Swipe

JavaScript **Arrays Methods** In Detail

Swipe

Array Methods

Part 1

1. toString( )
2. join( )
3. pop( )
4. push( )
5. shift( )
6. unshift( )

Part 2

7. delete
8. concat( )
9. sort( )
10. splice( )
11. slice( )
12. reverse( )

Part 3

13. isArray( )
14. indexOf( )
15. lastIndexOf( )
16. find( )
17. findIndex( )
18. includes( )

Part 4

19. entries( )
20. every( )
21. some( )
22. fill( )
22. copyWithin( )
23. valueOf()

Part 5

24. forEach( )
25. map( )
26. filter( )
27. reduce( )
28. reduceRight()
29. from( )

Check All Parts For All Array Methods

Swipe

1. toString()

The *toString( )* Returns a string with array values separated by commas.

And it does not change the original array.

```
array.toString() //Syntax
```

toString( ) Example

```
let bikes = ["yamaha", "Bajaj", "Honda", "TVS"];

console.log(bikes.toString());

// "yamaha,Bajaj,Honda,TVS"
```

Swipe

2. join( )

The *join( )* returns a new string by concatenating all of the elements in an array, separated by commas or a *specified separator string*.

```
array.join(separator) // Syntax
```

JS join( )

Example

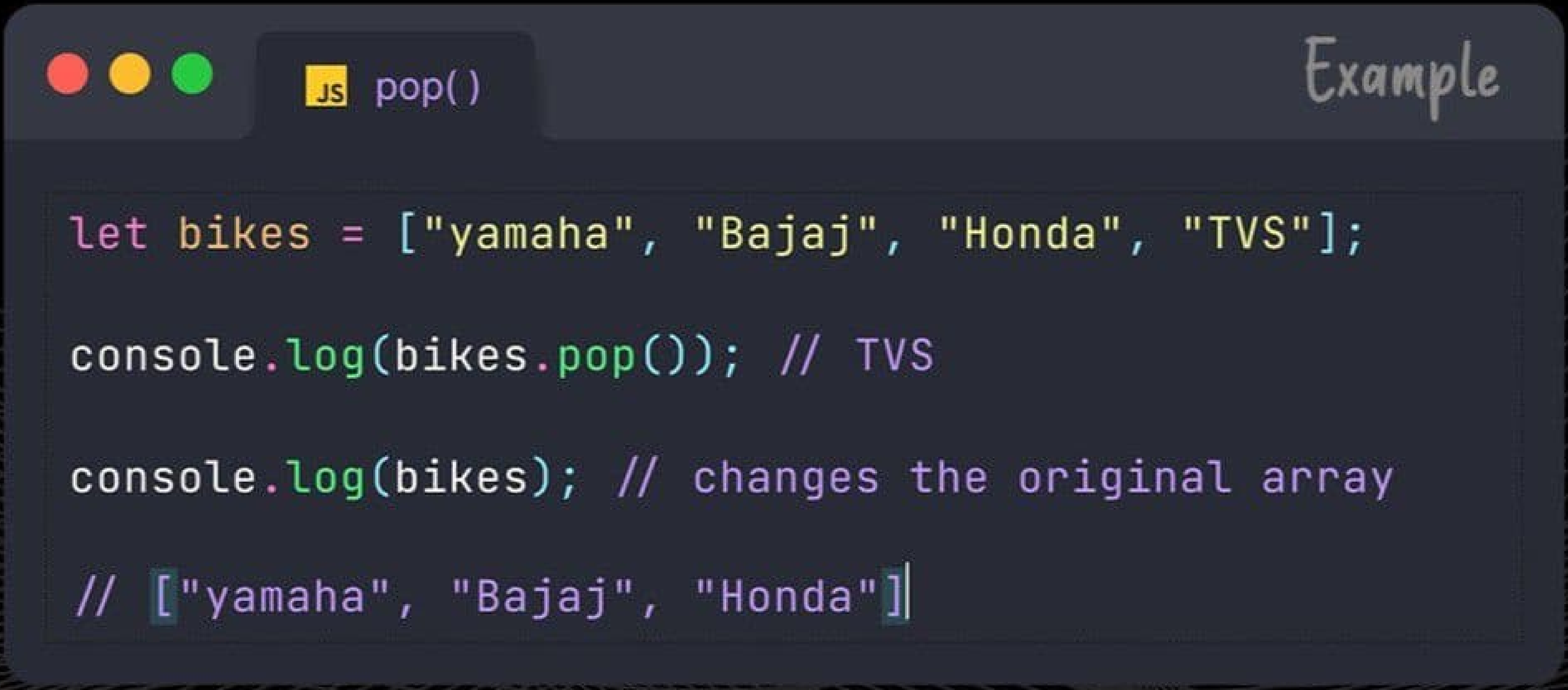
```
let bikes = ["yamaha", "Bajaj", "Honda", "TVS"];

console.log(bikes.join());
// Expected output: "yamaha,Bajaj,Honda,TVS"

console.log(bikes.join(""));
// Expected output: "yamahaBajajHondaTVS"

console.log(bikes.join("-"));
// Expected output: "yamaha-Bajaj-Honda-TVS"
```

3. pop( )

The *pop( )* method removes the *last element of* an array. and returns the removed element.

