Programming JavaScript

Santosh Kalwar, 17.01.2022

JavaScript Topics

General Discussion

Setting up environments
- VS Code
- Node
- NVM – node version manager
- XCode

Learning materials walkthrough
- Books
- Slides

Course structures

JS basic operations and Coding together
- Basics functions
- Basic strings
- Numbers
- Variables
- Practice session and minor Projects

Setting up environments

Setting up environments

- VS Code
- Node
- NVM – node version manager
- Xcode
- Chrome browser

Learning materials walkthrough

Two e-books are now available in itsLearning:

JavaScript for Kids

https://unelmacloud.com/drive/s/rtwzuQZCRByqu90EUhRvSNgz4bYWgA

Eloquent JavaScript

https://eloquentjavascript.net/Eloquent_JavaScript.pdf

GET STARTED

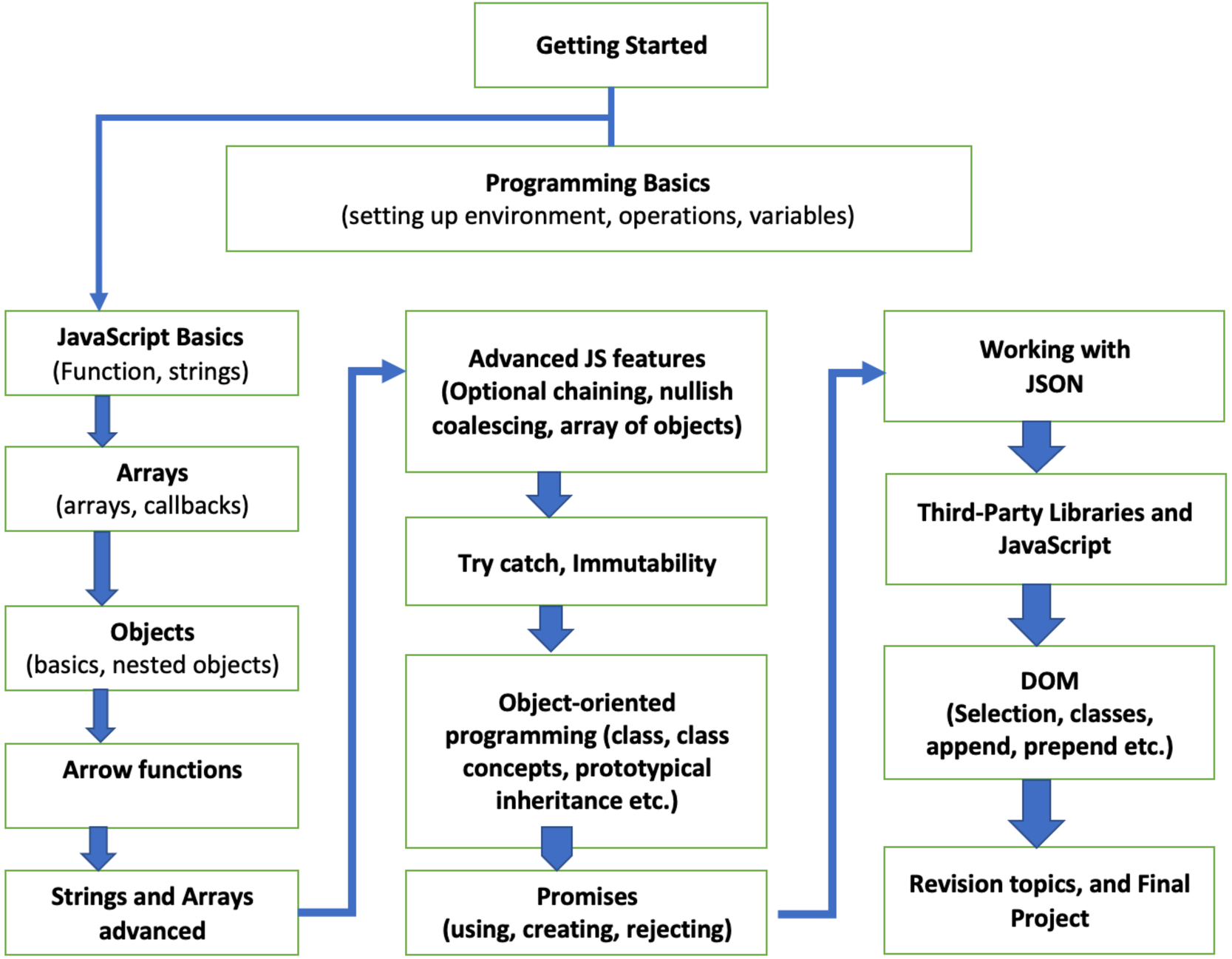
Top books to read in JS

https://www.guru99.com/javascript-books.html

Learning = making mistakes

Learning JavaScript (or any programming language) is a process of **making mistakes and learning from them**. Do NOT get discouraged when you can't solve a problem or challenge.

Programming JS Course Structure



Programming Basics

You do NOT need to have solid Mathematics knowledge to learn to program, even though it makes it easier if you're good at Mathematics, but it is not a prerequisite.

Some branches of programming (such as computer graphics programming or game development) require you to have solid Mathematics experience. But this is not the case with this course.

So as long as you know addition, multiplication, subtraction & division, you're good to go!

Where to get help?

- Google
- Search better, https://duckduckgo.com/
- Private search with no ads, https://unelma.xyz
- StackOverflow https://stackoverflow.com/
- MDN Web docs, https://developer.mozilla.org/en-US/docs/Web/JavaScript
- Forums and various online resources
- Ask, if any questions (MS Teams, Chat, Email etc)

Basic functions

Let's start with basic functions:

```
function sum(x, y) {
    return x + y;
}
```

This piece of code defines a function called **sum**.
This means that you can now call `sum(1, 3)` which returns 4.
You can run it again with different values, such as `sum(2, 5)` and it will return the result of 2 + 5 which is 7.

Returning the result

In JavaScript, you **have** to return from inside functions. If you forget to write `return`, your function will return `undefined`.

The `return` keyword will also **quit/exit** the function.

```
function sum(x, y) {
    return x + y;
    console.log("Hello World"); // this will NEVER run
}
```

The `return` keyword will quit the function with the result (which is `x + y`), so the code afterward will never run!

Strings

