter if they type in lower or upper case, the results would be the same. Example

*const* str = “hello World”;

*const* str2 = str.toLowerCase ();

console.log (str2);

if (str.toLowerCase() == “yes”)

str.trim()

removes some spaces from the beginning or the end of a string.

trimStart() removes it from the beginning of the string.

trim.End() removes it from the end of the string.

str.split()

splits a string into different elements. It will strip out all the spaces between the words and put those words in different elements. If we include a bunch of spaces inside the strings, we will obtain a bunch of empty strings in between the regular ones. The space is not the only type of separator between words. The same thing can happen if hello world my name is Tim was written like this: Hello, world, my, name, is, Tim

*const* str = “hello World my name is tim”;

*const* arr = str.split (“ “);

console.log (str);

console.log (arr);

the result is [‘hello’, ‘World’, ‘my’, ‘name’, ‘is, ‘tim’

**FOR LOOPS**

We can use them to repeat a block of codes a known number of times. One of the most common things used in programming.

How to write a for loop. In the parentheses we say what we want to do and how long we want it to loop for. To do that we need to write a numeric variable

for (*let* I = 0; I < 10; i++) {

}

*let* i = 0 we declare our variable. This is the looping variable. i stands for index

i>10 condition where we want to loop until this condition is false.

i++ increment condition. What we want to do to adjust the variable at the end of very single loop.

Anything inside of the curly braces is what it going to be repeated.

In this case we start the loop at 0. Every time we add 1 because ++ is the symbol for +1 increment. And we stop the loop just under 10 which is 9.

for (*let* i = 0; i < 10; i++) {

console.log (i)

}

The result will be 0 1 2 3 4 5 6 7 8 9 but all written on a different line

Another example

for (*let* i = 10; i <= 100; i+=10) {

console.log (i)

}

The result will be 10 20 30 40 50 60 70 80 90 but all on a different line

In this case let i = 10 is the starting point. i < =10 is the ending point.. i+= is the increment. So, every time we’ll add 10 until we reach 100 or just under 100.

We can also loop in the opposite direction. We just change our final value. For example, the final value is -100.

Important keywords for loops

We often will loop through a string, an array to find some kind of value. We can also look through a set of numbers to find one with a certain property.

*const* divBy2 = [];

for(*let* i = 0, i <= 100; i++) {

if (i % 2 == 0 && i !==0) divBy2.push (i)

}

console.log (divBy2);

In this case we want to print out all of the values divisible by 2, between 0 and 100.

*const* divBy2 = [ ] is what we write down to declare an array and to say we want all the elements that are divisible by 2.

i % 2 ==0 means 0 is the remainder after dividing i by 2. if the remainder is 0 it means the number is divisible by 2. To make sure that I is not equal to zero we will write down that i !==0

To exit the for loop we can use the break function. In this case, we want the loop to stop after 20 elements in the array. we stop at 40 because it is the 20th element of the array.

*const* divBy2 = [];

for(*let* i = 0, i <= 100; i++) {

if (i % 2 == 0 && i !==0) divBy2.push (i);

if (divBy2.length >= 20) break;

}

console.log (divBy2);

Beyond breaking we have the option to skip. In this case we want to collect the values that are divisible by 2 but don’t want to include the values divisible by 3. We can use the function continue. continue will skip to the next element.

*const* divBy2 = [];

for(*let* i = 0, i <= 100; i++) {

if (i % 3 === 0 && i !==0) continue;

if (i % 2 === 0 && i !==0) divBy2.push (i);

}

console.log (divBy2);

**FOR OF**

The previous method shown about looping over a string can be used but it’s not what you would prefer.

Looping through an array

*const* strs = [“one”, “two”, “three”];

for(*let* i = 0, i < strs.length; i++) {

*const* str = strs [i];

}

In this example, we have a string, and we want to look for a number and an index. This is the method we would use in the previous lesson, but we can write another one. The following syntax is the better way to write it down.

*const* strs = [“one”, “two”, “three”];

for(*const* str of strs) {

console.log (str);

}

The result will be

one

two

three

This way allows me to loop through all the elements of the array. In this example, we have a variable called str and for every single reiteration of this loop it’s going to be equal to the next element that exists inside of my array.

The reason why we are allowed to use *const* rather than *let* is because every single time this loop repeats, we are going to re-declare the next variable str to be equal to the next element inside of the array. It’s not changing str it’s creating a new one. This method is useful when you don’t care about the index or the order of the elements, but only care about their value. However, there is a way to get both the value AND the index of the elements. We’ll use entries.

Note: something called an iterable is something we can iterate over or loop over. Meaning we have multiple values or elements inside of an object that we can loop overusing for example a for loop. Strings and arrays are iterable.

Since every single time we go through entries, we get an array that contains something like [0, “one”], it can break the array apart and assign each value from the array to a variable.

*const* strs = [“one”, “two”, “three”];

for(*const* [I, str] of strs.entries()) {

console.log (i, str);

}

It’s essentially an array that contains all of the values that we have inside of our array except it contains an array for each element that has the index as well as the value. so when we loop through the str.entries, we get mini arrays and we’re saying that we want I to be equal to the first element of that array and str to be equal to the second element of that array. so I = 0 and str = one.

Looping through a string

The entries method is not accessible with the const str.

Nesting For Loops

Example: We want to create a 2-dimensional array structure also known as matrix.

[[1, 2, 3],

[4, 5, 6],

[7, 8, 9]]

We want these 3 arrays to be inside one big array. For that, we’ll use a nested for loop, which is a for loop inside of a for loop. First of all we create an array that’s going to include our other arrays.

*const* matrix = [ ]

After that, we need to loop through the number of rows/subarrays that we want to put inside of our array. we’ll write.

For (*let* i = 0; i < 3; i++) {

}

Inside of this for loop, we’re going to try to create the smaller arrays that all have 3 elements. So, we write the variable const number and we need to write those elements inside of the empty array. Instead of filling up the empty array with [i+1, i+2 etc] we’re going to use another technique.

For (*let* i = 0; i < 3; i++) {

*const* numbers = [ ];

}

We’re gonna write a for loop inside of the other for loop. We’ll use J because we want to have another name than i. we need a different variable name not to get mixed up.

The second for loop happens every single time we have a reiteration of the first for loop.

Once we have that all written out, we have the empty array populated. We then want that array to be put inside a matrix. So we’re going to write matrix.push (numbers);

*const* matrix = [] ;

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

*const* numbers = [];

for (*let* j = 0; j <3; j++) {

numbers .push (j + 1);

}

matrix.push (numbers);

console.log (matrix)

This result will give us the two dimensional array [[1,2,3], [1,2,3], [1,2,3]]

For the numbers to increase we’ll have to modify the for loop with j.

*const* matrix = [] ;

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

*const* numbers = [];

for (*let* j = i\*3 j < i\*3+3 ; j++) {

numbers .push (j + 1);

}

matrix.push (numbers);

console.log (matrix)

This result will be [[1,2,3], [4,5,6], [7,8,9]]

Other example

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

[[7,8], [9,10], [11,12]]

*const* matrix = [] ;

for (*let* i = 0; i < 2; i++) {

*const* arr = [];

for (*let* j = 0; j < 3; j++) {

*const* numbers= [];

*const* start = i \* 6 + j \* 2 + 1;

for (Let k = start; k < start + 2; k++) {

numbers.push (k);

}

arr.push (numbers);

}

matrix.push (arr);

}

**WHILE LOOPS**

A while loop is going to work the same way as a for loop but the difference is with a while loop we don’t know how many times we’re going to be looping. A while loop is the general loop system and the for loop is a specialized category where we know how many loops we want to have.

How to write the while loop

While () { }

We’re going to write the condition in between the regular braces and that condition is either going to be true or false. As long as the condition is true, we’re going to ask the computer to keep doing whatever is written under the block of code. If the condition is false, we’re going to stop.

*let* i = 0;

while (i < 10) {

console.log (i);

i++;

}

Note: with the while loop, it’s necessary to write the increment (i++). If we don’t, we’re going to be stuck in an infinite loop, a condition that will forever be true. In this case if we didn’t write the increment, the terminal would show the endless result of 0. To undo that action, CTRL+C.

