38 visual JavaScript tips, explanations, and algorithm implementations

@oliverjumpertz

JS

The Nullish Coalescing Operator

The logical ***OR*** reacts to ***falsy*** values.

This might often not be what you actually want.

```
const value = a || b;
```

The nullish coalescing operator only reacts to ***null*** and ***undefined***.

Falsy values are kept.

```
const value = a ?? b;
```

@oliverjumpertz

JS

Better Use The Strict Equality Comparison Operator ===

== is the abstract equality comparison operator.

It coerces types and leads to interesting results.

```
1 == "1"; // true

0 == ""; // true

null == undefined; // true

[] == ""; // true
```

Better use the strict equality comparison operator.

It doesn't coerce types and leads to more predictable results.

```
1 === "1"; // false

0 === ""; // false

null === undefined; // false

[] === ""; // false
```

@oliverjumpertz

JS

Use Numeric Separators To Keep Large Numbers Readable

Large numbers are difficult to read.

```
const largeNumber = 982563162;
```

Numeric separators help to make sense of larger numbers because they create readable groups.

```
const largeNumber = 982_563_162;
```

@oliverjumpertz

JS

Quickly Filter Out All Falsy Values From An Array

By using filter and passing the Boolean constructor, you can quickly filter out all falsy values from an array.

```
const array = [1, null, undefined, 0, 2, "", 4];

const result = array.filter(Boolean); // ⇒ [1, 2, 4]
```

filter filters out all values where the function passed returns ***false***.

Remember: JavaScript has first-class functions.

This is why you can directly pass the ***Boolean*** constructor function to ***filter***.

@oliverjumpertz

JS

Beware Of Automatic Semicolon Insertion

Although JavaScript doesn't require semicolons, it inserts them on certain occasions.

A ***return*** statement is one of those.

```
const myFunction = () => {
  // ...
  return
  {
    propertyOne: 1,
    propertyTwo: '2'
  };
};
```

This is actually evaluated as:
return undefined;

A ***return*** statement must be followed by something that keeps it open, like a curly brace or a bracket.

Then what follows is also taken into account up to a certain point.

```
const myFunction = () => {
  // ...
  return {
    propertyOne: 1,
    propertyTwo: '2'
  };
};
```

@oliverjumpertz

JS

Get **All Unique Values** From An Array

By creating a ***Set*** from your array and **spreading** this ***Set*** into a new array, you get an array only containing the unique values of the original one.

```
const array = [1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 4];

const uniqueValues = [ ... new Set(array)]; // ⇒ [1, 2, 3, 4, 5, 6]
```

Spreading the ***Set*** extracts its individual values and places them into a new array (because the expression is wrapped with square brackets).

A ***Set*** does only contain **unique values**.

When you construct the ***Set*** from the original array, all duplicates are **filtered** out.

@oliverjumpertz

JS

The Destructuring Assignment

Extracting properties one by one can **bloat** your code pretty fast.

```
const twitterResponse = await fetchUserData('@oliverjumpertz');

const name = twitterResponse.data.name;
const following = twitterResponse.data.following;
const followers = twitterResponse.data.followers;
```

Extracting the properties with a **destructuring assignment** removes a lot of repetitive code.

```
const twitterResponse = await fetchUserData('@oliverjumpertz');

const { name, following, followers } = twitterResponse.data;
```

@oliverjumpertz

JS

Array Destructuring

Extracting individual values from an array works but is inflexible.

It can use a lot of space.

```
const array = [1, 2, 3, 4, 5];

const firstValue = array[0];
const secondValue = array[1];
const thirdValue = array[2];
```

Array destructuring is flexible.

It uses less space, and even allows you to ignore values and get the remaining elements not yet extracted as a separate array.

```
const array = [1, 2, 3, 4, 5];

const [firstValue, , thirdValue, ...rest] = array;
```

@oliverjumpertz

JS

Use Array Destructuring To Swap Two Variables

Swapping two variables using a temporary variable works but takes quite a few lines of code.

```
let a = 1;
let b = 2;

let tmp = a;
a = b;
b = tmp;
```

You can use **array destucturing** to swap the two variables.