And this method changes the original array.

```
array.pop() // Syntax
```



```
let bikes = ["yamaha", "Bajaj", "Honda", "TVS"];

console.log(bikes.pop()); // TVS

console.log(bikes); // changes the original array

// ["yamaha", "Bajaj", "Honda"]
```

Swipe

4. push()

The *push( )* adds new items to the *end of an array*, and its changes the length of the array.

returns the new length.

```
array.push(item1, item2, ..., itemX) //Syntax
```

push( )

Example

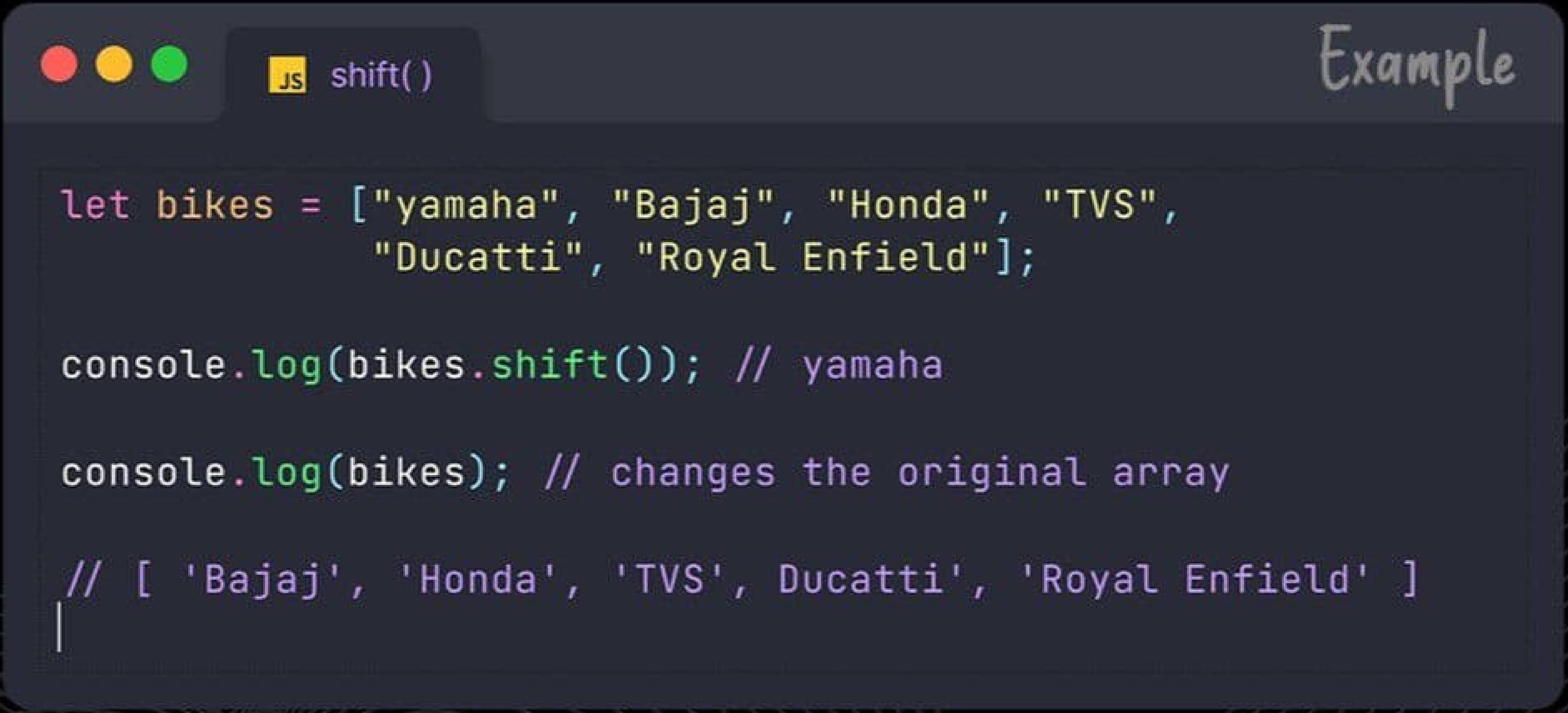
```
let bikes = ["yamaha", "Bajaj", "Honda", "TVS"];

console.log(bikes.push("Ducatti", "Royal Enfield")); // TVS

console.log(bikes); // changes the original array

// [ 'yamaha', 'Bajaj', 'Honda', 'Ducatti', 'Royal Enfield' ]
```

Swipe

5. shift( )

The *shift( )* emoves first element and returns it.

```
array.shift(); //Syntax
```