Another example: we’re going to ask the user to enter a number from 1 to 10. We’re going to keep asking the same question until they give an answer that is a number from 1 to 10.

*Let* number = *Number* (prompt(“Enter a number between one and ten…”));

While (! (number >= 1 && number <=10)) {

number = *Number* (prompt(“Enter a number between one and ten…”));

console.log (number);

First line: Here we enter the question, and we want to make sure the answer is converted to a number, so we write it in front of the prompt.

Second line: we enter the while loop and the condition inside the purple braces is what we want to get. A number greater or equal to 1 and smaller or equal to 10. We’re going to negate that condition with the exclamation point right in front of it. Why? Because as long as the condition is false (the user doesn’t give an answer between 1 and 10), then we’re going to keep repeating the same question, which is why we have that question being repeated under that negated condition.

The inconvenience with this method is that we’re pretty much repeating the same line of code twice. The reason is because we need to declare the number variable before we can use it in the condition. If we took out the first line, we would get an error.

The way to go around it is to write the line of code like this. We declare that we have a number variable, then we use the do while loop. Inside the do while loop we write what we want to get. Whatever is inside the do means we’ll do it one time and we’re going to continue doing that while the condition is true.

*Let* number;

do {

number = *Number* (prompt(“Enter a number between one and ten…”));

} while (! (number >= 1 && number <= 10) );

console.log (number);

We can also write the same request while using a break statement.

while (true) {

*const* number = *Number* (prompt (“Enter a number between one and ten…”));

if (number >= 1 && number <= 10) break;

}

console.log (number)

In this case we’re going to state the opposite. In the previous example, we were looping until the condition was true. Here we’re going to say, when the number is within the range 1-10, THEN we’re going to break. This is a different writing style and it is easier to read.

Note: We can use the continue key word with the while loop as well.

**SETS**

Similar to an array but we use it to keep track of unique elements. We won’t keep track of the frequency of different elements. We only want to know if an element exists or doesn’t exist.

The reason we choose to use sets instead of arrays when we want to check if an element exist/doesn’t exist or if we want to add or delete an element. It’s easier than an array because of the absence of repetition of elements. Easier to go through all of them. Its runs faster than an array.

How to make an empty set

const s = new *Set* ();

If we want to initialize a set with different elements in it, we can write an array inside with those elements. The result shows 4 elements because the set only contains unique elements and doesn’t reciprocate them. They also do contain an order. In this case 1,2,3,4 were the 1st four elements and appeared in that order. If we were to add a number after that, the element would appear after the 1st four elements in the new result.

*const* s = new *Set* ([1,2,3,4,4,3,2,1]);

s.add (-1);

console.log (s)

Results :

Set (4) {1,2,3,4}

Set (5) {1,2,3,4,-1}

If we want to remove an element, we’ll type delete and if we want to know the size of the set we’ll log the size.

*const* s = new *Set* ([1,2,3,4,4,3,2,1]);

s.add (-1);

s.delete (4);

console.log (s.size);

Results:

Set (4) {1,2,3,-1}

4

If we were to try and access an index, the result would be undefined because we don’t have access to indices with sets.

*const* s = new *Set* ([1,2,3,4,4,3,2,1]);

s.add (-1);

s.delete (4);

console.log (s[1]);

If we wanted to know the value of the element, we would need to iterate over them. We type this and we get each element individually iterated.

*const* s = new *Set* ([1,2,3,4,4,3,2,1]);

s.add (-1);

s.delete (4);

for (*const* number of s) {

console.log (number);

}

Result

1

2

3

-1

If we want to check if the element 1 exists, we’ll type this console.log (s.has (1));

The result will be a Boolean value, either true or false depending on if the set contains a 1. These types of operations are faster and easier to run with sets than arrays. In the example below, we’ll look through an array of numbers and we want to know if some of those numbers are duplicated elements. We have plenty of ways to do it but this way will use 2 for loops.

Here, the first line is asking the program to look through the array and to give us the index and value of all the elements. Then through all the values that we have, we need to look through the array a second time to see if that value is duplicated. The reason why we use .slice is because as soon as we checked if an element is duplicated or not, we don’t need to check that again. In this case, we’re saying idx+1, meaning we’re going to start by looking at the next element to the right in the array.

Then we’re gonna type a condition saying that if the value is equal to value 2 then we’ll log the found duplicate. The reason this will work is we’ll never check the same value in both of these for loops. If we didn’t have the +1 we would be checking the same value and this operation wouldn’t be possible.

*const* arr = [1,3,4,5,2,1,31,2];

for (*const*  [idx, value] of arr. entries ()) {

for (*const* value2 of arr.slice (idx +1)) {

if (value === value2) {

console.log (“Found duplicate of”, value);

break;

}

}

}

Result :

Found duplicate of 1

Found duplicate of 2

The reason why we don’t want to use this method is because it’s a very inefficient way of checking the code, especially if it’s very long. It’s a massive amount of operations.

To avoid doing all that, we’re gonna write a set under the array. then we’re gonna write a for loop and check if the current value we’re looking at exists inside of the array. if it does we’’ las for that value to be written inside of the set. If the value exists again, we’ll write the found duplicate. If the value doesn’t exist we’ll add the value to the set numbers.add(value);

*const* arr = [1,3,4,5,2,1,31,2];

*const* numbers = new *Set* ();

for (*const* value of arr) {

if (numbers.has (value)) {

console.log (“Found duplicate of”, value);

}

numbers.add (value);

}

Useful trick using a set:

If we want to use a set to determine all of the unique values in an array.

*const* arr = [1,3,4,5,2,1,31,2];

*const* numbers = new *Set* (arr);

console.log(numbers);

Result : Set (6) {1,3,4,5,2,31}

If we want to result to be given as a type array we’ll type

*const* arr = [1,3,4,5,2,1,31,2];

*const* numbers = new *Set* (arr);

*const* uniqueNumbers = *Array*.from (numbers);

console.log(uniqueNumbers);

Result : [1,3,4,5,2,31]

The result now appears inside a array which is useful because it gives us access to indices and other properties that arrays have.

**MAPS**

A map is a way to associate a key to a value. the value can be of different kinds, strings, number, Boolean etc. It allows us to be more flexible with our data and to have keys not solely associated with a number.

We’re going to create a map and set different values or different keys.

Here we declare the variable map. Then we set different values to a and b. the result will show that the latest value attribuated to a is gonna be the one that sticks.

*const* m = new *Map* ();

m.set (“a”, 3);

m.set (“b”, 2);

m.set (“a”, 4);

console.log (m)

Result :

Map (2) {‘a’ => 4, ‘b’ => 2}

If we want to access a specific key, we’ll type this last line, and the result will be 4

*const* m = new *Map* ();

m.set (“a”, 3);

m.set (“b”, 2);

m.set (“a”, 4);

console.log (m.get (“a”));

We also have the has function. console.log(m.has(“a”)); we don’t check the value when we have **has,** we only check the unique key. It’s also very useful and fast to add or remove keys with the map element. We can also get the size of the map, the number of pairings or mappings that we have.

*const* m = new *Map* ();

m.set (“a”, 3);

m.set (“b”, 2);

m.set (“a”, 4);

m.delete (“a”);

console.log (m.size);

Example of the use of the map

The most famous is keeping track of the frequency of characters from a string or array using a mapping type. Here we have a string. We want to get the frequency of the characters.

*const* str = “hello world my name is tim”

*const* freq = [];

for (*const* char of str) {

Let found = false;

for (*const* [idx, pair] of freq.entries ()) {

*const* [key, value] = pair;

if (key === char) {

freq [idx] = [char, value + 1];

found = true;

break;

}

}

if (!found) freq.push ([char, 1])

This would be what we write down if we were to use an array. It’s complicated and the result would show the frequency of all the characters so we would have to go through all of them.

Instead, we’ll use the map method

*const* str = “hello world my name is tim”;

*const* freq = new *Map* ();

for (*const* char of str) {

if (freq.has (char)) {

freq.set (char, freq.get (char) + 1)

} else {

freq.set (char, 1)

}

}

console.log(freq);

We would get this type of result.

