Beforehand: these notes are just me going through javascript.info and condensing them to refresh myself on the basics.

# Fundamentals

## Hello, World!:

We can use a <script> tag to add JavaScript code to a html page.

The type and language attributes are not required.

A script in an external file can be inserted with <script src="path/to/script.js"></script>.

## Code Structure:

TLDR: Statements are written on separate lines with a semicolon. Though in most instances a semicolon can be omitted in place of a line break, there are exceptions when the statement is assumed to continue.

- You can make comments using // or /* */.

## Use Strict:

TLDR: JavaScript is an ongoing developing language. However, devs can't really go back and change things because it might break old websites. This was the case until ECMAScript 5 was introduced, which added new features and modified the old ones. Though you need to explicitly enable them with "use strict";.

## Variables:

Naming Variables

- Use human-readable names like userName or shoppingCart.
- Stay away from abbreviations or short names like a, b, and unless you know what you’re doing.
- Make names maximally descriptive and concise. Examples of bad names are data and value. Such names say nothing. It’s only okay to use them if the context of the code makes it exceptionally obvious which data or value the variable is referencing.

- Agree on terms within your team and in your mind. If a site visitor is called a “user” then we should name related variables currentUser or newUser instead of currentVisitor or newManInTown.

DataTypes:

There are 8 basic data types in JavaScript. We can put any type in a variable and switch them at our leisure. Programming languages like these are called "dynamically typed".

I'll do more of this section later because it's getting kind of boring.

- More Info

Interaction:

- alert
  - Shows a message.
- prompt
  - Shows a message asking the user to input text.
  - It returns the text or, if Cancel button or Esc is clicked, null.
- confirm
  - Shows a message and waits for the user to press “OK” or “Cancel”.
  - It returns true for OK and false for Cancel/Esc.

Type Conversions:

The three most widely used type conversions are to string, to number, and to boolean.

- String conversion occurs when we output something. Can be performed with String(value). The conversion to string is usually obvious for primitive values.
- Numeric conversion occurs in math operations. Can be performed with Number(value).

Basic Operators:

The plus + exists in two forms: the binary form that we used above and the unary form. The unary plus or, in other words, the plus operator + applied to a single value, doesn’t do anything to numbers. But if the operand is not a number, the unary plus converts it into a number.

Let’s note that an assignment = is also an operator. It is listed in the precedence table with the very low priority of 2.

We often need to apply an operator to a variable and store the new result in that same variable. This notation can be shortened using the operators += and *=.

- Short “modify-and-assign” operators exist for all arithmatical and bitwise operators: /=, -=, etc.

Increment ++ increases a variable by 1.

Decrement -- decreases a variable by 1.

- If we’d like to increase a value and immediately use the result of the operator, we need the prefix form
- If we’d like to increment a value but use its previous value, we need the postfix form

Bitwise operators are not necessarily useful for web dev. For more info: MDN Web Docs

Comma operator is one of the rarest and most unusual operators. Sometimes it's used to write shorter code. Allows evaluation of several expressions dividing them with a comma. Each of them is evaluated but only the last expression is returned.

- Commas have very low precedence.

Sometimes, people use it in more complex constructs to put several actions in one line.

For Example:

```
// Three operations in one line
for (a = 1, b = 3, c = a * b; a < 10; a++){
 ...
}
```

Comparisions:

There are the basic comparisons…

- Less Than: a < b

- Greater Than: a > b
- Equals: a == b
- Not Equal: a != b

Boolean is the result of the comparison; True or False.

You can also compare strings; the string is compared letter by letter using lexicographical order.

- If the first letter of the string a is greater or less than than string b then the comparison is done.
- Otherwise, if the first letter is the same they move on to comparing the second letters.
- If both strings end at the same length, then they are equal. Otherwise, the longer string is greater.

  For example:

```
'Z' > 'A' //true
'Glow' > 'Glee' // true
'Bee' > 'Be' //true
```

You can also compare different type values; JavaScript converts the values into numbers

- ‘1’ becomes 1
- ‘02’ becomes 2
- True becomes 1
- False becomes 0

Operands of different types are converted into numbers by the == operator.

Using === instead is called a strict equality and does not convert the values into numbers.

- So (true === 1) would be false.

There is non-intuitive behavior when null or undefined are compared to other values

These do not apply to strict equality checks.

- null == undefined // true
- For other comparison operators
- null becomes 0
- Undefined becomes NaN

Conditional Branching:

The if(...) evaluates the expression within the parenthesis and if it is true the corresponding code executes. The condition inside the parenthesis follows the same conversion rules as discussed previously.

For Example:

```
if(true)alert('code executes');
```

or

```
if(true){
    alert('this code executes');
    alert('this code executes');
}
```

You can add an else block after the if block and that will execute if the expression is false.

You can add an else if(...) block after the if block that will only execute if the previous condition is false and the current condition is true.

These blocks can be chained together, however there must only be one else and it must be at the end of the chain.

