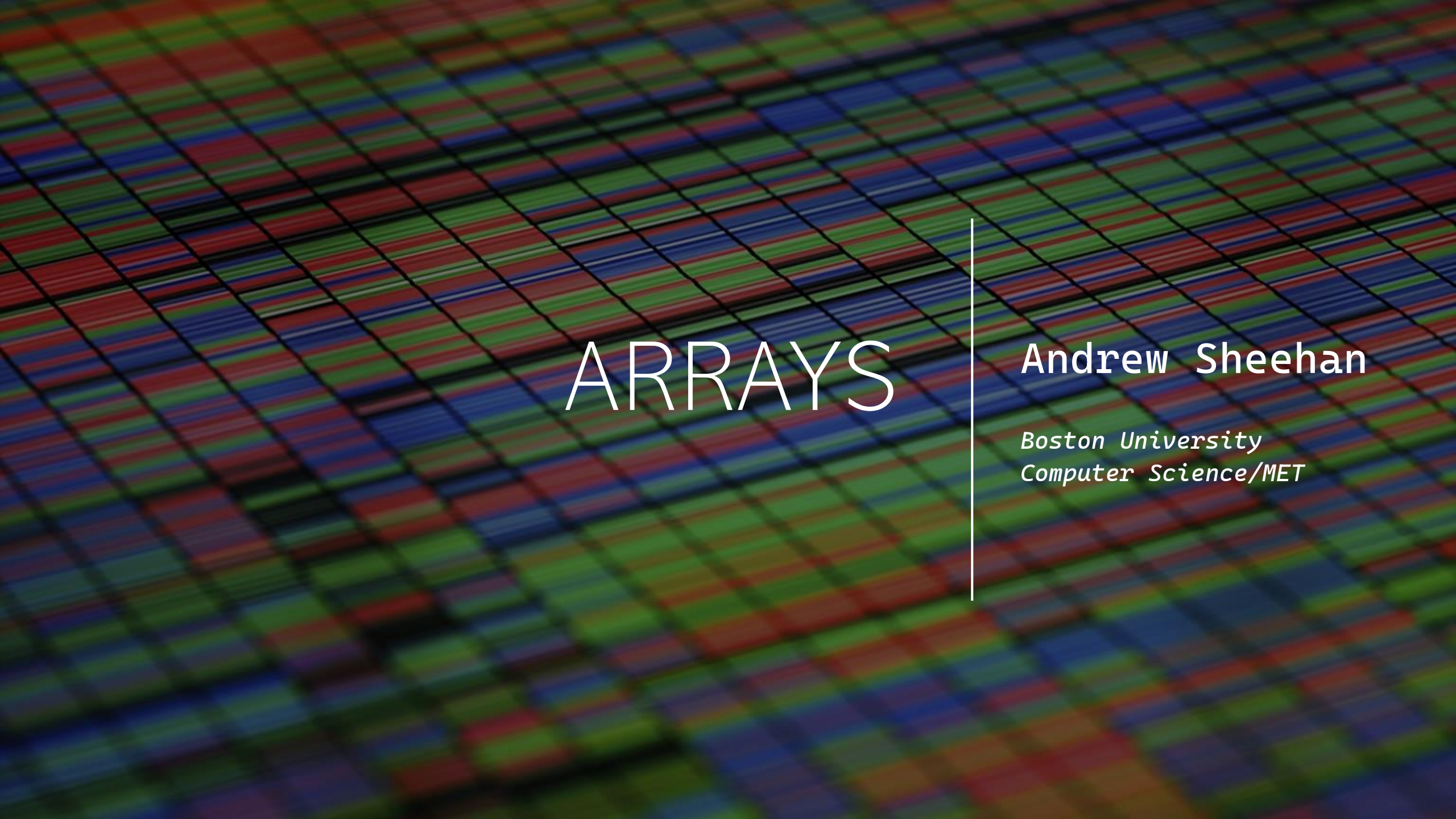


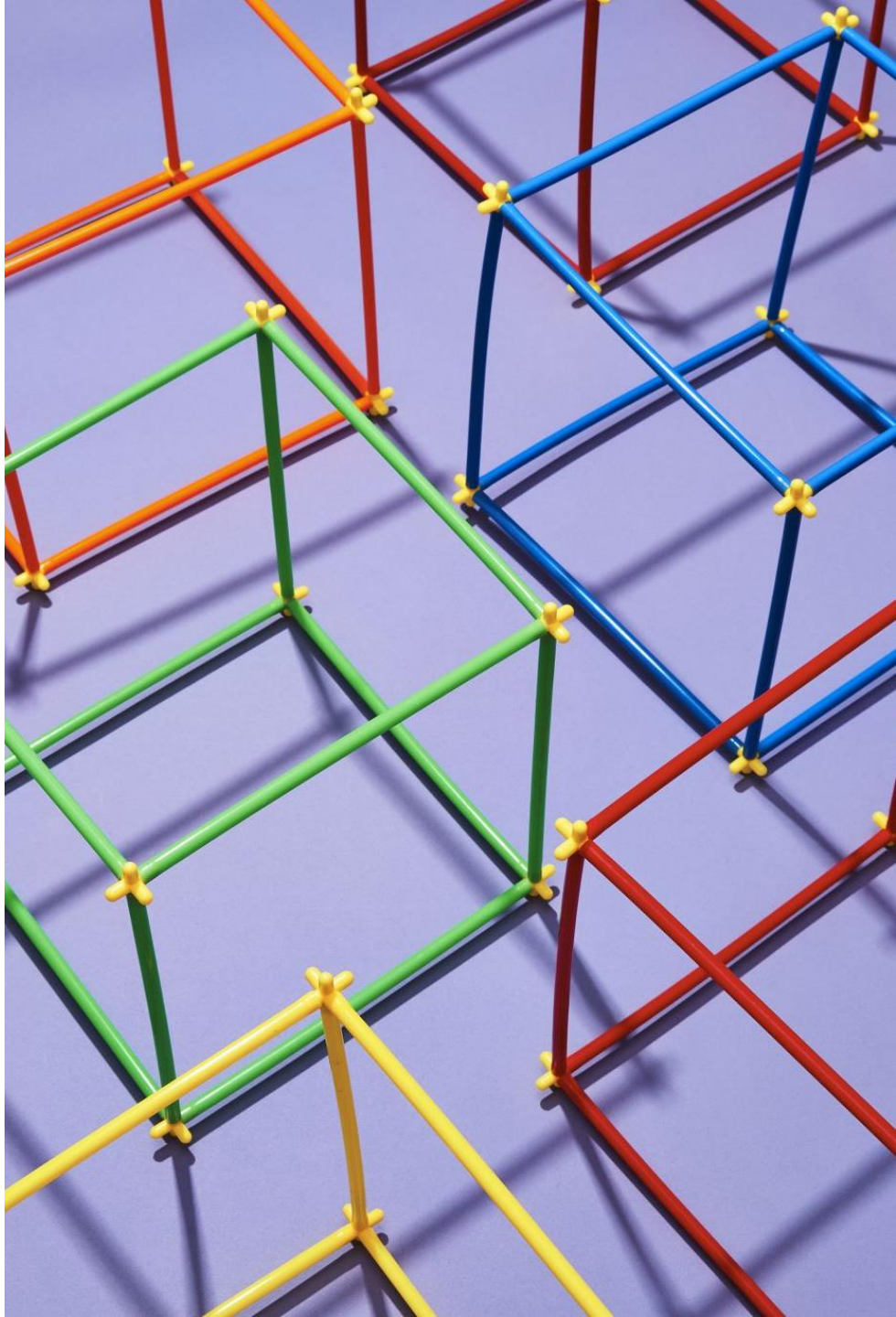
ARRAYS

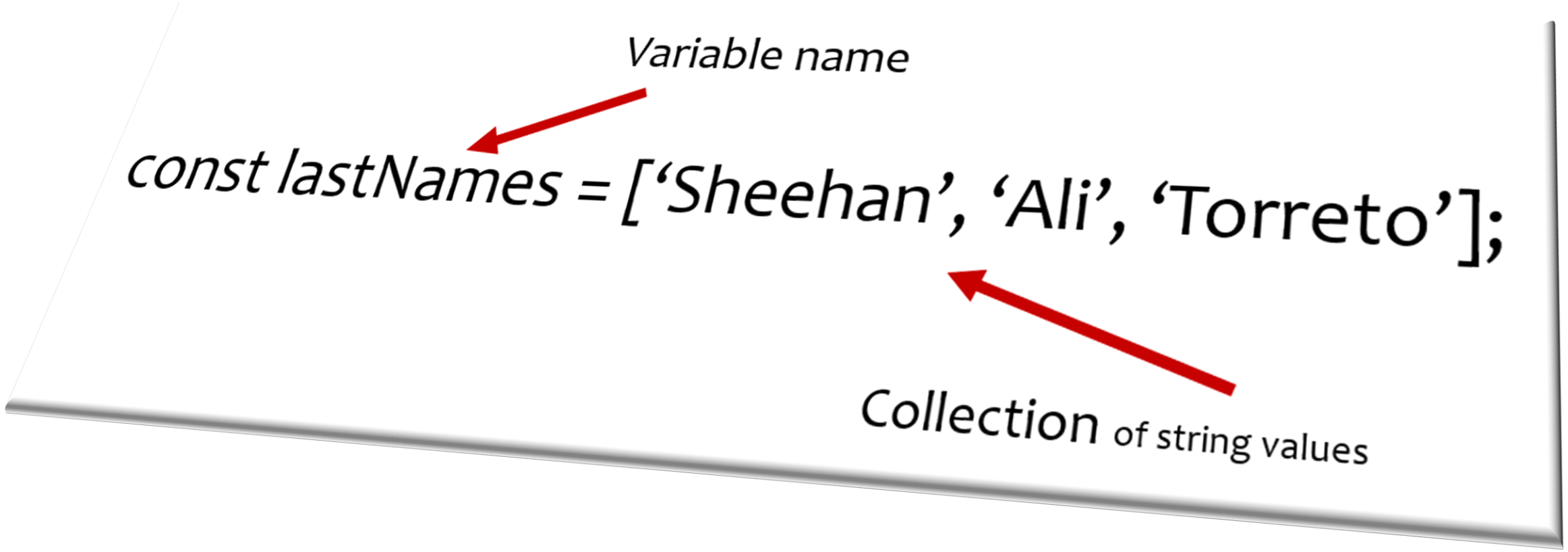
Andrew Sheehan

Boston University
Computer Science/MET

DATA STRUCTURES

Structures/Objects - allowing us to store zero to many things.





ARRAYS

CAN CONTAIN ANY TYPE

ARRAY SYNTAX

```
var ages = new Array();
```

Do not use this type of syntax.

ARRAY SYNTAX

```
const lastNames = [  ];
```

Use brackets.

You can have them close together
or have one space between them.

ARRAY SYNTAX

```
const brothers = [
  'Andrew',
  'Joseph',
  'Casey
];
```

Space.

You can have as much
space between your values.