‘m’=> 3,

‘y’ => 1,

‘n’ => 1,

‘a’ => 1,

‘i’ => 2,

‘s’ => 1,

‘t’ => 1

}

If we wanted to access a specific character, we would write this inside the console log. Here we want to know how many times the spacing character appears. It appears 5 times. So, the result will show 5 in the terminal.

console.log (freq.get (“ “));

If we wanted to delete something from the map we would write this and the character we want deleted inside the quotes freq.delete(“b”) if we want to delete b.

Freq.delete (“ “)

**OBJECTS**

An object is a way to store multiple properties in one variable or one reference. We can come up with our own custom objects that can store complicated data, store other objects, arrays, maps etc. JavaScript objects can get complicated, but they are used all over the place. The acronym JSON stands for JavaScript object notation.

*const* obj = {}

With the object we can have different properties that we assign after we’ve created the object or while creating it. A property is any kind of key or name inside of our object. The property name can’t have a space inside of it. We can have some specific characters like – or \_ inside of it but

*const* obj = {

name: “tim”,

};

console.log ();

Here we have the property name (name) and we assign to it a value, which is Tim. The name doesn’t need quotations marks and the same properties apply like a variable name. we can have a number inside the name, but it can’t start with a number.

*const* obj = {

name: “tim”,

};

console.log (obj);

If we log this in to access the property, this will be the result {name: ‘tim’}

If we use the .name we’ll have for result tim.

*const* obj = {

name: “tim”,

};

console.log (obj. name);

We can also access the property by typing this, and the result will be simply tim like the previous example.

*const* obj = {

name: “tim”,

};

console.log (obj [“name”]);

We can also assign properties and change them. Here we’ll change the name Tim to Joe. The result will show joe.

*const* obj = {

name: “tim”,

};

obj.name = “joe”;

console.log (obj [“name”]);

If we want to add a property, for example an age, we’ll write this.

*const* obj = {

name: “tim”,

};

obj.age = 13;

console.log (obj);

Result : {name: ‘tim’, age: 13}

We can also nest an object inside another object.

In this example, the object is a person named Tim. Tim has children. We have an array of children and inside the array we have an object with the name of the children. The purple brackets show the array and the blue brackets show the objects with the name inside of them.

*const* person = {

name: “tim”,

children: [{name: “joe”}, {name: “sally”}];

};

console.log (person);

Result : { name: ‘tim’, children: [ { name: ‘joe’ }, { name: ‘sally’ } ] }

If we want to access the first child, we will write this down and put the index 0

*const* person = {

name: “tim”,

children: [{name: “joe”}, {name: “sally”}];

};

console.log (person.children [0]);

Result : { name: ‘joe’ }

If we wanted the actual name, we would type console.log (person.children[0].name); and the result would be joe.

How to delete a name from the object. Here the name Tim is deleted

*const* person = {

name: “tim”,

children: [{name: “joe”}, {name: “sally”}];

};

delete person.name;

console.log (person);

Result : { children: [ { name: ‘joe’ }, { name: ‘sally’ } ] }

How to check if a property exists inside of an object

*const* person = {

name: “tim”,

children: [{name: “joe”}, {name: “sally”}];

};

console.log (“name” person);

Here we want to check if the name exists in the person object. Since it’s true, the result will show true. If we were to check if name1 existed, the result would be false, because we don’t have any name1.

How to loop through an object

*const* person = {

name: “tim”,

children: [{name: “joe”}, {name: “sally”}];

};

for (*const* property in person) {

console.log (property);

}

Result :

name

children

If I wanted to access the value of the property, I would write this below

*const* person = {

name: “tim”,

children: [{name: “joe”}, {name: “sally”}];

};

for (*const* property in person) {

console.log (person [property]);

}

Result

tim

[ { name: ‘joe’ }, { name: ‘sally’ } ]

The difference when we loop through an object is we will use the term in when we iterate, instead of of.

If we want to access the property names AND the value, we can write this down

*const* person = {

name: “tim”,

children: [{name: “joe”}, {name: “sally”}];

};

for (*const* [property, value] of *Object*.entries (person)) {

console.log (property, value);

}

Result :

name tim

children [ { name: ‘joe’ }, { name: ‘sally’ } ]

Object.entries gives us an array that contains arrays and inside of those subarrays, we’ll have the value associated with the property names.

If we only want the value of the object

*const* person = {

name: “tim”,

children: [ {name: “joe”}, {name: “sally”}];

};

for (*const*  value of *Object*.values (person)) {

console.log (value);

}

The result will show tim

The example below is another way of accessing the property name (instead of using the in)

*const* person = {

name: “tim”,

children: [ {name: “joe”}, {name: “sally”}];

};

for (*const*  value of *Object*.keys (person)) {

console.log (value);

}

How to merge different objects

Here we have two variables. P1 and p2. We make them equal to person. If we console.log, we get the exact same result for both.

*const* person = {

name: “tim”,

children: [ { name: “joe” }, { name: “sally” }];

};

*const*  p1 = person;

*const* p2 = person;

console.log (person, p1, p2);

Result : { name: ‘tim’, children: [ { name: ‘joe’ }, { name: ‘sally’ } ] } { name: ‘tim’, children: [ { name: ‘joe’ }, { name: ‘sally’ } ] } { name: ‘tim’, children: [ { name: ‘joe’ }, { name: ‘sally’ } ] }

If we got to p1 and delete the name, it will affects all of the objects. Why because there is only 1 object that we’re referencing from different variable names. This is known as mutability, meaning that we can alter the object once it’s been declared. In this case, when we declare the variable p1, we just assigned an additional name to person. Same as p2. This rule applies to any of the mutable types for example maps or sets. The additional name is known as an alias. The difference with a string for example is that if we were to do the same with a string, we would create a new string.

How to copy objects

*const* obj = {

name: “tim”,

};

*const* obj2 = obj;

console.log (obj);

console.log (obj2);

Result:

{ name: ‘tim’ }

{ name: ‘tim’ }

Any change to obj will affect obj2. If we don’t want that to happen, we can copy the contents of obj into a new object. To do that we write the obj index and in front of it we add the spread operator, the 3 dots. It copies all the properties from our object into this new object.

*const* obj = {

name: “tim”,

};

*const* obj2 = { … obj } ;

obj2.name = « red » ;

console.log (obj);

console.log (obj2);

Result:

{ name: ‘tim’ }

{ name: ‘red’ }

We can see here that it only affects the second object. With the spread operator we can also add new properties to the object. Here we added an age to the person.

*const* obj = {

name: “tim”,

};

*const* obj2 = { … obj, age: 2 } ;

console.log (obj);

console.log (obj2);

Result:

{ name: ‘tim’ }

{ name: ‘tim’, age: 2 }

Note: when you have properties that are mutable values, it can alter your object. Here we added the array with 1,2,3 and logged it in.

*const* obj = {

name: “tim”,

nums: [ 1, 2, 3 ],

};

*const* obj2 = { … obj, age: 2 } ;

console.log (obj);

console.log (obj2);

Result :

{ name: ‘tim’, nums [1, 2, 3 ] }

{ name: ‘tim’, nums [1, 2, 3 ], age: 2 }

But if we type in that the index is equal to 100, we change the result in both of the objects. The array is a mutable variable.

*const* obj = {

name: “tim”,

nums: [ 1, 2, 3 ],

};

*const* obj2 = { … obj, age: 2 } ;

obj.nums [0] = 100;

console.log (obj);

console.log (obj2);

Result :

{ name: ‘tim’, nums [100, 2, 3 ] }

{ name: ‘tim’, nums [100, 2, 3 ], age: 2 }

**ERROR HANDLING**

In this context an error is something that happens at runtime and causes the program to crash.

Syntax error and using a method that doesn’t exist. In JavaScript we can throw our own errors which is known as throwing or raising errors and we can also run these errors to make sure that the code doesn't end up like this and crash

*const* x = 2;

x.toUpperCase ();

Reason : x.toUpperCase is not a function

To handle the error, we’re going to take the code that contains the error and wrap it into a try block. By also adding a catch, it essentially allows us to enter this block if an error occurs inside the previous block, which is the try block. So, with the try/block method, we try to catch an error in the try block first; if the try block doesn’t fix the error, the catch block should.