You can create a string in JavaScript by using the double quotes (") or single quotes (').

Here's an example:

```
"This is a string";
'This is another string!'
```

There is no difference between using a double quote or a single quote. They are exactly the same. **Neither** of these strings support interpolation (which means replacing a variable with its value inside of a string). String interpolation will be covered in a future lesson.

String property

The **.length** property is used to return the length of the string.

Here's an example of getting the length of "Nice!":

```
"Nice!".length;
 //5
```

Assuming you have a variable called **text**, here's how you'd get its length:

```
let text = "Hello World";
text.length; // 11
```

Basic String methods

Here are some common methods that you can call on strings:

.toLowerCase()

This will return a new string that has all of its characters in lower case:

```
"BLUE".toLowerCase(); // "blue";
```

Note that **.length** should not have `()` after it because it is a property (a value that has already been computed). Whereas `.toLowerCase()` is a method that requires the `()` because it's an **action** that you are performing.

.toUpperCase()

This will return a new string that has all of its characters in upper case:

```
"red".toUpperCase(); // "RED";
```

Character access

You can access a specific character in a string by using the square brackets syntax `[]`.

You have to provide the **index** of the character that you'd like to access, starting from 0.

Let's take an example where the variable `language` has the value: `"JavaScript"`. Here's how you access the 1st character, the 2nd, and the 3rd:

```
language[0]; //first character
language[1]; //second character
language[2]; //third character
```

If you'd like to *debug* your code and see the result of `language[1]` in the console, you have to console log it as follows:

```
console.log(language[1]);
```

Character access

Combining it with length

You can also combine the character access with the **.length** property. So using the same `language` variable, here's how you get the second to last character from it:

```
language[ language.length - 2 ]; // "p" because it's the second to last character from "JavaScript"
```

Note that **language[language.length]** will be `undefined` because character access starts at 0.
So for a string of 9 characters, 8 is the position of the last character.

Substrings

A substring is a part or a portion of a string. For example, "rain" is a substring of the string "brain" because you can get "rain" by taking the last 4 characters.

When working with strings, you often need to get a few characters of a string rather than all of it. Thus we use the substring method.