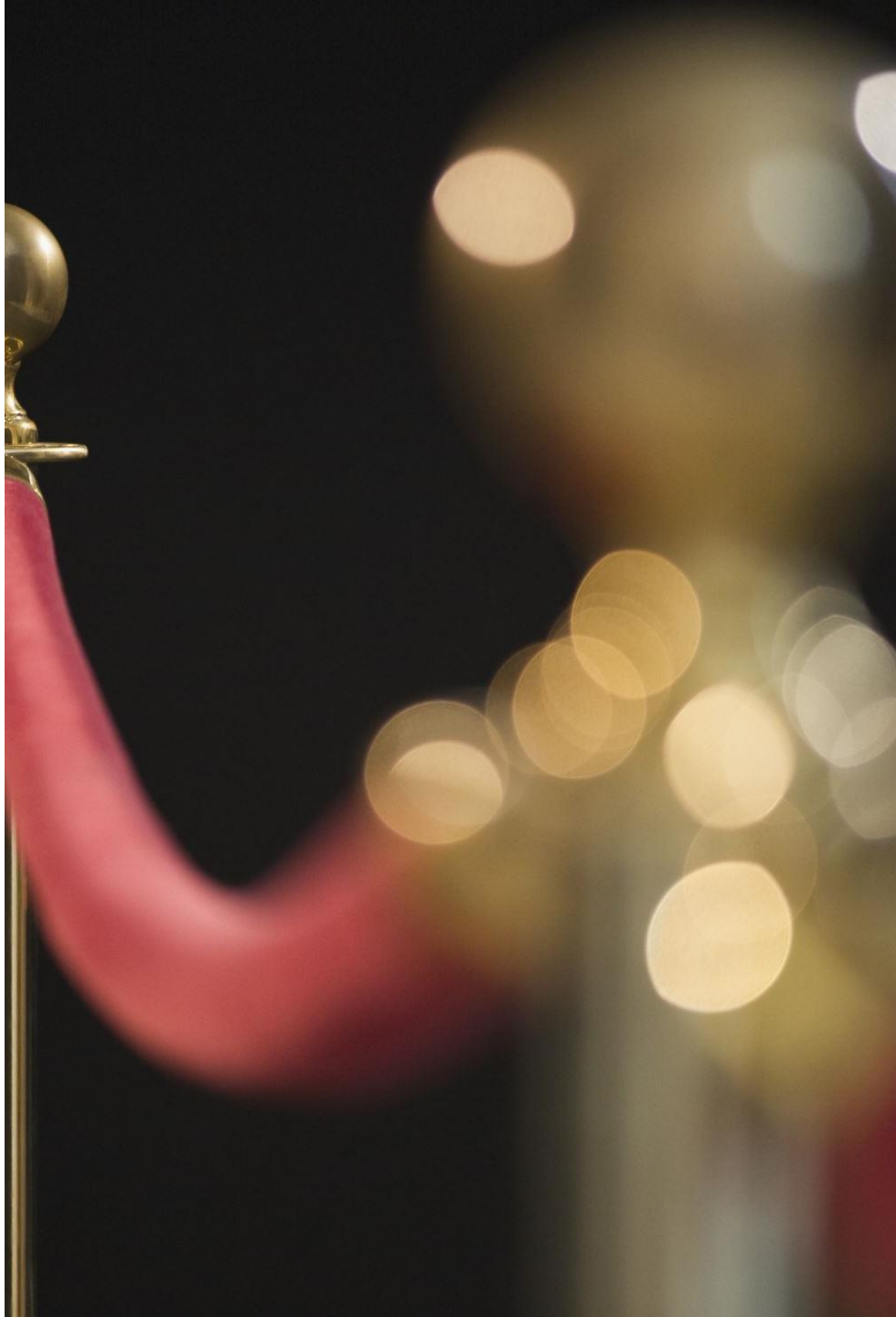
It does not matter.

```
const lastNames = [ 'Sheehan', 'Smith' ];

lastNames.forEach( (item, index) => {
    console.log(`${item}, ${index}`);
});
```

ITERATION: FOREACH()

INDICES START AT 0.

```
const cities = [
  "Boston", "Denver", "Rochester",
  "Miami" ];

const denver = cities[1];
```