*const* x = 2;

try {

x.toUpperCase ();

} catch {

console.log (“recovery”);

}

Result : recovery

Here it printed recovery because what happened in the try block was unsuccessful. As soon as an error or an exception is raised in a try block, we’re going to immediately go into catch.

*const* x = 2;

try {

console.log (“try”);

x.toUpperCase ();

console.log (“success”);

} catch {

console.log (“recovery”);

}

Result :

try

recovery

Here we have try and recovery printed. Why ? because in the first console.log, everything is fine. The error is located after that. The error being x.toUpperCase();

We then have recovery printed because since the second console.log contains the error and didn’t work, we went right to the catch block. That block doesn’t contain any errors, so it prints normally.

To understand what kind of error we get, we can type this, and the terminal will tell us what it is.

*const* x = 2;

try {

console.log (“try”);

x.toUpperCase ();

console.log (“success”);

} catch (err) {

console.log (err.message, err.name);

}

Result:

try

x.toUpperCase is not a function TypeError

Err stands for error and we want two things. We want a message explaining to us what the error was, and we want the name of the error, which is TypeError

Another option is to use finally. Finally allows us to have some operation that happens no matter what. Whether the try/catch options were successful or not, it doesn’t matter because the finally block will make it happen.

*const* x = 2;

try {

console.log (“try”);

x.toUpperCase ();

console.log (“success”);

} catch (err) {

console.log (err.message, err.name);

} finally {

console.log (« finally ») ;

}

Result :

try

finally

How to throw your own error

It’s not typically something you would want to do. This may be useful to create our own code, program and we want to tell the user that whatever action he just did, it’s not correct. We would have to type the details of why it’s an error.

To do so, we write throw inside the code and the next line appears blurred out because it knows we’re never going to enter it. We then delete it that blurred out line.

*const* x = 2;

try {

console.log (“try”);

throw “this is an error” ;

console.log (“success”);

} catch (err) {

console.log (err.message, err.name);

} finally {

console.log (« »finally ») ;

}

Result : undefined. Because the error object is now just a string.

*const* x = 2;

try {

console.log (“try”);

throw “this is an error” ;

} catch (err) {

console.log (err.message, err.name);

} finally {

console.log («finally ») ;

}

Result :

try

undefined undefined

finally

If we change the way we console.log, we can get the result we were looking for.

*const* x = 2;

try {

console.log (“try”);

throw “this is an error” ;

} catch (err) {

console.log (err);

} finally {

console.log («finally ») ;

}

Result:

try

this is an error

finally

We can throw a string, a number, anything we want but what happens is JavaScript is going to treat it like an error, so it will crash the program if we don’t handle it and it will pass that error object in something like a catch block where we can handle it there.

If we wanted to throw the error in a way that it’s an object, we can change the syntax to make it an object

*const* x = 2;

try {

console.log (“try”);

throw { name: “error”} ;

} catch (err) {

console.log (err.name);

} finally {

console.log («finally ») ;

}

Result:

try

error

finally

We can also write error which is a reserved key word in JavaScript and place inside the brackets what the error message is. The result is the text and right next to it, the name of the error.

*const* x = 2;

try {

console.log (“try”);

throw *Error* (name: “this is an error”) ;

} catch (err) {

console.log (err.message, err.name);

} finally {

console.log («finally ») ;

}

Result:

try

this is an error Error

finally

Note : There are other ways to write errors.

If we want to throw our own errors, it’s best to write the error with the reserved key word in JavaScript.

**FUNCTIONS**

A function is a reusable block of code that we can call multiple times and use throughout our program. To make a function, we have a designated function term and we’re going to give the function a name. The same rules apply for the syntax as the rules for variables. So, we can have numbers but not at the beginning of the word. We can have – or \_ etc

Here the name of the function is add. Anything inside of the curly braces is going to be executed when we call this function.

*function* add () {

}

To use the function, we’re going to write the function and then a set of parentheses. It’s important that we write the parenthesis because it tells the program that we want to execute the function. If we don’t write the parenthesis, it will only reference the function. So when we call the function add(); we will enter the block of code and start executing it.

*function* add () {

console.log(“add”);

}

add ();

Result : add

If we call the function again, it will show up again in the result.

*function* add () {

console.log(“add”);

}

add ();

add ();

Result:

add

add

The function allows us to repeat a code multiple times without having to use the for loop. The function is also useful to organize a code and to split different logics into separated logical code blocks. That way it’s easier to understand what’s going on and we have different sets of codes separated inside their respective functions.

Note: The difference between JavaScript and other programming languages is that we can call a function before defining it and it works.

add ();

*function* add () {

console.log(“add”);

}

Besides the organization aspect, the function allows us to do other things. We can accept what we call parameters. The parameters are inputs to our function that the function will run on. In this case we have an add function and we can specify that we want to have the inputs x and y, known as parameters and now when we call our add function, we can pass values for x and y and use that inside of our function.

*function* add (x, y) {

console.log(x + y );

}

add (10, 8);

Result : 18

The parameters follow the same syntax rules as the variables.

When we pass values to the functions add (10, 8); to substitute for the parameters, those values are known as arguments. In this case, 10 and 8 are arguments.

Note: sometimes people mess up the names and call x and y the arguments, instead of parameters.

We can pass strings and arrays as well as numbers.

Different types of parameters

Here we have a function called math that contains two values a1 and a2 and another value that is optional, so someone could pass but doesn’t need to pass. If we want num to be optional it to set it equal to a value. If we don’t pass a value for num, 1 will become the default value.

*function* math (*a1*, *a2*, *num* = 1) {

console.log(*a1* \* *a2* \* *num*);

}

math (10, 20);

math (-10, 10, -1);

Result:

200

100

Sometimes we don’t want to print the results to get them. We want to get the results where we print the function. If we have a complicated operation inside of the function and want to capture the results then do something with those results outside of the function, we’ll introduce the return statement. Return allows us to take some value and return to whoever called the function.

*function* math (*a1*, *a2*, *num* = 1) {

return *a1* \* *a2* \* *num*;

}

*const* result = math (10, 20);

console.log (result);

Result: 200

To return multiple values from our function we will use square brackets and use the array syntax.

*function* math (*a1*, *a2*, *num* = 1) {

return [*a1* \*\*2,  *a2* \*\*2];

}

*const* result = math (10, 20);

console.log (result);

Result:

[100, 400]

The result here is equal to an array since we had the return inside square brackets.

To get the individual values we could change the syntax for result and write this.

*function* math (*a1*, *a2*, *num* = 1) {

return [*a1* \*\*2,  *a2* \*\*2];

}

*const* [result, result2] = math (10, 20);

console.log (result, result2);

Result: 100 400

If we were to remove the brackets for the multiple values, we would get an error in the terminal.

Examples of functions

In this case we want to have a function that is going to take every single value inside of an array and perform some operation on it. We have multiple strings, and we want to remove a specific character. The name of the function here is removeAllChars.

*function* removeAllChars (*arrayOfStrings*, *charactersToRemove*) {

[“hello”, “hello”] [“a”, “b”]

}

For our parameters we will need our *arrayOfStrings* so we’ll write that inside the parenthesis, followed by the characters we want removed. So, for example, we have our array of strings “hello”, “hello” then we have the characters “a” and “b” that we want removed from the strings. (The hello strings and a, b aren’t going to be the thigs used in the following example)

To do that we’re going to need to create a new array that we’re going to return with all our new strings. We could modify the array if we wanted to though.

Here we’re going to request a new array to be created and we will loop through all the strings and for every single string we will remove the characters.

We’re going to write let newStr then we’ll loop though every single character that we have and compare it to every single character inside of the string and if the string doesn’t contain that character, we will add that character to the new string.

*function* removeAllChars (*arrayOfStrings*, *charactersToRemove*) {

*const* newArray = [ ];

for (*const* str of *arrayOfStrings*) {

*let* newStr = “ “ ;

for (*const* char of str) {

if (*charactersToRemove*.includes (char) ) {

continue;

}

newStr += char;

}

newArray.push (newStr);

}

return newArray;

}

*const* result = removeAllChars (

[“hello world”, “1234”, “I am the best”, “yes”],

[“e”, “2”, “t”]

Result :

[‘hllo world’; ‘134’, ‘I am h bs’, ‘ys’ ]

Summary:

To define a function we use the function keyword *function*

We give a name to the function removeAllChars. The name should be descriptive as to what the function does

In parenthesis we have different parameters. Sometimes the brackets can be empty. *arrayOfStrings, charactersToRemove*. The parameters will have the name of what they are. We can also have default or optional parameters.