Substring signature

A function signature gives you an explanation of the parameters that you need to pass for that method. Let's take a look at the signature of substring:

```
someString.substring(indexStart, indexEnd)
```

This means that when you call substring, you can give it 2 parameters, the first one for the indexStart and the second one for indexEnd.

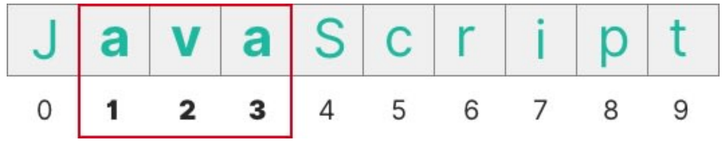
- indexStart: the position of the first character you'd like to **include**
- indexEnd: the position of the first character you'd like to **ignore**

This means an indexEnd of **5**, will only include up to character **4**.

The combination of these 2 will give you a substring.

Substrings example

Let's take an example with a variable named `language` with a value `JavaScript`, and let's get the substring with indexStart of 1 and indexEnd of 4. This will return a string made up of all the characters from positions 1 to 3 because 4 is the first character that is ignored:



The result of such substring is `"ava"`.

Here's how you'd write it in JavaScript:

```
const language = "JavaScript";
language.substring(1, 4); //"ava"
```

Optional parameters

The `indexEnd` parameter is optional, which means you can pass the `indexStart` and it'll assume the `indexEnd` to be the same as the string length. Here's an example:

```
language.substring(4); //"Script"
```

It assumed that the `indexEnd` is the length of the string (10 in this example).

Legacy note

If you already know a bit of JavaScript, you might have used another method that performs a similar result. You can find the name of the function below, but we do not recommend that you use it because it's deprecated.

Do not use the `.substr` method as it's deprecated and works differently. Always use the `.substring` method.

Plus operator

In JavaScript, the plus operator (+) will behave differently based on the types of values you use it with.

You've already seen that `1 + 3` will return the number `4`.

However, you could also use the `+` to concatenate 2 strings together, which means merging them together into 1 string.

Here's an example:

```
"Hello" + "World" //"HelloWorld"
```

will return one string: "HelloWorld". This would be useful if you'd like to concatenate 2 or more strings together. For example:

```
let prefix = "Mrs.";
let name = "Sam";
let string = prefix + " " + name; // Mrs. Sam
```

Plus operator, +=

+= operator

Say you have the following code:

```
let name = "Sam";
name = name + " Blue";
console.log(name); // "Sam Blue"
```

You can rewrite the `name = name +` in a shorter way using the `+=` operator:

```
let name = "Sam";
name += " Blue";
console.log(name); // "Sam Blue"
```

Template strings

You already know that you can create strings with double quotes or single quotes, but as you already know, these strings do not support interpolation.

Template strings, however, support interpolation and other nifty features.

Your first template string

```
`This is a template string`
```

The only difference is that template strings start and end with a backtick `` ` `` character.

The backtick is above the **tab** key on International keyboard layouts.

Multiline strings

Unlike single quote and double quote strings, template strings can span multiple lines. Here's an example:

```
let text = `This is a multiline
string that
just works!`
```

Whereas this would have **not** been possible with a normal string (single quotes or double quotes).

Interpolation

Template strings support interpolation! This means you could write a variable in your string, and get its value. The syntax is straightforward, you wrap your variable name with a dollar sign and curly braces. Let's take an example where we have a variable `language` with a value of `JavaScript`.

```
let language = "JavaScript";
`I am learning ${language}`; //"I am learning JavaScript";
```

Remember that string interpolation only works with **backticks**. If you ever try it and it doesn't work, double-check that you're using backticks rather than single or double-quotes.

Numbers

- 1
- 2
- -5
- 3.5
- 2000
- 2021
- -23.51

All of these are examples of Numbers in JavaScript. It doesn't matter if it's negative or positive, or if it has decimals (values after the `.`) or not. We call them numbers.

Converting from number to string

Though rarely used, you can convert a number to a string by calling the `.toString()` method.
Let's take an example where we have a variable called `answer` with a value `42`;

```
let answer = 42;
answer.toString(); //"42"
```

Documentation

If you take a look at online documentation, you will often see `String.prototype.toString()`. Why is there a `prototype`? This is covered in-depth later in this course. For now, every time you see `String.prototype.something()`, it means there is a method `something()` that you can call on a String.