This takes less code, and often has better readability.

```
let a = 1;
let b = 2;

[a, b] = [b, a];
```

@oliverjumpertz

JS

Destructuring From A Possibly Undefined Property

Optional chaining does **not** work for the last layer of nesting.

```
const obj = {
  nested: undefined
};

const { one, two } = obj?.nested;
```

TypeError: Cannot read property 'one' of undefined

Defining a fallback with the ***nullish coalescing operator*** makes this operation safe at runtime.

```
const obj = {
  nested: undefined
};

const { one, two } = obj?.nested ?? {};
```

@oliverjumpertz

JS

Get The Last Items Of An Array

```
const array = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

const lastElement = array.slice(-1); // => [9]
const lastTwoElements = array.slice(-2); // => [8, 9]
const lastThreeElements = array.slice(-3); // => [7, 8, 9]
```

You can use ***Array.prototype.slice*** with a negative number to get the last items of an array.

@oliverjumpertz

JS

Swap Elements Within An Array With Destructuring

Destructuring also works for array elements and swaps the position of both elements in the array.

```
const array = [1, 2, 3];

[array[0], array[2]] = [array[2], array[0]]; // => [3, 2, 1]
```

This is the same as swapping two regular variables.

@oliverjumpertz

JS

Destructure An Object Into Existing Variables

You **can't** simply destructure an object into existing variables.

```
const obj = {
  a: 1,
  b: '2'
};

let a, b;

{ a, b } = obj;
```

SyntaxError: Unexpected token

If you wrap the statement in **braces**, it works.

```
const obj = {
  a: 1,
  b: '2'
};

let a, b;

({ a, b } = obj);
```

@oliverjumpertz

JS

Destructure Function Parameters To Increase Readability For Users

Too many parameters make a function very difficult to use for users.

It's hard to tell which parameter is exactly what.

```
const myFunction = (one, two, three, four, five) => {
 // processing
};

myFunction(1, 2, 3, '4', 5);
```

Destructuring the parameters allows your users to pass a single object where each parameter is **named**.

This can drastically increase readability.

```
const myFunction = ({one, two, three, four, five}) => {
 // processing
};

myFunction({
  one: 1,
  two: 2,
  three: 3,
  four: '4',
  five: 5
});
```

@oliverjumpertz

JS

Make Arguments Required By Using Default Parameters

This is just an arrow-function.

Whenever it is called, it simply throws an error.

```
const required = (parameterName) => {
  throw new Error(
    `${parameterName} is a required parameter`
  );
};
```

Use like this.

If you assign the arrow function as a **default parameter**, you throw an error whenever someone leaves out an otherwise required parameter.

```
const myFunction = (argument = required('argument')) => {
  // processing...
};

myFunction();
```

Throws because the required argument is not provided.

@oliverjumpertz

JS

Anatomy Of A For Loop

The **loop variable** needs to be **mutable**.

This is the **loop variable**.
It is usually modified on each iteration.

This operation is performed **after** each iteration.

```
for (let loopVariable = startValue; loopVariable < endValue; loopVariable++) {
  someArray[loopVariable];

  doSomethingWith(loopVariable);
}
```

Everything within the curly braces is performed **once** for each iteration.

This is the value the loop variable is set to in the **beginning**.

This check is performed **before** each iteration.

As soon as this check yields ***false***, **the loop ends**.

@oliverjumpertz

JS

Anatomy Of A While Loop

This condition is checked **before** each step of the iteration.

If it yields ***false***, the loop **ends**, even directly in the beginning.

You can also write a short condition like ***true*** or use a variable.
JavaScript basically calls the Boolean constructor with your variable like: ***Boolean(variable)***

```
while (condition === true) {
  doWork();
}
```

Everything **within** the **curly braces** is performed **once** for each **iteration**.

If the condition checked **never** yields ***false***, the loop continues **forever**.

@oliverjumpertz

JS

Use The Performance API To Measure Execution Times

Using ***Date*** to measure execution times is not very precise.

```
const begin = new Date();

someOperation();

const end = new Date();

const ellapsedTime = end.getTime() - begin.getTime();
```

The ***performance API*** is made to measure time with **microsecond** precision.

```
const begin = performance.now();

someOperation();

const end = performance.now();

const ellapsedTime = end - begin;
```

There is **one drawback**, though:
All major browsers have reduced precision to **prevent fingerprinting**.
Precision is **unaffected** in **Node**.

@oliverjumpertz

JS

Anatomy Of A Do-While Loop

Everything within the curly braces is performed **at least one time** until the condition yields ***false***.

You can also write a short condition like ***true*** or use a variable.

```
do {
  doSomething();
} while (condition === true);
```

This condition is checked **after** each **iteration**.
If it yields ***false***, the loop **ends**, but but only after it finished at least the first iteration.

If the condition checked **never** yields ***false***, the loop continues **forever**.

@oliverjumpertz

JS