Everything inside the yellow curly braces is the block of code, or the function body and this is what’s going to be executed when we call the function. We will run the function and return the new Array. When we have a return statement, this is immediately going to exit the function and return the value to whoever called it. So, if the return statement was located right after the if statement for example (instead of continue), we would return directly to whoever called it. The return statement means immediate exit.

We can use the return statement to exit the function early.

In this example, we don’t have any characters to remove. We write an if statement to say that if the length of the characters to remove is equal to 0, we will return, meaning we exit the function right away. Anything written under that line will be ignored.

*function* removeAllChars (*arrayOfStrings*, *charactersToRemove*) {

*const* newArray = [ ];

if (*charactersToRemove*.length === 0) return;

for (*const* str of *arrayOfStrings*) {

*let* newStr = “ “ ;

for (*const* char of str) {

if (*charactersToRemove*.includes (char) ) {

continue;

}

newStr += char;

}

newArray.push (newStr);

}

return newArray;

}

If we do this, by default we’re going to have a undefined result, which is called a naked return statement, it means we want to exit the function at an early point.

Note: if we don’t pass the required parameters (aka we didn’t specify which characters to remove), we’re also going to have an undefined result.

**EVENTS & THIS**

An event is something that can occur in our html documents. It can be something as simple as pressing a button, changing an input field, moving the mouse around. We can listen to these html events and trigger some type of corresponding JavaScript function.

Example

In this case we create a form in our html document. We give it an id and instead of using the submit button, we will use the button tag. We’ll also use the onclick function, meaning that when the user clicks on the button, it will trigger some reaction into the JavaScript documents. Inside the quotes, we will specify the name of the JavaScript function that we want to call when this button is clicked.

<!DOCTYPE html>

<html lang=”en”>

<head>

<title>Document </title>

</head>

<body>

<form>

<input type= “text” id=”username” placeholder=”username” />

<button onclick=”enterUsername ()”>Enter</button>

</form>

<script src=”test.js”></script>

</body>

</html>

We’re going to copy the name that we entered after onclick, in this case enterUsername() and we’ll define the function into JavaScript

*function* enterUsername () {

*const* username = document.getElementById (“username”).value;

alert (“You entered” + username);

}

In the following example, we got rid of the form format and the button. Instead, we used onchange. onchange is an event that will run every single time the text field is changed. For example, as we type in the word hello into the text field, the change appears simultaneously.

<!DOCTYPE html>

<html lang=”en”>

<head>

<title>Document </title>

</head>

<body>

<input

type= “text”

id=”username”

placeholder=”username”

onchange= “enterUsername ()”

/>

<p id= “text”></p>

<script src=”test.js”></script>

</body>

</html>

**THIS**

Inside a function, we have a keyword called this. This will access different values depending on the context in which it’s used in. it will going to refer to whatever called this function. So as we write this inside the JavaScript document, this refers to the html code that called this function.

Note: in the case of onchange, the live server window is what is going to be responsible for calling this function. The window is responsible for calling the function enterUsername() when onchange occurs

onchange= “enterUsername ()”

When the input event occurs, it calls for the onchange function.

Youtube video link

[What is THIS in JavaScript? 👈 - YouTube](https://www.youtube.com/watch?v=Jdlo8ZDt5Jg)

// this = reference to the object where THIS is used (the object depends on the immediate context)

// person.name = this.name

*const* person1 = {

name: “Spongebob”,

favFood: “hamburgers”,

sayHello: *function* () {console.log (‘Hi I am ${this.favFood}’)},

eat: *function* () {console.log (‘${this.name} is eating ${this.favFood}’)},

}

*const* person2 = {

name: “Patrick”,

favFood: “pizza”,

sayHello: *function* () {console.log (‘Hi I am ${this.favFood}’)},

eat: *function* () {console.log (‘${this.name} is eating ${this.favFood}’)},

}

person1.eat ();

person2.eat ()

Result :

Spongebob is eating hamburgers

Patrick is eating pizza

What we have here is two objects. We have an object 1 which is person 1 and an object 2 which is person 2. THIS is simply replacing person1 and person2 when we actually write in the code. Example with person 2 below. THIS is referencing to the object that it’s located into, in this case person2.

*const* person2 = {

name: “Patrick”,

favFood: “pizza”,

sayHello: *function* () {console.log (‘Hi I am ${this.favFood}’)},

eat: *function* () {console.log (‘${this.name} is eating ${this.favFood}’)},

}

person1.eat ();

person2.eat ()

If we just console.log (this); we will have window as a result. Note: if we wanted to reference a function we could write :

*const* person = {

name: “Tim”,

lastName: “Ruscica”,

getName: *function* () {

console.log (this.name + this.lastName);

},

};

person.getName;

The last line of code is what we wrote to reference the function. If we wanted to call the function we would add the double set of parenthesis behind it. person.getName();

If we have the THIS keyword inside of a standard function (not an object), THIS is going to access the global object.

*function* test () {

console.log (this);

}

test ();

**ARROW FUNCTIONS**

Special type of function in JavaScript. They have similarities to the other functions but have a different type of style or notation than the standard function.

*const* func = () => {

console.log (“hello”);

};

func ();

The result we get is hello

Similarities with other functions

* Can accept parameters

Just like standard functions, arrow functions can accept parameters. Here the result is 3

*const* func = (*x*, *y* ) => {

console.log (*x* + *y*);

};

func (1, 2);

* Can return values

*const* func = (*x*, *y* ) => {

return *x* + *y* ;

};

*const* result = func (1, 2);

console.log (result);

The result is 3

* Can have default parameters

*const* func = (*x*, *y* = 2) => {

return *x* + *y* ;

};

*const* result = func (1, 2);

console.log (result);

The only difference with the regular functions is that we have some shorthand syntax for returning values. We can actually write x + y right after the arrow and since it’s the only expression that’s here, by default it’s going to return the value without us having to write the return keyword

*const* func = (*x*, *y*) => *x* + *y*

*const* result = func (1, 2);

console.log (result);

Note: if we were to write the curly braces for the function value after the arrow, the result would be undefined. Why? Because x+y aren’t right after the arrow anymore but they are inside the curly braces.

*const* func = (*x*, *y*) => {

*x* + *y*;

}

*const* result = func (1, 2);

console.log (result);

If we were to write the curly braces, we would have to write the return keyword for the operation to work.

* Can return an object

We can return an object using this syntax. Because we have parenthesis after the arrow, it’s indicating that whatever is inside of the parenthesis, we want to return. In this case it’s the object {x: *x*, y: *y* }

*const* func = (*x*, *y*) => ( { x: *x*, y: *y* } )

*const* result = func (1, 2);

console.log (result);

Result : { x: 1, y: 2}

If we were to remove the parenthesis, we would get an error code, because it’s not a valid syntax.

Differences with standard functions

*function* getName () {

return this.firstName + “ “ + this.lastName;

}

*const* person = {

firstName: “Tim”,

lastName: “Ruscica”,

getName: getName,

};

*const* personName = person.getName ();

console.log (personName);

Result : Tim Ruscica

On line 11, the person from person.getName() is what is calling the function .getName()

Even though the function is defined at the very top and NOT directly inside of the object (const person block), since the keyword (line 1-3) accesses what’s calling it, it’s still getting access to this object’s properties (line 6-7).

If we were to copy paste the object and create a person2, give them a different name, if we use person2 when we call the function, we would get the result Joe Smith because we are accessing the context in which the function was called

*function* getName () {

return this.firstName + “ “ + this.lastName;

}

*const* person = {

firstName: “Tim”,

lastName: “Ruscica”,

getName: getName,

};

*const* person2 = {

firstName: “Joe”,

lastName: “Smith”,

getName: getName,

};

*const* personName = person2.getName ();

console.log (personName);

If we were to only write getName on the second to last line, we would get the results undefined

*const* personName = getName ();

console.log (personName);

Why? Because it is now accessing the global context which doesn’t have a first or last name property.