Mozilla Developer Network is the most authoritative website for JavaScript documentation. If you're getting started with JavaScript, however, you might find it a bit hard to grasp.

Therefore in this course, we are aiming to provide you with short and easy-to-consume explanations, so you don't feel overwhelmed when you are just beginning your learning journey.

When you are comfortable with what Learn JavaScript says about a topic and want to dig deeper and find out more, look for the MDN logo  and click the link that follows it to access much more in-depth information.

NaN

You may sometimes encounter `NaN` which stands for **Not a Number**. For example, if you try to multiply a number by a string (which you should not do):

```
// ❌
"abc" * 4; // NaN
```

NaN is often a sign that something is wrong with your code, most often you forgot to convert a string to a number. One of the most common cases is when an object property evaluates to `undefined` because of a typo and then it's used as if it was a valid number (more on that in later chapters as we learn about objects and object properties).

Convert string to number

In some scenarios (explained below), you'd like to convert from a string to a number. For that, you'd have to use the `Number.parseInt()` method.

Here's an example:

```
let str = "42";
Number.parseInt(str, 10); //42
```

There's a lot going on here, so let's break it down step by step.

The function name is called `Number.parseInt()`. Yes, including the `Number.` bit. This is because there's a global object called `Number` which contains a method called `parseInt()`.

This `Number.parseInt()` method expects 2 parameters:

```
Number.parseInt(string, radix);
```

The first parameter is the string that you'd like to convert into a number. The second argument is the **radix** that will be used in the conversion.

The `radix` is the base of the numerical system that you'd like to use. For most use cases the radix you'd like to use is `10` which represents the way we count numbers in our everyday lives. This system is called the decimal system (because we have 10 fingers, so we use the digits from 0 to 9).

Another example of radix is `2` which represents binary (a numerical system used by computers). If you'd like to dig deeper into this concept, check out this simplified Wikipedia page about **Mathematical bases**.

As a quick summary, the radix will most often be `10`. If you're not sure what radix to choose, then it's most likely 10.

Can I skip the radix?

Can I skip the radix?

Even though the radix is an optional parameter, you should **not** skip it. That's because it does **not** always default to 10.
So make sure to always pass the radix as the 2nd parameter.

Number.parseInt(string, radix) does **not** always default to a radix of 10.

If you do try `Number.parseInt()` without a radix of 10, it will work. However, there are some edge cases (numbers starting with `0x`) that would break. Thus, to be safe, it's always recommended to pass the radix.

Make sure to always specify the radix to avoid unpleasant surprises.

 **Number.parseInt()** on MDN

Legacy notes

JavaScript is an evolving language that is over 25 years old. It keeps changing and evolving. Legacy notes will explain some confusing behavior or old functions that you may have encountered a while ago.

If you are learning JavaScript for the first time, then you don't need to spend a lot of time in those legacy notes.

Number.parseInt() and parseInt() are exactly the same thing. Prefer `Number.parseInt()` over `parseInt()`

A while ago, `parseInt(string, radix)` was the only way to convert numbers, however, a while later, this function has been cloned under the `Number` object and became `Number.parseInt(string, radix)` in an effort to group similar functions together under their relevant object.

They both work exactly the same. We do recommend that you stick with the modern approach which is `Number.parseInt()`.

Use cases for converting to a number

There are several reasons why you'd like to convert a string to a number, but the most common one is when the number is entered by the user in a text box or the number is being read from the DOM (which is explained later on).

As you will see in the next challenge, these values will always be a string (even if the user writes a number). Thus, it is your job to convert it to a number.

If you forget to convert a string to a number, you will see that the intended addition is behaving like concatenation:

```
let a = 10;
let b = "20"; // we forgot to convert it to a number
a + b; // "1020" (concatenation instead of sum)
```

The `Number.parseInt()` method will try to convert the string it receives into a number. As you can see below, it most often works when the string starts with a number and ends with non-numeric values:

```
Number.parseInt("123abc", 10); // 123
Number.parseInt("5 meters", 10); // 5
```

Operations

As you might expect, numbers can be multiplied and divided. For division, you need to use the `/` operator.

