

Prevent An Object Key From Being Enumerable

An object key set to ***undefined*** is still **enumerable**.

If anyone iterates over that object's **keys**, they will get the key and read ***undefined*** as its value.



```
● ● ●  
const a = 1;  
const b = undefined;  
  
const obj = {  
  a,  
  b  
};  
  
obj.hasOwnProperty('b'); // => true
```



@oliverjumpertz

If you want to prevent a property set to ***undefined*** from being enumerable, you can spread the property conditionally.



```
● ● ●  
const a = 1;  
const b = undefined;  
  
const obj = {  
  a,  
  ...(b ? { b } : {})  
};  
  
obj.hasOwnProperty('b'); // => false
```

JS

A Memoized Fibonacci Function

memo is especially meant for internal recursive calls.

We calculate the nth fibonacci number.
The 0th and 1st number are both 1.

```
● ● ●  
const fibonacci = (n, memo = {}) => {  
  if (memo[n]) {  
    return memo[n];  
  }  
  
  if (n <= 1) {  
    return 1;  
  }  
  
  return (memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo));  
};
```

If we already calculated a fibonacci number **once**, we don't need to calculate it again because it is already **memoized**.

This expression does a lot in one line. But simplified:
Calculate the n - 1st and n - 2nd **fibonacci number** and **memoize** it. The result is additionally returned.



How To Format The Output Of `JSON.stringify`

```
● ● ●  
const formattedJson = JSON.stringify(  
  {  
    handle: '@oliverjumpertz',  
    mission: 'help everyone to become a better dev',  
  },  
  null,  
  '  '  
);  
  
console.log(formattedJson);  
// => prints  
// {  
//   "handle": "@oliverjumpertz",  
//   "mission": "help everyone to become a better dev"  
// }
```

The **third argument** of `JSON.stringify` controls how to indent the output. It can either be a **string** or a **number**.



How To Create A Cancelable Promise Delay

```
const delay = (delay, value) => {
  let timeout;
  let _reject = null;
  const promise = new Promise((resolve, reject) => {
    _reject = reject;
    timeout = setTimeout(resolve, delay, value);
  });
  return {
    promise,
    cancel() {
      if (!timeout) {
        return;
      }
      clearTimeout(timeout);
      timeout = null;
      _reject();
      _reject = null;
    }
  };
};
```

You could also accept a **function** to directly run when the **delay** is **over**.



@oliverjumpertz



Combine a **Promise** with **setTimeout** and return an **object** that stores all information necessary to **cancel** the latter again.

```
const delayed = delay(5000, 'value');

delayed
  .promise
  .then((value) => console.log(value))
  .catch(() => console.error('Promise rejected'));

delayed.cancel();
```

JS

A Lazy Range Function

This **generator function** creates a lazy sequence of numbers.

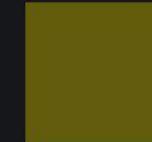
You can't use your usual array methods with it, but it is more **memory-efficient**, especially for very large sequences.



```
function* range(begin, endExclusive, step = 1) {  
  let current = begin;  
  while (current < endExclusive) {  
    yield current;  
    current += step;  
  }  
}
```



@oliverjumpertz



```
for (const value of range(0, 5)) {  
  console.log(value); // => prints: 0, 1, 2, 3, 4  
}  
  
for (const value of range(0, 10, 2)) {  
  console.log(value); // => prints: 0, 2, 4, 6, 8  
}
```

You can use the **generator function** with a **for..of loop**.

Only **one** value is kept **in memory** at a time.

JS

A Python-Like Range Function

This Python-like **range function** creates an **eager** array, **pre-filled** with the values requested.



```
const range = (begin, endExclusive, step = 1) => {
  return Array.from(
    { length: (endExclusive - begin) / step },
    (_, i) => begin + i * step
  );
};
```



The result is an **eagerly filled array**.
Huge ranges will thus consume **a lot** of memory.

```
range(0, 3).forEach((i) => console.log(i));
// => prints: 0, 1, 2
```

```
range(2, 10, 2).forEach((i) => console.log(i));
// => prints: 2, 4, 6, 8
```



Sum Up Array Values With Reduce

The actual goal of **reduce** is to create a **single value** out of your array, although you can also use it differently.

This is the **current value**.
It **changes each time** reduce iterates further.

```
● ● ●  
const array = [1, 2, 3, 4, 5, 6, 7, 8, 9];  
  
const sum = array.reduce((accumulator, current) => accumulator + current, 0);  
// => sum is now 45
```

The accumulator is the result of all previous operations.
In this case the result of summing up all previous values.

This is the initial value.

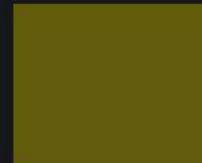


Make Objects Immutable With Freeze

Object.freeze freezes your object and prevents modifications.



```
const obj = {  
    propertyOne: 1,  
    propertyTwo: 'a string'  
};  
  
Object.freeze(obj);
```

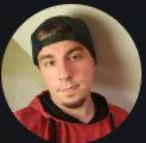


Properties **can't be modified** on a frozen object.



```
// All these throw a TypeError  
  
obj.propertyOne = 2;  
  
delete obj.propertyOne;  
  
Object.defineProperty(obj, 'propertyThree', { value: false });
```

Properties of nested, non-frozen objects **can still be modified**, though.



@oliverjumpertz

JS

Make Arrays Immutable With Freeze

`Object.freeze` freezes your array and prevents modifications.



```
const array = [1, 2, 3];  
  
Object.freeze(array);
```

A frozen array **can't be modified**.