ARRAY LENGTH

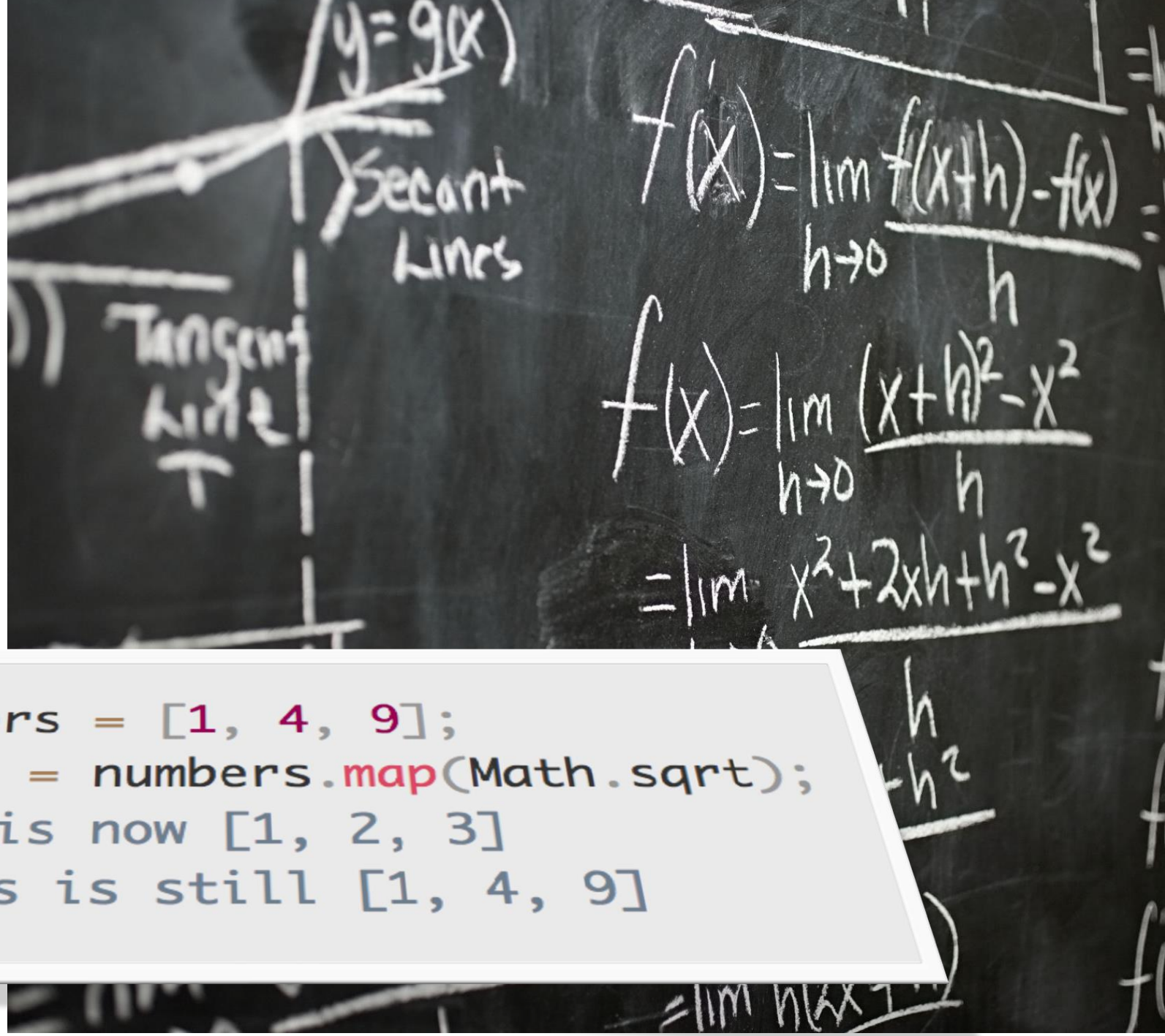
Remember, length is NOT a function.

ANOTHER EXAMPLE

```
1  var a = ['a', 'b', 'c'];
2
3  a.forEach(function(element) {
4      console.log(element);
5  });
```

MAP()

BUILT-IN ARRAY FUNCTION




```
var numbers = [1, 4, 9];
var roots = numbers.map(Math.sqrt);
// roots is now [1, 2, 3]
// numbers is still [1, 4, 9]
```

MAP()

Creates a new array

Based on what you do with the original

```
1  var numbers = [1, 5, 10, 15];
2  var doubles = numbers.map(function(x) {
3      return x * 2;
4  });
5  // doubles is now [2, 10, 20, 30]
6  // numbers is still [1, 5, 10, 15]
7
8  var numbers = [1, 4, 9];
9  var roots = numbers.map(Math.sqrt);
10 // roots is now [1, 2, 3]
11 // numbers is still [1, 4, 9]
```