Division remainder

You can also use the remainder operator `%` which returns the division remainder. Here's an example:

```
8 % 2; // Division remainder is 0
7 % 2; // Division remainder is 1
```

Here's how the division remainder of 8 % 2 is calculated:

8 / 2 = 4
4 / 2 = 2
=> division remainder is 0 because 8 = 4 * 2 + 0

Where as for 7 % 2:

7 / 2 = 3 (rounded)
=> division remainder is **1** because 7 = 3 * 2 + **1**

We'll use the division remainder in 2 chapters from now to check whether a number is **even** or **odd**.

Number methods

While there are some other methods you could call on numbers, they are not very commonly used. What is commonly used, however, is the `Math` object which contains methods such as `min()`, `max()`, `round()`, etc. This is covered in a dedicated chapter later in this course.

Variables, let

There are 2 ways to define a variable in JavaScript. Let's take a look at the difference between `let` and `const`.

let

The first time you define a variable, you have to prefix it with `let =` . Let's take an example:

```
let name = "Sam";
console.log(name);
```

This defines a variable called `name` with a value of `"Sam"`. The next time you'd like to use that variable, you reference it by its name (you only use the `let` keyword for assignment).

Variables defined with `let`, can be re-assigned later on:

```
let language = "C++";
language = "JavaScript";
```

Another example with numbers:

```
let sum = 0;
sum += 1;
```

This is especially useful when you want to create a variable that needs to be incremented/decremented (such as a counter).

Variables, const

const

Variables defined with `const` cannot be re-assigned. This means you can use the `=` sign once when the variable is defined. Here's an example:

```
const language = "C++"; // Cannot be re-assigned anymore
console.log(language); // "C++"

language = "Python" // ❌ Type error: this will break your script
```

 **const** on MDN

A note about const

An important note about `const` is that it does **not** create a Constant or an Immutable value. This will be thoroughly explained once we learn about arrays & objects. What you need to know, for now, is that you can only use the equal sign once, but you can still change elements **inside** an array or object.

let vs const

How do you decide if you're going to use `let` or `const`? The general rule is easy. Always go with `const`, until you realize that you need to be able to re-assign the variable later on, then switch it to `let`.
With time it becomes easier. For example, when you define a variable `count` (that you expect to increment), you will immediately realize that and use `let`.

The benefit of using `const` is that once a variable is an array, it will always be an array (but as you will see later on, the elements inside the array might change). This allows you to confidently use array methods on that variable because you know it will always be of type array.

Can I use ‘var’

When you're browsing the Internet for documentation, or Questions & Answers on StackOverflow, you will see a lot of code snippets using `var` instead of let & const.
Even though `var` still works, its usage is discouraged as it may be confusing in a lot of scenarios. So you can simply replace `var` with `let` (or `const` if the variable is not being reassigned).

Avoid using `var` when defining variables. Use `let` or `const` instead.

Practice

- Complete the function **sum** such that it returns the sum of **a** and **b**.
- Complete the function multiply such that it returns the product (result of multiplication)
- Complete the function `getCharCount` such that it returns the number of characters in the `str` parameter that it receives.
- Complete the function shoutMyName such that it returns the name parameter it receives all in upper case.
- Complete the function `lowerName` such that it returns the `name` parameter it receives all in lower case.
- Complete the function getFirstCharacter such that it returns the first character of the name parameter it receives.
- Complete the function getLastCharacter such that it returns the last character of the name parameter it receives.
- Complete the function skipFirstCharacter such that it returns all the character except the first one from the text parameter it receives.
- Complete the function concatInitials such that it returns the firstNameInitial followed by the lastNameInitial
- Complete the function sayHello such that it interpolates the variable **name** into a string “Hello name”
- Complete the function `getFullName` such that it returns the full name of the person using **interpolation**.
- Complete the function `capitalize` such that it capitalizes the `name` parameter it receives. There's no capitalize method in JavaScript, so you have to write it yourself.
- Complete the function `convertNumberToString` such that it converts the number it receives into a string.
- Complete the function `getNextAge` such that it returns the age next year (by adding 1 to the current age).
- Define a variable called **count** with an original value of 0 and then increment it (add 1 to it) on the following line.
- Define a variable ageLimit that cannot be re-assigned and give it a value of 18