All of this changes when we use an arrow function.

*const* getName = () => {

return this.firstName + “ “ + this.lastName;

};

*const* person = {

firstName: “Tim”,

lastName: “Ruscica”,

getName: getName,

};

*const* person2 = {

firstName: “Joe”,

lastName: “Smith”,

getName: getName,

};

*const* personName = person.getName ();

console.log (personName);

The result here is going to be undefined undefined. The reason is that the arrow function has THIS keyword always bound to where this function was defined not where it was called from. In this case we defined it in the global space.

**MAP, FILTER, REDUCE**

Map function

We have an array and a lot of numeric values in this array. If we want to get the squares of these values, we can write a function that does that or we can use the map function. What we do with map, is pass what’s known as a callback which is going to accept a parameter and then return a value. In this case we can use an arrow function, which is the callback. For the callback we need to have a parameter. This parameter is going to be each individual element in my array. In this case we’ll use x. We then return x to the exponent 2.

We are saying here that we want to map every single value of my array to this function.

( ( *x* ) => *x* \*\* 2)

In other words, we pass 1, that returns 1. We pass 2, that returns 4, we pass 3, that returns 9 etc. all these returned values get added to a new array which is what the squares array is.

*const* arr = [1, 2, 3, 4, 5, 6];

*const* squares = arr.map ( ( *x* ) => *x* \*\* 2) ;

console.log (squares);

Result : [1, 4, 9, 16, 25, 36]

It’s a new array so it doesn’t affect the original array.

If we wanted to write a map function on our own. We can use an array function or a standard function. In this case we’ll use a standard function that we’ll call map. Inside of map we’ll take an array and a function that we want to apply all the values to this array for.

Note: it’s a little bit different than when we call .map on the array because in this case we are not associating this with an object, we are just saying we’ll take any array and any function to apply to all of the elements in the array.

*const* arr = [1, 2, 3, 4, 5, 6];

*function* map (*arr, func*) {

*const* newArr = [ ];

for (*const* value of *arr*) {

*const* newValue = func (value);

newArr.push (newValue);

}

return newArr;

}

*const* result = map (arr, ( *x* ) => *x* \*\* 2) ;

console.log (result);

We have a function here, meaning we can call the function and pass a value to it. we’re saying we’ll make a new array, we’ll loop through all of the values in the original array, we’ll now calculate a new value by passing this value to our function and we’ll take whatever that value is and push that into our new array.

We get the same exact result as the previous method we used

Filter function

It will only keep values inside of an array if they pass the filter.

In this case we pass a callback function again. This callback function is going to accept a value and then we need to return true or false indicating whether we want to keep this value in the array. Here by using mod % we say we want to keep the values inside of an array only if they are divisible by 2.

*const* arr = [1, 2, 3, 4, 5, 6];

*const* filteredArr = arr.filter ( (*value*) => *value* % 2 == 0);

console.log (filteredArr);

It’s going to return a new array that contains all of the elements from the original array, in their original order that passed the filter, meaning if the function returns true or a value that evaluates to true for these elements. The result we get is [2, 4, 6]

If we wanted to build filter from scratch:

*const* arr = [1, 2, 3, 4, 5, 6];

*function* filter (*arr, func*) {

*const* newArr = [ ];

for (*const* value of *arr*) {

if ( func (value) ) {

newArr.push (value);

}

}

return newArr;

}

*const* result = filter (arr, ( *x* ) => *x* % 2 = = 0) ;

console.log (result);

We get the same result [2,4,6]

Reduce function

The reduce function will reduce an array or iterable object into something that is a single value. There are a lot of different ways we can use the reduce function. It allows us to keep track of something like a sum or a count or some other value as we iterate though this array.

Example

*const* arr = [1, 2, 3, 4, 5, 6];

*const* sum = arr.reduce ( (*prev*, *current* ) = > *prev* + *current* );

console.log (sum);

The result here is 21, so the sum of the array.

|  |  |
| --- | --- |
| *const* arr = [1, 2, 3, 4, 5, 6];  *const* sum = arr.reduce ( (*prev*, *current* ) = > *prev* + *current* );  console.log (sum);  *const* arr = [1, 2, 3, 4, 5, 6];  *const* sum = arr.reduce ( (*prev*, *current* ) = > *prev* + *current* );  console.log (sum);  *const* arr = [1, 2, 3, 4, 5, 6];  *const* sum = arr.reduce ( (*prev*, *current* ) = > *prev* + *current* );  console.log (sum); | The previous value is equal to  the previous value returned  by this function  This function is called with a new previous value the number of elements times in our array. |

Meaning previous starts out at 0, current starts out at first element which is 1

(0, 1) => 0 + 1 1

(1, 2) => 1 + 2 3

(3, 3) => 3 + 3 6

(6, 4) => 6 + 4 10

(10, 5) => 10 + 5 15

(15, 6) => 15 + 6 21

So, we are reducing the values inside of this array to 1 single value.

Here we are saying we’re going to start at 0

*const* arr = [1, 2, 3, 4, 5, 6];

*function* reduce (*arr, func*) {

*let* currentResult = 0 ;

for (*const* num of *arr*) {

currentResult = func (currentResult, num);

}

return currentResult;

}

We then loop through all of the numbers in the original array

*const* arr = [1, 2, 3, 4, 5, 6];

*function* reduce (*arr, func*) {

*let* currentResult = 0 ;

for (*const* num of *arr*) {

currentResult = func (currentResult, num);

}

return currentResult;

}

We then say the current result is equal to the function with whatever the currentResult is (aka the previous value) + the current number

*const* arr = [1, 2, 3, 4, 5, 6];

*function* reduce (*arr, func*) {

*let* currentResult = 0 ;

for (*const* num of *arr*) {

currentResult = func (currentResult, num);

}

return currentResult;

}

Meaning we’ll pass the previous and we’ll pass the next which is what we are currently on in our array. Then we’ll continue to loop through for every single value in our array and currentResult should increase or decrease or reduce to whatever value the function defines it should.

Example

Here we are essentially taking whatever previous value (starts at 0) and we are subtracting the next value every single time

*const* arr = [1, 2, 3, 4, 5, 6];

*function* reduce (*arr, func*) {

*let* currentResult = 0 ;

for (*const* num of *arr*) {

currentResult = func (currentResult, num);

}

return currentResult;

}

*const* result = reduce (arr, (prev, next) => *prev* - *next* );

So we’ll have

(0, 1) => 0 - 1 -1

(-1, 2) => -1 - 2 -3

(-3, 3) => -3 - 3 -6

(-6, 4) => -6 - 4 -10

(-10, 5) => -10 - 5 -15

(-15, 6) => -15 - 6 -21

**SCOPE**

Knowing everything we’ve learned so far, it’s important to know what happens *inside* of different types of blocks of code. What happens when we declare a variable inside of a function? What is the scope of that variable? Where is it accessible? Is it accessible inside the function? Outside of the function? Etc.

**The global scope**

It’s not a function, it’s not an if statement and we can access it anywhere from our program.

We can access the variable right here

*Let* x = 2;

console.log (x)

We can also access it from a function. The result here will be 4.

*let* x = 2;

*function* accessX ( ) {

x = 4 ;

}

accessX ( ) ;

console.log (x) ;

Now if we were to move let x = 2 inside the function, we would get an error result

*function* accessX ( ) {

*let* x = 4 ;

}

accessX ( ) ;

console.log (x) ;

It says that x is not defined because the scope of this variable is this function. It’s not the global scope which is where console.log is looking. So, we cannot access the x variable.

When we define a variable inside of a function, the scope of the variable is the function.

We can define a function inside of another function. Since that sub function is inside the original function it allows us to access the variable x

*function* accessX ( ) {

*let* x = 4 ;

*function* test ( ) {

console.log (x);

}

test ( ) ;

}

accessX () ;

If I call the test function inside of the function (line 8, where it’s located currrently), it works and we get the result 4.

If we were to call the test function outside of the function (line 11), it would tell us test is not defined.

*function* accessX ( ) {

*let* x = 4 ;

*function* test ( ) {

console.log (x);

}

test ( ) ;

}

test () ;

Examples

Difference between the *let* and the *var* keyword.