Sometimes, we need to assign a variable depending on a condition. For this we can use a “conditional operator” or a “?” operator

Example:

```
let accessAllowed = (age > 18) ? true : false;
```

A sequence of question mark operators ? can return a value that depends on more than one condition.

For instance:

```
let age = prompt('age?', 18);
let message = (age < 3) ? 'Hi, baby!' :
(age < 18) ? 'Hello!' :
(age < 100) ? 'Greetings!' :
'What an unusual age!';
alert( message );
```

Here's how this looks using if..else:

```
if (age < 3) {
   message = 'Hi, baby!';
} else if (age < 18) {
   message = 'Hello!';
} else if (age < 100) {
   message = 'Greetings!';
} else {
  message = 'What an unusual age!';
}
```

Sometimes the question mark ? is used as a replacement for if:

```
let company = prompt('Which company created JavaScript?', '');
(company == 'Netscape') ?
   alert('Right!') : alert('Wrong.');
```

Here is the same code using if for comparison:

```
let company = prompt('Which company created JavaScript?', '');
if (company == 'Netscape') {
    alert('Right!');
} else {
    alert('Wrong.');
}
```

Logical Operators:

There are Four Logical Operators

- || (OR)
- && (AND)
- ! (NOT)
- ?? (Nullish Coalescing) ??

OR

The result is always true unless both sides of the condition are false

Most of the time or is used in an if statement to check if any of the conditions are correct

There can be more than two conditions

Example:

```
result = value1 || value2 || value3;
```

Evaluates operands from left to right, and for each operand it converts them into a bool.

If the operand it is checking is true, it stops evaluating and returns the original value of that operand.

In the case they are all false, it returns the last operand.

This leads to interesting use cases:

```
alert(false || false || "True");
```

Or operators have a feature called “short-circuit” evaluation, it means the OR processes arguments only until a truthy value is reached.

For Example:

```
true || alert("not printed");
false || alert("printed");
```

AND

The result is false unless all operands contain a truthy value.

In the case that all operators are true it returns the last operand

In the case that the and operator goes through a falsy value it immediately returns it

Note: && has a higher precedence than ||

NOT

This operator only accepts a single operand

For Example:

```
Result = !value;
```

In the example, the value variable is converted to a boolean then return the inverse value.

Double Not (!!) may be used just to convert a type to boolean. Since the inverse of the inverse is just the boolean conversion of that value.

For Example:

```
Result = !false //true
Result = !0 //true
Result = !!0 //false
```

The nullish coalescing operator (??) returns the first argument if it’s not null or undefined.

Technically Or (||) can do the same thing, but the distinction is that the nullish coalescing operator returns the first defined value (not null or undefined) and || or returns the first truthy value.

They both have the same precedence.

It’s forbidden to use it with || or && without explicit parentheses.

Loops:

Loops are used to repeat actions.

The While Loop:

```
while(condition){
      //code
}
```

As the name suggests, the while loop repeats the code inside the block as long as the condition is true.

A single execution of the loop is called an iteration

Any variable can be a condition not just a comparison.

The do-while loop:

```
do{
//code
} while(...);
```

The loop will execute the body, and then check the condition; In other words the condition doesn’t have to be truthy to run the first but does have to be true for subsequent interactions to occur.

The big one, The for loop:

The for loop is slightly more complex, but it is the most used one. A professor once told the class that a for loop can take the place of any other loop type.

```
for (begin; condition; step){
```

```
    //code
}
```

Begin: Executes once upon entering the loop.

condition: If false, the loop stops.

Body: Runs again and again while the condition is truthy.

Step: Executes after the body on each iteration.

In most cases, the step is an iteration of a variable such as i++ i-- i+= etc…

We can skip any of the steps and the loop will still function, if you skip all of them then it's is just an infinite loop;

Normally, the loop stops when the condition is not met, but we can also use the break directive, which will just break out of the loop on the spot.

```
while(true){

    if(true) break;
}
```

Usually, due to the condition always being true it would be an infinite loop but the break line breaks out of the loop. Good name!

There is also a continue directive which just disregards the code after it in the loop block and goes to the next iteration.

```
while(true){

    alert('wow he must really like me!');

    if(true) continue;

    alert('why does he not like me...');
}
```

Switch Statement:

The switch statement is a statement that can replace multiple if checks

Syntax:

```
switch(x) {
  case 'value1':  // if (x === 'value1')
    ...
    [break]

  case 'value2':  // if (x === 'value2')
    ...
    [break]

  default:
    ...
    [break]
}
```

If no case is true than the default code runs. Also you can remove the break directive and all of the cases which values are true will run.

You can group two cases by putting them in the adjacent lines and the both cases will run the same code as such:

```
switch (x) {
  case 4:
    break;

  case 3: // (*) grouped two cases
  case 5:
    alert('*'); //same code will run twice
    break;

  default:
    alert('default);
}
```

Important note, the equality check is strict!

Function Declarations:

Oh my god we're finally here…

Sometimes, we need to perform the same action throughout the code. So rather than copy-paste the same code one-million times, we can just put it in a function.