JS
Strings
.toUpperCase()
.toLowerCase()
.substring()
.startsWith()
.split()
.replaceAll()
.replace()
.length
.includes()
.endsWith()
Concatenation
.trim()
Template strings
[] (character access)
?? (nullish coalescing)
Variables
var (legacy)
let
const
Arrays
... (spread)
.every()
.filter()
.find()
.forEach()
.includes()
.join()
.length
.map()
.push()
.reduce()
.some()
.splice()
?. (optional chaining)
[] (get item)
destructuring
Functions
=> (arrow functions)
?. (optional chaining)
basic functions
default parameters
implicit return
lexical scope
Number
.toString()
Division remainder (%)
NaN
Number.parseInt()
Objects
property access
object shorthand
dynamic properties
destructuring
Object.values()
Object.keys()
Object.entries()
?? (nullish coalescing)
?. (optional chaining)
... (spread)

Conditional Operations

- Conditions
- Practice

Conditions

Conditions in JavaScript have the following blueprint:

```
if (condition) {
    //do something
}
```

Let's take an example:

```
const grade = 15;

if (grade >= 10) {
    console.log("Passing grade");
}
```

The code above will output to the console: "Passing grade".

Conditions: else

else

You can also add an `else` block for all other cases:

```
const grade = 3;

if (grade >= 10) {
    console.log("Passing grade");
} else {
    console.log("Failing grade");
}
```

The code above will output to the console: "Failing grade”.

Conditions: else if

else if

Several conditions can be checked sequentially using `else if`. For example:

```
const grade = 10;

if (grade > 10) {
    console.log("Passing grade");
} else if (grade === 10) {
    console.log("Passing on the limit");
} else {
    console.log("Failing grade");
}
```

The code above will output to the console: "Passing on the limit”.

ligatures

A note regarding ligatures

Note that the `===` that you're seeing is in fact **3 equal signs** after each other ===
The fact that they show up as a single character is a feature that you can enable in your code editor, it's called a **ligature** and is supported by some fonts.
Here are some other examples of ligatures that you will see:

- `>=` for >=
- `<=` for <=
- `===` for ===
- `!==` for !==

Advanced if

It is possible sometimes to drop the `else`. Let's take a look at an example:

```
function canVote(age) {
    if (age >= 18) {
        return true;
    } else {
        return false;
    }
}
```

Since this function is performing two different actions based on the result of the if condition and its opposite (else), then we can rewrite it by dropping the `else` keyword:

```
function canVote(age) {
    if (age >= 18) {
        return true;
    }
    return false;
}
```

These two functions will have the exact same result. That's because the `return` keyword will quit the function with the result. So, when the age is bigger than or equals 18, the function will return `true` and the rest of the code will **not** execute.
However, when the age is less than 18, then the code inside the `if` condition does not execute. Thus, the only line that executes is the last one, which is `return false`.

We will take advantage of this tip later in this course to learn about a common pattern called **early return**.

Advanced if (legacy notes)

A quick legacy note. If you encounter == (double equal) in JavaScript, aim to replace it with === triple equal. The double equal operator performs some conversions that you wouldn't expect. Always stick with triple equal instead.

Always use triple equal === when comparing 2 values in JavaScript.

If you'd like to know more why == is not recommended, then check this comparison:

```
"2" == 2;
```

Would this return true or false?

It would return true because JavaScript will try to convert both values into the same data type. Please don't consider this a "feature". Instead, you should avoid it and always use triple equal ===.

Returning booleans

Whenever you're returning a boolean (`true` or `false`), it's quite redundant to use `if` and `else`. Here's an example:

```
function isPassing(grade) {
    if (grade >= 10) {
        return true;
    } else {
        return false;
    }
}

isPassing(12);
```

This is redundant because `grade >= 10` on its own, evaluates to **true** or **false** depending on the grade. This means you don't need to wrap it with an if/else statement.

That's why you can refactor it like this:

```
function isPassing(grade) {
    return grade >= 10;
}
```

without using if/else which will always return a boolean!

This only works whenever you're returning a boolean from a function.

