# Vanilla JS:
# 11 ways of assigning the same value to multiple variables in a single statement

By Dan Cortes

# What

Assigning the same value...

```
const val = Symbol();
const x = val;
const y = val;
const z = val;
console.log(typeof x === 'symbol' && x === y && y === z); // true
```

...but in a single outermost statement.

```
const ??? = ???;
```

# Why

1. JS lets you do weird things
2. Doing weird things is fun
3. Maybe learn some language features?

- Note that this is not an endorsement of using these in production code 🤓

# Let's dive in 

## 1. The old school way

The most succinct way I can think of:

```
const x = y = z = Symbol();

console.log(typeof x === 'symbol' && x === y && y === z); // true
```

But this only works sometimes. Does anyone know why?

# 1. The old school way (cont.)

Strict mode (which is enabled by default in modules)
doesn't let you assign values to undeclared variables.

```
(function() {
    'use strict';
    // Uncaught ReferenceError: z is not defined
    const x = y = z = Symbol();
    console.log(typeof x === 'symbol' && x === y && y === z);
})();
```

...so how do can do this succinctly and in strict mode?

Note: The rest of the examples will be strict mode friendly.

## 2. Binding lists

You can declare multiple variables using a binding list, and assign the previously assigned variables to the later variables.

```
const x = Symbol(), y = x, z = y;

console.log(typeof x === 'symbol' && x === y && y === z); // true
```

Neat, eh? How else can we do it?

## 3. Object destructuring

We can use object destructuring assignment and rename a single property multiple times.

```
const { data: x, data: y, data: z } = { data: Symbol() };

console.log(typeof x === 'symbol' && x === y && y === z); // true
```

...but what if we don't want to rename the property?

# 4. Object destructuring of self-referential static class members

We can lean into self referential static class members to assign the same value to multiple static properties.

```
const { x, y, z } = class Temp {
    static value = Symbol();
    static x = Temp.value;
    static y = Temp.value;
    static z = Temp.value;
};

console.log(typeof x === 'symbol' && x === y && y === z); // true
```

...can we do it without classes?

## 5. Object destructuring IIFE

We can use an IIFE (immediately invoked function expression) to build an object with multiple properties that all have the same value.

```
const { x, y, z } = (function() {
  const value = Symbol();
  return {
    x: value,
    y: value,
    z: value
  };
})();

console.log(typeof x === 'symbol' && x === y && y === z); // true
```

That's kinda weird, can we make weirder?

## 6. Proxied object

We can use a proxy that returns the same property from the object regardless of what property was being accessed.

```
const { x, y, z } = new Proxy({ data: Symbol() }, {
  get(target) {
    return target.data;
  }
});

console.log(typeof x === 'symbol' && x === y && y === z); // true
```

## Question: Proxied array?

Question: Will the previous implementation work with array destructuring assignment?

```
const [x, y, z] = { /* Same implementation as previous slide */ }
```

## Answer: Proxied array?

Nope! While arrays are just objects, array destructuring assignment
works on the iterable protocol, which the previous example did not implement.

```
// Uncaught TypeError: Proxy is not a function or its return value is not iterable
const [x, y, z] = new Proxy({ data: Symbol() }, {
  get(target) {
    return target.data;
  }
});
```

...so how can we assign the same value to multiple variables
in a single declaration using array destructuring?

# 7. Array destructuring IIFE

We can once again lean on IIFEs (which are basically cheating 🤪).

```
const [x, y, z] = (function() {
  const value = Symbol();
  return [value, value, value];
})();

console.log(typeof x === 'symbol' && x === y && y === z); // true
```

...can we do this without without repeating the value three times?

## 8. Destructuring a recursively built array

Still an IIFE, but this time we're using recursion to build the array.

```
const [x, y, z] = (function repeat(times, value, _arr = []){
  return times === 0
    ? _arr
    : repeat(times - 1, value, [..._arr, value]);
})(3, Symbol());

console.log(typeof x === 'symbol' && x === y && y === z); // true
```

## 9. Array destructuring IIGFE

Are `IIGFE`s (immediately invoked generator function expression) a thing? They are now!

Func fact, the return value of a generator function is iterable.

```
const [x, y, z] = (function*() {
  const value = Symbol();
  while (true) {
    yield value;
  }
})();

console.log(typeof x === 'symbol' && x === y && y === z); // true
```

...can we get weird with iterators?

# 10. Roll your own iterator

If your `*` key is broken, you can always make your own iterator.

```
const [x, y, z] = {
  value: Symbol(),
  [Symbol.iterator]() {
    return this;
  },
  next() {
    return {
      done: false,
      value: this.value
    }
  }
};

console.log(typeof x === 'symbol' && x === y && y === z); // true
```

...but can we make this even weirder?

# 11. Proxy as iterable

You can recreate the iterator protocol in a proxy.

```
const [x, y, z] = new Proxy({ data: Symbol() }, {
  get(target, prop, receiver) {
    if (prop === Symbol.iterator) {
      return () => receiver;
    }

    if (prop === 'next') {
      return () => ({
        done: false,
        value: target.data
      });
    }
  }
})

console.log(typeof x === 'symbol' && x === y && y === z); // true
```

## Thanks for listening!

- Questions?
- Comments?
- Can you think of any other ways to do this?