Here is how to declare a function:

```
function exampleFunction(parameter1, parameter2,... paramaterN){
                //code
}
```

You can call a function and the code written inside the function will run:

```
exampleFunction(argument1, argument2,... argumentN);
```

There's some interesting things here.

First of all you can see there are things called parameters and arguments.

When creating a function you can create parameters and when calling a function you can pass arguments, but you don't have to if you don't need to.

For Example:

```
function sayName(fullName, age){
    alert('Hello, ' + firstName + ' age: ' + age );
//alert in it of itself is a function that is built
in JS
}
sayName("John Smith", 57);
sayName("Roman Martinez", 21);
```

A function can also return a value of any type as such:

```
function sum(a, b) {
    return a + b;
    alert('return worked!');
}
```

```
let result = sum(1, 2);
alert( result ); // 3
```

However, there is a problem with this function… the alert will never happen because as soon as return is called the function breaks and the rest of the code does not run. It can be used in place of the break directive if you so choose and you should probably choose because that's what most people do.

If a function returns nothing, or return is never called the value of the function will automatically return undefined.

That's the basics, but there are some extra things that are useful to know.

> In the case of calling a function and passing parameters you can add default values to the parameters that will take the place of the undefined argument.

```
function showMessage(from, text = "no text
given") {
    alert( from + ": " + text );
}

showMessage("Ann"); // Ann: no text given
```

> Also returns can be written like this:

```
return (
  some + long + expression
  + or +
  whatever
 )
```

If you try to do a long return statement and you do a new line and you don't do it like this, after the first line break it will just return and the code after that will do nothing.

Some more rules to follow:

A variable made inside a function called a local variable is only available within the scope of the function. But the fun thing is that you can use variables outside the function (global variables). Only of course if the variables are not inside another function or an if statement or a loop.

Now you may be asking, what if there is a global variable and a local variable that share the same name? well you probably shouldn't do that, but the local variable takes precedence.

Also, a function should only do one function. So, try to follow that principle and you'll have cleaner code.

Also don't get freaky with the names, they should be verbs like getSomething or findSomething.

Function Expressions:

Functions are not just a structure built into the language, they are actually a value. Before we discussed function declarations, which is one way to write a function, but we can also write a function in the middle of any expression.

For Example:

```
Let sayHi = function(){
     alert("Hello")
}; //DON'T FORGET THE SEMICOLON!
```

Here we don't need to name the function because it is already assigned a value. This tells us that before when creating a function declaration in the last section, we were actually assigning the function to a variable. In this case, let's say there was a function declaration of sayHi it would automatically be assigned to a value named sayHi.

If you need further proof, besides me telling you… you can try to create an alert with sayHi as the argument (without parenthesis because we're not calling the function).

For Example:

```
alert(sayHi);
```

You would find that alert would print out the function as you wrote it instead of the value that would be returned if it was called.

That being said like normal values you can copy the value to another variable as such:

```
Let sayHi2 = sayHi;
alert(sayHi);
```

When you pass a function as an argument, you can make use of something called a Callback Function.

Pontificate this:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

function showOk() {
  alert( "You agreed." );
}

function showCancel() {
  alert( "You canceled the execution." );
}

ask("Do you agree?", showOk, showCancel);
```

It’s interesting because we gave the yes and no parameters showOk()’s contents and showCancel()’s contents to the no parameter. showOk and showCancel are the ‘callbacks’ in the term ‘callback function’ fyi.

But as you can see this is not a function expression, but a function declaration. We can make use of function declarations to make this code a bit shorter:

```
function ask(question, yes, no) {
```

```
  if (confirm(question)) yes()
  else no();
}

ask(
  "Do you agree?",
  function() { alert("You agreed."); },
  function() { alert("You canceled the execution."); }
);
```

Though function declarations and expressions are similar, there are slight differences. Functions made with expressions become available only after they are run in the code. But function declarations can be called whenever throughout the code as long as they are written.

Also, when in strict mode, if a function declaration is within a code block, it's only visible locally not globally.

Only if a variable is made globally can it be assigned a function expression within a code block and it would be able to be used anywhere outside of the code as such:

```
let age = prompt("What is your age?", 18);

let welcome;

if (age < 18) {

  welcome = function() {
    alert("Hello!");
  };

} else {

  welcome = function() {
    alert("Greetings!");
  };

}

welcome(); // ok
```

Arrow Functions:

There’s yet another way to create functions. Often this way is preferable to function expressions.

Let’s see:

```
let function = (arg, arg2, ..., argN) => expression
```

‘So you can see it is similar to function expressions except there is no ‘function’ keyword insight. So I guess it looks cooler and it’s shorter to write so it's fun! Some things I can’t show by syntax alone is that if there is only one arg than there is no need for parentheses:

```
let function = arg => expression
```

but if there is no arguments then there must be parenthesis enclosed in nothing:

```
let function = () => expression
```

Also, the expression can be replaced by curly-braces and be multi-line:

```
let sum = (a, b) => {
  let result = a + b;
  return result;
};

alert( sum(1, 2) ); // 3
```