*function* test (x) {

if ( *x* === 3) {

*let* val = 5;

} else if ( *x* === 4) {

*let* val = 6;

} else {

*let* val = 0;

}

console.log (val)

}

test (3);

If we call test for 3, we should get the result 5. We get an error saying that val is not defined. Why? Because when we use the *let* keyword it’s only defining the variable in the scope of the block in which it’s defined. Not just the function but the block.

In this case, val is located inside of the if statement so its scope is limited to this selected area

if ( *x* === 3) {

*let* val = 5;

}

If we were to put val higher up, we could access it because it’s not located inside the if statement, and it is part of the entire block of the function (including the if/else statements). So, anything inside of the if statements can access it as well.

*function* test (x) {

*let* val;

if ( *x* === 3) {

*let* val = 5;

} else if ( *x* === 4) {

*let* val = 6;

} else {

*let* val = 0;

}

console.log (val)

}

test (3);

If we were to change the *let* keyword with *var*, it IS accessible. Why ? the var keyword defines a variable within the most outerscope that it can. In this case, it’s the test function.

*Let* is going to respond that whatever block of code it’s defined in*. var* is going to respond to whatever function it’s located into, even if it’s written inside a bunch of if statements, it’s still going to find that function located above. The result here will be 5

*function* test (x) {

if ( *x* === 3) {

*var* val = 5;

} else if ( *x* === 4) {

*var* val = 6;

} else {

*var* val = 0;

}

console.log (val)

}

test (3);

If we were to type this below, we would run into an error. Why? Because *var* is located inside the function on line 2. So, it will respond to that function and not the function on line 1, which is the one defining test (x)

*function* test (x) {

*function* test2 ( ) {

if ( *x* === 3) {

*var* val = 5;

} else if ( *x* === 4) {

*var* val = 6;

} else {

*var* val = 0;

}

}

test2 ( );

console.log (val)

}

test (3);

If we were to use *const*, it would work the same as *let*.

*function* test (x) {

if ( *x* === 3) {

*const* val = 5;

} else if ( *x* === 4) {

*const* val = 6;

} else {

*const* val = 0;

}

console.log (val)

}

test (3);

Example

*const* x = 5;

*function* test (x) {

if (x ===3) {

console.log (“one”);

} else if (x ===4) {

console.log (“two”);

} else {

console.log (“three”);

}

}

test (3);

The question here is what x are we going to be referencing. Is it going to be x from line 1 (*const* x= 5) or from line 3 (*function* text (x) ) . The result will show one. Why?

We’re referencing the x from line 3. The if statement is telling us that if x is strictly equal to 3, the result will be 1.

The reason why we used that x and not the one from line 1 is because we are going to look at the variable that is closest in the current scope. In this case x is the closest variable to us. It’s defined in the scope of the function whereas the x from line 1 is global.

If x didn’t exist in the line 3, we would use the x from line 1. The result is equal to three in that case. Why? Because we declared that x is equal to 5

Knowing that fact, x is strictly not equal to 3, and x is strictly not equal to 4. We then have to execute the last else statement.

Quiz example

*let* x = 10;

*function* change ( ) {

x = 20;

console.log (x);

}

changeX ( );

console.log (x);

The answer is

20

20

If we were to take out changeX () we would get a single answer as 10. The console.log here are executing what is directly above them. The first one inside the function, the second one at the very end. Whether we have x declared as a variable at the very top doesn’t matter let x = 10 so if we were to take it out we would still get the results 20 20. Because the console.log are directly executing the line right before them.

**MUTABILITY**

There are a lot of immutable types in JavaScript . What that means is once we declare them, they cannot change their value. As opposed to arrays, maps, sets which are mutable.

true

3

“str”

undefined

null

Example

*let* x = 10;

*const* y = x;

x = 20;

console.log (x, y);

The result here will be 20 10

X doesn’t change when I make a change to x. The reason is that the number type 10 is immutable.

When we say const y = x, we say store whatever x is storing, in this case it’s 10. Since it’s an immutable type, by default we make a copy of 10. We don’t store a reference to the same instance, we don’t store a reference to the same 10. We are actually making a new 10 and putting that inside of y.

*let* x = 10;

*const* y = x; // copy of 10

x = 20;

console.log (x, y);

Now y is equal to 10 and when we go and say x=20, there is no link between y and x. Us changing x doesn’t affect the copy of them we store inside of y.

This is what happens when we use immutable values.

Example of an array

*let* x = [1, 2, 3, 4, 5];

*let* y = x;

x [0] = 100;

console.log (x, y);

Result : [ 100, 2, 3, 4, 5 ] [ 100, 2, 3, 4, 5 ]

This affects both x and y. In this example, what happens is when we say let x = [1,2,3,4,5], we are creating a new array, and we are storing a reference to the array inside of x

X is storing a reference, and this reference points to the array [1,2,3,4,5]. So, whenever we access x, it’s accessing this underlying array but it’s first doing that going through the reference. When we store mutable types, we’re storing a reference to that type.

When we say let y = x, we are copying the reference not the actual array into y.

*let* x = [1, 2, 3, 4, 5]; // x -> ref -> [1, 2, 3, 4, 5]

*let* y = x; // y -> ref -> [1, 2, 3, 4, 5]

x [0] = 100;

console.log (x, y);

If we are modifying x or y, we are modifying the underlaying array that they’re both pointing to. This happens because they are mutable types.

What happens when we reassign x

*let* x = [1, 2, 3, 4, 5]; // x -> ref -> [1, 2, 3, 4, 5]

*let* y = x;

x = [1];

console.log (x, y);

Result : [1] [1, 2, 3, 4, 5]

Here something different happens.

x has changed but y is still assigned to the previous value. When we assign a new array to x, we changed the reference that x is storing but we don’t modify the underlying array.

If we were to type x[0] =1, we are using the reference to access the underlaying array and we’re changing it.

*let* x = [1, 2, 3, 4, 5]; // x -> ref -> [1, 2, 3, 4, 5]

*let* y = x;

x [0] = 1

console.log (x, y);

If we say x =[1], it’s going to be a brand new array

*let* x = [1, 2, 3, 4, 5]; // x -> ref -> [1, 2, 3, 4, 5]

*let* y = x;

x = [1] // x -> new ref -> [1]

console.log (x, y);

Let y = x is also known as alias because it’s another name of referencing an object.

If we want to avoid this below

*let* x = [1, 2, 3, 4, 5]; // x -> ref -> [1, 2, 3, 4, 5]

*let* y = x;

y = [1]; // y -> new ref -> [1]

console.log (x, y);

Result : [1, 2, 3, 4, 5] [1]

We need to make a copy of that object. There are multiple ways of doing this. The most common way to do so is to use the spread operator. The spread operator will take all the values from the original array and put them inside of a new array. As a result, when y is modified, x won’t be.

*let* x = [1, 2, 3, 4, 5]; // x -> ref -> [1, 2, 3, 4, 5]

*let* y = [ …x] ;

y [0] = 100;

console.log (x, y);

Result : [1, 2, 3, 4, 5] [100, 2, 3, 4, 5]

**MATH**

**How to do rounding in JS**

A lot of times we’ll have some kind of decimal value, and we’ll need to round up or down to the nearest integer.

**First method: rounding to the closest integer**

*const* num = 2.55;

*const* roundedNum = Math.round (num);

console.log (roundedNum);

Note : When we want to do any type of Math operation, we write Math. and the program will suggest different possibilities.

How to round to a certain decimal place

There isn’t a way to ask the program to stop after 1, 2, 3 decimals. So we have to resort to multiplying the number then dividing it by that same number we multiplied it with. If we multiply by 10, we’ll divide by 10. The result here will be 2.6

*const* num = 2.55;

*const* roundedNum = Math.round (num \* 10 ) / 10;

console.log (roundedNum);

Other example. The result here will be 2.557

*const* num = 2.557262;

*const* roundedNum = Math.round (num \* 1000 ) / 1000;

console.log (roundedNum);

**Second method : round down/up no matter what**

Math.floor will always round down so it will strip the decimal off

Math.ceil will round up

**How to get the maximum or minimum value**

*const* x = 10;

*const* y = 20;

*const* max = Math.max (x, y) ;

Console.log (max**) ;**

Here we’re going to ask the program to tell us what the maximum value is. The answer will be 20 since 20 is bigger than 10. If we were to ask what the minimum value is, we would get 10. For this kind of operation, we can have as many values as we want inside the braces.