ARROW FUNCTIONS

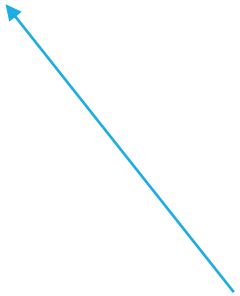
USED WITH ARRAYS

```
8       let numbers = [1,2,4,8,16];
9       let doubles = numbers.map((value, index, arr) => {
10          return index === 0 ? -1 : value * 2;
11      });
12      doubles.forEach((value, index) => {
13          if ( index === 0 ) {
14              console.info("zeroth doesn't matter.");
15          } else {
16              console.info("doubled is: " + value);
17          }
18      });
```

REDUCE()
to a single value

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue

// 1 + 2 + 3 + 4
console.log(array1.reduce(reducer));
// expected output: 10

// 5 + 1 + 2 + 3 + 4
console.log(array1.reduce(reducer, 5));
// expected output: 15
```

```
[0,1,2,3,4].reduce((stor, item) => {
  stor + item
}
```



Your **reducer** function's returned value is assigned to the accumulator, whose value is remembered across each iteration throughout the array and ultimately becomes the final, single resulting value.

THE ACCUMULATOR

REDUCE EXAMPLES

```
const numbers = [175, 50, 25];
document.getElementById("demo").innerHTML = numbers.reduce(myFunc);

function myFunc(total, num) {
 return total - num;
}
```

```
const numbers = [15.5, 2.3, 1.1, 4.7];

document.getElementById("demo").innerHTML = numbers.reduce(getSum, 0);

function getSum(total, num) {
 return total + Math.round(num);
}
```

PUSH()
ADD TO THE END

```
var fruits = ['Apple', 'Banana'];

var newLength = fruits.push('Orange');

// ["Apple", "Banana", "Orange"]
```

KEYS()

KEYS()

keys() will return an iterator that contains all the keys in the array.

```
1  var arr = ['a', 'b', 'c'];
2  console.log(Object.keys(arr)); // console: ['0', '1', '2']
3
4  // array like object
5  var obj = { 0: 'a', 1: 'b', 2: 'c' };
6  console.log(Object.keys(obj)); // console: ['0', '1', '2']
7
8  // array like object with random key ordering
9  var anObj = { 100: 'a', 2: 'b', 7: 'c' };
10 console.log(Object.keys(anObj)); // ['2', '7', '100']
11
12 // getFoo is property which isn't enumerable
13 var myObj = Object.create({}, {
14   getFoo: {
15     value: function () { return this.foo; }
16   }
17 });
18 myObj.foo = 1;
19 console.log(Object.keys(myObj)); // console: ['foo']
```

undefined is returned when no match

```
const array1 = ['a', 'b', 'c'];
const iterator = array1.values();