You can't replace entries.

You can't push to it.



```
// All these throw a TypeError now  
  
array[1] = 2;  
  
array.length = 0;  
  
array.push(4);
```



@oliverjumpertz

JS

Infinitely Flatten Any Array

Calling **flat** on an array with the argument **Infinity** leads to the array being flattened until there is only one layer left.



```
const flatten = (...args) => args.flat(Infinity);
```



```
const result = flatten([1, 2, [3, 4], [[5], 6]]);  
// => [1, 2, 3, 4, 5, 6]
```

You can **unpack** and **flatten** any array with it.



Anatomy Of A For-Of Loop

The reference of the **loop variable** can be **immutable**.

It is **reset on each iteration**.

```
for (const value of iterable) {  
    doSomethingWith(value);  
}
```

Everything **within the curly braces** is performed **as often** as the **iterable** has **more values** to process.

An **iterable** is any object that implements the method **[Symbol.iterator]**.

value can contain anything from primitives to objects.

```
...  
object[Symbol.iterator] = function() {  
    return {  
        next: function() {  
            return {  
                value: anyValue,  
                done: true | false  
            };  
        };  
    };  
};
```

As long as **done** is set to **false**, the **for-of loop continues**.

Only as soon as it is set to **true**, the loop will terminate.

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



Anatomy Of An Object Destructuring Assignment

You can give **another name** to the local **variable** the **property** is **extracted** to.

You can access **nested objects** as deep as necessary on the right-hand side.



```
const { propertyToExtract: renamedLocalVariableName = defaultValue } = objectToDestructure.nested;
```

This is the **property** you want to **extract**.

If the property you try to extract is not present or **undefined**, you can set a **fallback value** here.



@oliverjumpertz

JS

Anatomy Of Array.prototype.map

The function you provide is called **once** for each **element** of the original array.

The **order** of the elements the function is called for is the **same as in the original array**.

index is always set to the **index** of the current element within the original array.

If you provide an object or any reference here, referencing **this** within your function will actually point to the **reference provided**.



```
[apple, orange, pear, lemon].map((currentValue, index, array) => putIntoMixer(currentValue), thisOverride);  
// => [juice, juice, juice, juice]
```

The resulting array has the **same length** as the original array.

This is the **value currently** processed.

This is a **reference** to the original array.

You can use it to create **nested loops**.



Short Circuit Evaluation With The Logical AND

This is a boolean condition.

The logical **and** does only return **true** if **both sides** yield **true**.

If the first one is already false, the second one **doesn't need** to be **checked**.



```
condition && doAction();
```



it is actually the same as the code below.

Transpilers and minimizers use this to reduce code size.



```
if (condition) {  
    doAction();  
}
```

You **should not replace** all your ***if-statements*** with this.

Readability can suffer a lot.

It's a **good** to know.



Short Circuit Evaluation With The Logical OR

This is a boolean condition.

The logical or does return **true**

as soon as one **sides** yield **true**.

If the first one is false, the second one
needs to be **checked**.



```
condition || doAction();
```

You **should not replace** all your
if-statements with this.

Readability can suffer a lot.

It's a **good** to know.



it is actually the same as the code
below.

Transpilers and minimizers
use this to **reduce code size**.



```
if (!condition) {  
    doAction();  
}
```

Use Array Spreading To Keep Your Arrays Immutable

Pushing to an **array** is perfectly fine but mutates the **original array**.

This can have **unforeseen side-effects** if you reuse the array often.



```
const array = [1, 2];  
  
array.push(3);  
  
// array => [1, 2, 3]
```



Copying the **array** with the **spread operator** and appending to the new one keeps the **original array** intact.

It supports **immutability** and prevents unforeseen **side-effects**.

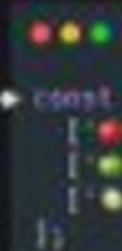


```
const array = [1, 2];  
  
const newArray = [...array, 3];  
  
// array => [1, 2]  
// newArray => [1, 2, 3]
```



Anatomy of Array.prototype.flatMap

fruits is an **array** that contains more **arrays** with two elements each.



This is the current **element** of the iteration.

```
const fruits = [  
  [ 'red', 'green' ],  
  [ 'yellow', 'blue' ],  
  [ 'orange', 'purple' ],  
  [ 'pink', 'brown' ]  
];
```

This is the current **index** of the iteration.

1

This is the **reference of the array** you are iterating over.

```
const shakes = fruits.flatMap((currentFruits, index, fruitsArray) => {  
  // index 0, then 1, then 2  
  // fruitsArray points to fruits  
  // currentFruits is always an array of two elements in this case  
  // we return an array containing n shakes for n elements passed  
  return mix(currentFruits);  
}, thisOverride);
```

// so shakes now looks like this: [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]

The overall goal of **flatMap** is to **unpack** multiple arrays returned by the function into one layer less.

You can set a reference that becomes **this** within the function.



Anatomy of `Array.prototype.forEach`

```
const numberArray = [1, 2, 3, 4, 5];  
  
numberArray.forEach(currentElement, index, originalNumberArray) => {  
  doSomethingWith(currentElement, index, originalNumberArray);  
}, thisOverride);
```

This is the `current element` of the iteration.

This is the `current index` of the iteration.

You can set a reference that becomes `this` within the function.

This is the `reference of the array` you are iterating over.

The idea of `forEach` is to give you a way to iterate over an `array` and perform side-effects.