If we had an array, we would need to use the spread operator when we pass this function, otherwise the result would be NaN. The result here is 5.

*const* arr = [1, 3, 4, 5, 2, 1, 2];

*const* min = Math.max (…arr) ;

console.log (min) ;

**How to get the absolute value of a number**

This will get rid of whatever sign is in front of the number so in this case, we’ll get 9

*const* abs = Math.abs (-9);

console.log (abs);

The Math. function can also be used for logarithms, cosign, tangent etc

The concept of infinity exists in JS. Sometimes when you’re writing some kind of algorithms, you do need to have some number be like an infinite value so that every number you’re comparing it to, will be less than that. Useful to know what the biggest possible set of numbers is when we work on JS.

*const* x = Infinity

If we want to know what the smallest possible set of numbers in JS is, we would place a minus in front of infinity.

*const* x = -Infinity

**RANDOM**

How to generate random numbers

*const* num = Math.random ( );

console.log (num);

Result : 0.8802253634108337

This function is used to specify what kind of random number we’re going to get. Once we get our random number, we can round it, multiply it, etc to get a number that’s within the range that we’re trying to generate.

Example

To generate a random number between 1 and 10.

*const* num = Math.round (Math.random ( ) \* 10);

console.log (num);

If we run this a few times, the result will be different each time, because it’s random. The only thing in common is these numbers will all be between 1 and 10.

To get a number between 10 and 20 we’ll write this. The minimum value will be 10.

*const* num = 10 + Math.round (Math.random ( ) \* 10);

console.log (num);

How to write a function that will generate between a max and a min

*function* randomNum (*min*, *max*) {

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

}

*const* num = randomNum (20, 80);

console.log (num);

We write Math.floor because we want to exclude the max number. If we didn’t plan on excluding it, we could write Math.round

+min will give us the absolute minimum.

*function* randomNum (*min*, *max*) {

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

}

*const* num = randomNum (20, 80);

console.log (num);

This whole expression can never be greater than whatever max minus min is.

How to write a program that allows the user to enter a bunch of numbers and guess which number we have said. We’ll generate a random number and have the user guess it. We’ll see how many tries it takes them to guess.

<!DOCTYPE html>

<html lang = “eng”>

<head>

<title>Document </title>

</head>

<body>

<p> Try to guess a random number between 1 and 100. </p>

<label for = “guess”> Guess : </label>

<input id = “guess” type = “number”/>

<button onclick= “MakeGuess ()”>Make Guess </button>

<p> Number of guesses: <span id = “num-guesses” >0 </span> </p>

<p id = “feedback”> </p>

<script src =”test.js”> </script>

</body>

</html>

JS document

*function* randomNum (*min*, *max*) {

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

}

*let* num = randomNum (1, 100);

*let* numGuesses = 0;

*function* reset () {

num = randomNum (1, 100);

numGuesses = 0;

document.getElementById (“num-guesses”).innerHTML = 0;

}

*function* makeGuess () {

*const* guessInput = document.GetElementById (“guess”);

*const* guess = Number (guessInput.value);

guessInput.value = “ “;

*const* feedback = document.getElementById (“feedback”);

numGuesses++;

document.getElementById (“num-guesses”).innerHTML = numGuesses;

if (guess === num) {

feedback.innerHTML = “You guessed it”;

reset ();

} else if (guess > num) {

feeback.innerHTML = “it is lower”;

} else {

feedback.innerHTML = “it is higher”;

}

}

Line 1-3

Function that generates a random number

Line 5

we generate the random number as soon as html doc is loaded.

Line 6

we set the number of guesses equal to 0

Line 8 – 12

This function generates a new random number, set the number of guesses to 0, and then update our paragraph tag inside our html document.

Line 14-31

Responsible for

15-16 grabbing the value of the guesses that the user had

17 resetting that value field to be blank so we can enter a new number

18 we grab the feedback documents since we’re going to use that to actually update and tell the user if it’s higher or lower

20 we update the number of guesses

21 we set our number of guesses in our html document.

23-31 we check if the guess is correct, greater than or less than

The function starting on line 8 is going to be linked to the button tag inside the html document. Inside the makeGuess function, we’re gonna have to grab whatever the value of the <input> field is. So, we will write line 15 of JS. We will need to convert the string to a number.

Quiz review

Which line of code gives us a random integer from 0 to 100 and saves it to a constant?

Const num = Math.floor (Math.random () \* 101);

Explanation

To generate a random integer from 0 to 100, we can use the Math.random () function in combination with Math.floor ().

Here is an explanation of the correct line of code:

* Math.random () generates a random floating number between 0 (inclusive) and 1 (exclusive).
* Multiplying Math.random () by 101 gives us a random number between 0 (inclusive) and 101 (exclusive).
* Math.floor () rounds down the result to the nearest integer, effectively removing the decimal part.
* By multiplying Math.random () by 101 and applying Math.floor (), we get a random integer between 0 and 100 (inclusive)
* Finally, the value is saved to the num constant using the const keyword.

**SORTING**

Typically, we sort arrays but we have other possibilities. It’s common to want to sort different inputs when we write our program, and they won’t always be numeric inputs. We can have objects, subarrays.

**Example**

Here we have an array with some numeric values inside of it. By default, the program will read them as strings and sort them by the ASCII value, so the value of the first character of the numbers. In this case, it works because all the values are single digits.

*const* num = [1, 2, 3, 4, 2, 5, 2, 1, 2];

*const* sortedNums = num.sort ();

console.log (sortedNums);

How to sort without using the ASCII value?

To do that, we need to provide what is called a custom sort function

((a, b) = > a - b )

*const* num = [1, 2, 3, 34, 12, 50, 2, 1, 2, 6];

*const* sortedNums = num.sort ((a, b) = > a - b ) ;

console.log (sortedNums);

A is going to be the first number that we’re attempting to sort. B is going to be the second number. Here we’re comparing a and b. we want to return (write) a – b. that means that if larger than b, then we’ll get a positive value which indicates that a is larger than b. The result here will be

[ 1, 1, 2, 2, 2, 3, 6, 12, 34, 50]

If we wanted to maintain our original array still use the reverse function without altering the array, we could use the operator and it would give us a new array

This specific function is very useful to sort through much more complex values.

Example

Here we have different model cars from different years and we want to sort them by the oldest car first

*const* objs = [

{year: 2012, make: “Ford” },

{year: 2017, make: “Porsche” },

{year: 1999, make: “Volvo” },

{year: 2022, make: “Chevy” },

];

To do so, we can use the sort function on the array and use this logic. Since these values are all objects, we can write the .year and have access to the .year property

*const* objs = [

{year: 2012, make: “Ford” },

{year: 2017, make: “Porsche” },

{year: 1999, make: “Volvo” },

{year: 2022, make: “Chevy” },

];

*const* sortedObjs = objs.sort ( ( *a*, *b* ) => *a*.year - *b*.year )

console.log (sortedObjs)

If we wanted to do the opposite with the newest car first, we would write b.year – a.year.

If we had cars from the same years and we wanted to prioritize specific model cars, we can organize by the make.

*const* sortedObjs = objs.sort ( ( *a*, *b*) => {

*const* diffInYear = *b*.year - *a*.year;

if (diffInYear === 0) {

if (*a*.make > *b*.make) {

return 1

} else if (*a*.make < *b*.make) {

return -1

} else {

return 0

}

}

return diffInYear

} );

console.log (sortedObjs);

Here we ask the program to organize by year and if the difference in year is equal to 0, then we’ll go the next if condition.

Note: if we wanted to organize our list in a decreasing order without having to rewrite the code, we could use the reverse function in our console.log and it would reverse the results. It’s not giving us a new array, it’s mutating our original array **console.log (sortedObjs.reverse () );**

If we wanted to maintain our original array still use the reverse function without altering the array, we could use the operator and it would give us a new array.

[…sortedObjs].reverse ();

console.log (sortedObjs);