for (const value of iterator) {
  console.log(value); // expected output: "a" "b" "c"
}
```

VALUES()

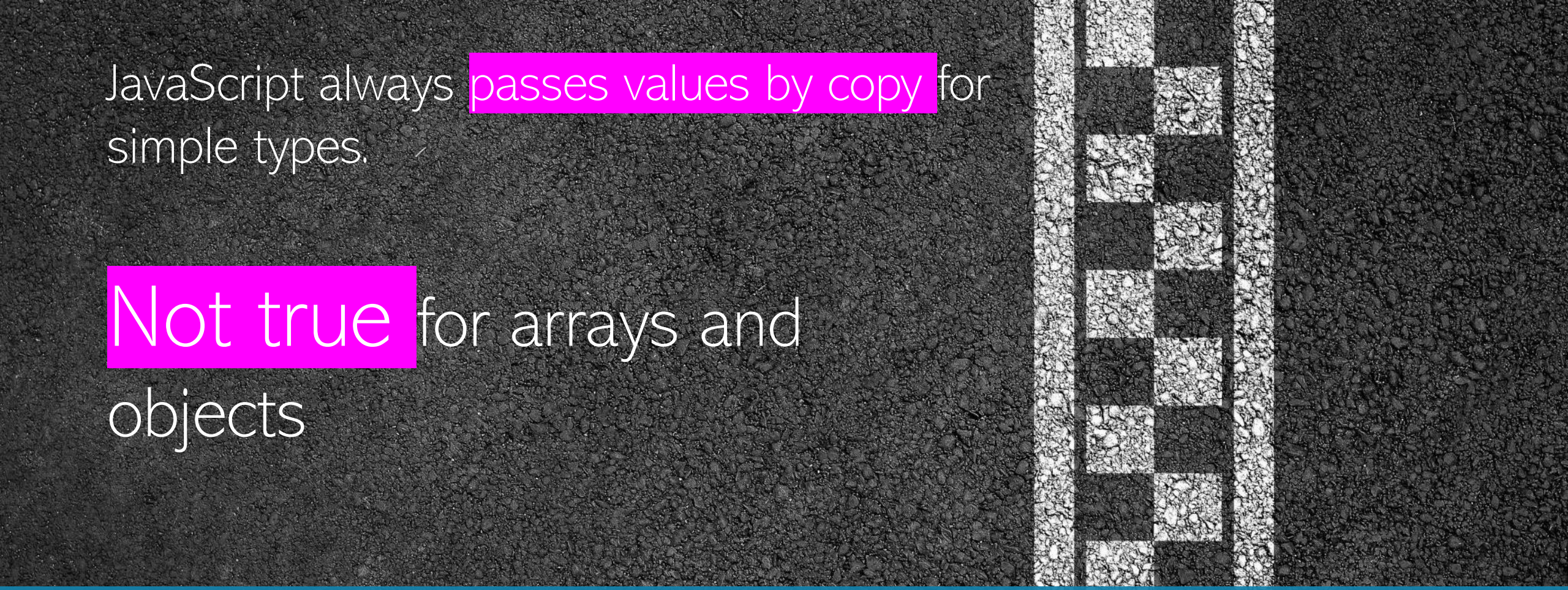
```
let rocks = [‘granite’,‘limestone’,‘quartz’];
const result = rocks.pop(); // result = ‘quartz’


// rocks contains [‘granite’,‘limestone’]
```

POP()

REMOVING AT THE END

SHIFT()

```
const names = [
  'drew', 'joe', 'fran','mike', 'johnny', 'stew'
];

//'drew'
let result = names.shift();
```



ARRAYS ARE PASSED BY REFERENCE

JOINING
WITH A TOKEN

```
1 var a = ['Wind', 'Rain', 'Fire'];
2 a.join();     // 'Wind,Rain,Fire'
3 a.join('-'); // 'Wind-Rain-Fire'
```

FIND()

```
const ages = [3, 10, 18, 20];

document.getElementById("demo").innerHTML = ages.find(checkAge);

function checkAge(age) {
  return age > 18;
}
```

Returns the value of the first element in an array that passes a test

JOINING

```
var parents = ["John", "Mary"];
var children = ["Susan", "Michael"];