Even & Odd

Back in the Numbers chapter, we explained the Remainder operator (`%`). When you get the division remainder of a number by two, you will either get `0` or `1`. This will help you know whether the original number is *even* (divisible by two without a remainder) or *odd*.

Let's take a look at some numbers:

```
// even numbers
4 % 2 // 0
6 % 2 // 0
8 % 2 // 0
10 % 2 // 0

// odd numbers
3 % 2 // 1
5 % 2 // 1
7 % 2 // 1
9 % 2 // 1
```

Notice how the division remainder by two is always 0 when the number is **even**. Whereas it's 1 when the number is **odd**.

In the next challenge, we'll use an **if condition** to show whether the number entered by the user is even or odd!

Practice

- Implement the function `canVote` such that it returns `true` whenever the age 18 or above and `false` in all other scenarios.

- Complete the function `getNextAge` such that it returns the age next year (by adding 1 to the current age).

- Implement the function `canVote` such that it returns `true` whenever the age 18 or above and `false` in all other scenarios. You should **not** use an `if condition` (or ternary).

- Complete the function `evenOrOdd` such that it returns the string **"even"** when the number parameter it receives is even and **"odd"** otherwise. Can you make it work with negative numbers too?

- Write a function named greaterNum that: takes 2 arguments, both numbers. returns whichever number is the greater (higher) number. Call that function 2 times with different number pairs, and log the output to make sure it works (e.g. "The greater number of 5 and 10 is 10.”).

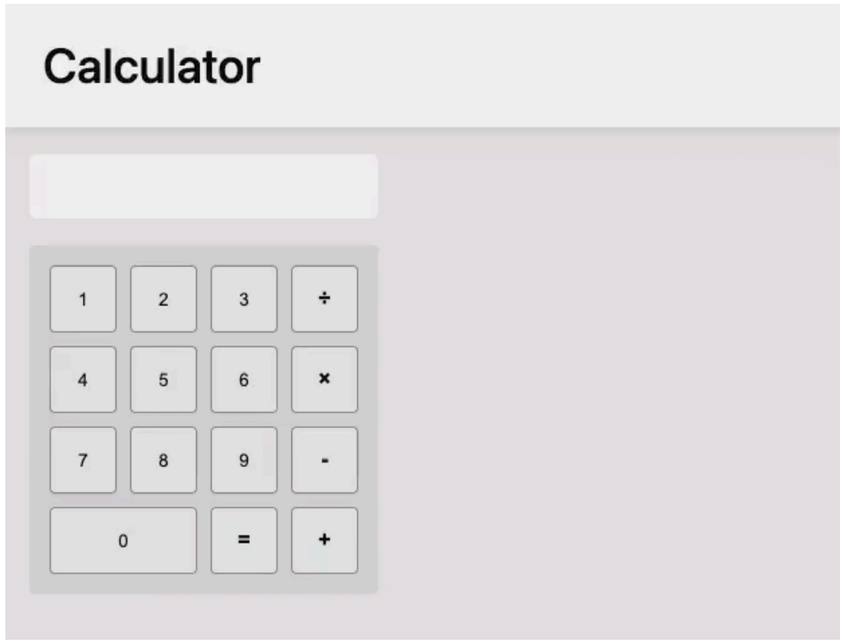
- Write a function named assignGrade that: takes 1 argument, a number score. returns a grade for the score, either "A", "B", "C", "D", or “F". Call that function for a few different scores and log the result to make sure it works.

- Write a function named pluralize that: takes 2 arguments, a noun and a number. returns the number and pluralized form, like "5 cats" or "1 dog”. Call that function for a few different scores and log the result to make sure it works. Bonus: Make it handle a few collective nouns like "sheep" and “geese".

Supermini Project 01

In this supermini project, you will complete the 4 existing functions in the **index.js file** so that we can have this end result!

You can try the calculator by clicking on the **BROWSER** tab above. Start making it work step by step and keep on trying the calculator in the browser ;)



Supermini Project 02

In this project, you will get to practice your knowledge so far by creating a **name variations** app. When you enter a name in the text box, it will tell how many characters it is, how it would look in lower case, and how it would look in upper case. This will let you practice your knowledge with functions, strings, and programming logic!