var family = parents.concat(children);
```

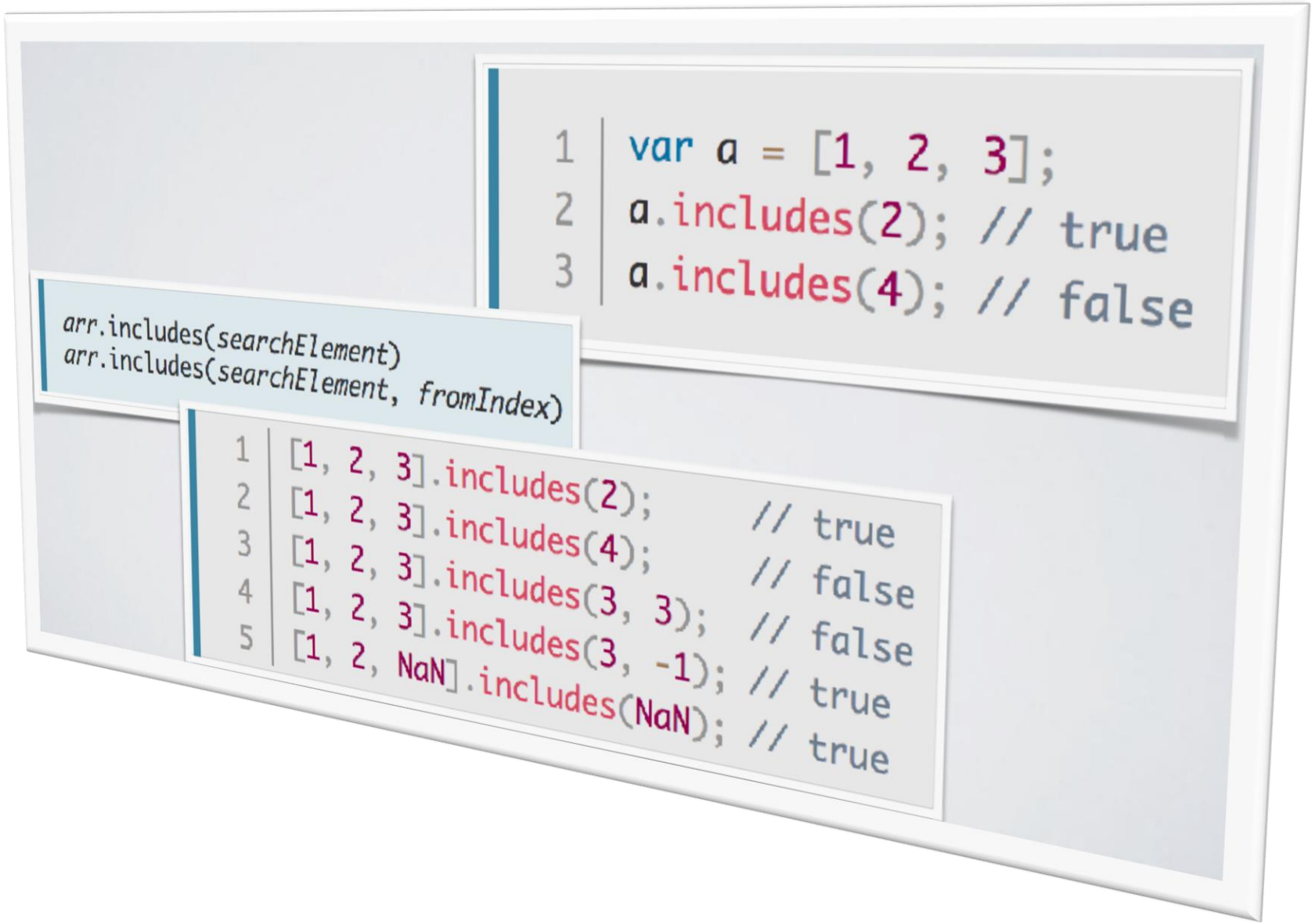
OF()

The of() method creates a new array, with a variable number of args, regardless of type.

```
1  Array.of(7);          // [7]
2  Array.of(1, 2, 3); // [1, 2, 3]
3
4  Array(7);             // [ , , , , , , , ]
5  Array(1, 2, 3);       // [1, 2, 3]
```

```
1  Array.of(1);          // [1]
2  Array.of(1, 2, 3);    // [1, 2, 3]
3  Array.of(undefined); // [undefined]
```

INCLUDES()


1 var a = [1, 2, 3];
2 a.includes(2); // true
3 a.includes(4); // false
arr.includes(searchElement)
arr.includes(searchElement, fromIndex)
1 [1, 2, 3].includes(2); // true
2 [1, 2, 3].includes(4); // false
3 [1, 2, 3].includes(3, 3); // false
4 [1, 2, 3].includes(3, -1); // true
5 [1, 2, NaN].includes(NaN); // true

arrayA.sort()

Sorts the array (numerically or alpha)

arrayB.join(byYourDelimiter)

creates string, separated by your delimiter

arrayC.push(object)

places it on the front of array

arrayD.toString()

Array elements output as string

arrayE.reverse(object)

returns new array in reversed order.

arrayF.slice(from, [to])

returns new array