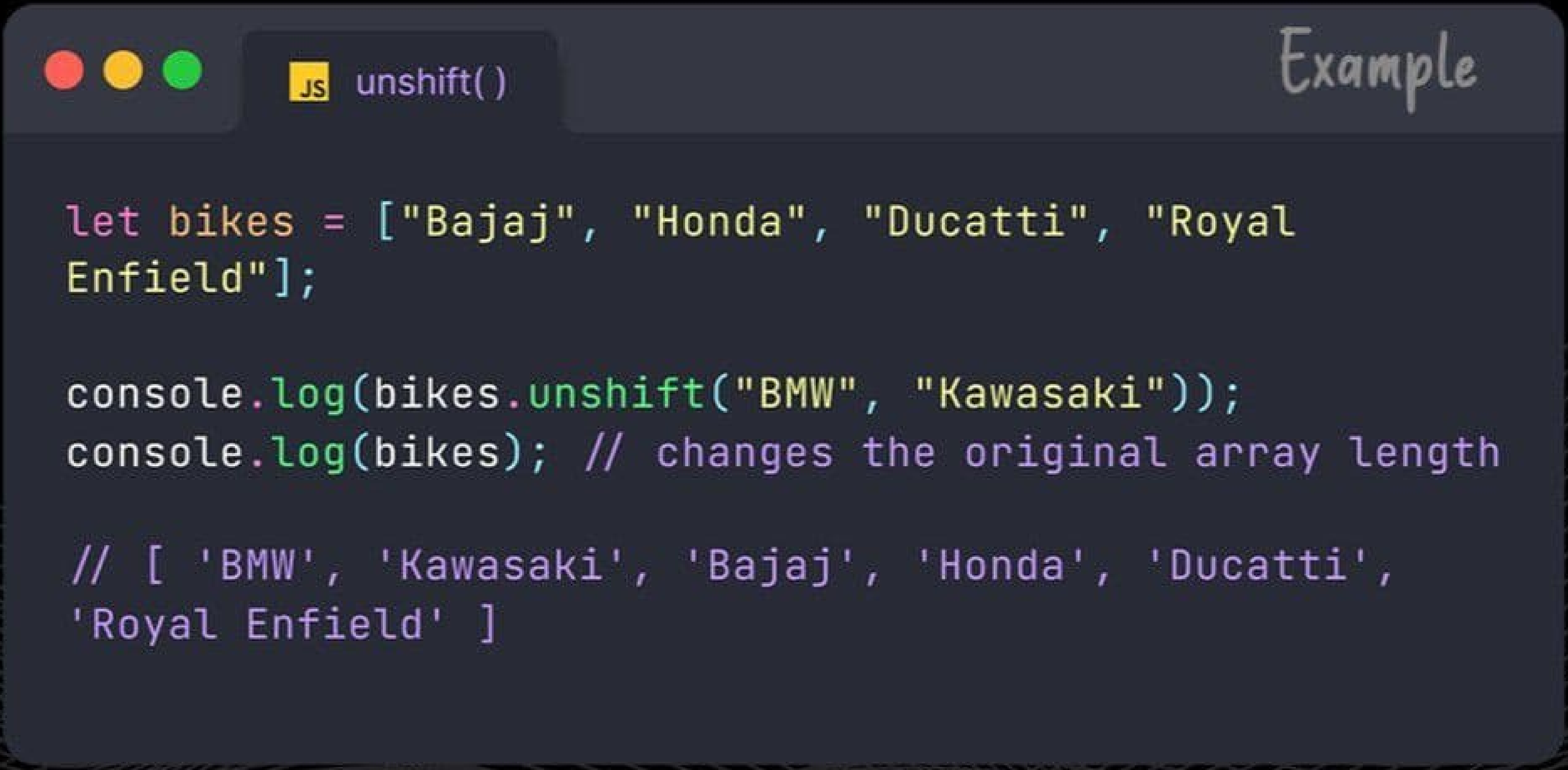
```
let bikes = ["yamaha", "Bajaj", "Honda", "TVS",
              "Ducatti", "Royal Enfield"];

console.log(bikes.shift()); // yamaha

console.log(bikes); // changes the original array

// [ 'Bajaj', 'Honda', 'TVS', Ducatti', 'Royal Enfield' ]
```

Swipe

6. unshift( )

The *unshift( )* adds element to beginning and Retuns new array length.

```
array.unshift(item1, item2, ..., itemX) //Syntax
```



```
let bikes = ["Bajaj", "Honda", "Ducatti", "Royal
Enfield"];

console.log(bikes.unshift("BMW", "Kawasaki"));
console.log(bikes); // changes the original array length

// [ 'BMW', 'Kawasaki', 'Bajaj', 'Honda', 'Ducatti',
'Royal Enfield' ]
```

Swipe

7. delete (operator)

Array elements can be deleted using the JavaScript operator delete,

Using delete leaves *undefined holes* in the array.

```
delete array[index]; // syntax
```

delete operator

Example

```
let fruits = ["banana", "apple", "grapes"];

delete fruits[1]; //

console.log(fruits);

// [ 'banana', <1 empty item>, 'grapes' ]
```

Swipe

8. concat( )

The *concat()* method concatenates (joins) two or more arrays. returns a new array, containing the joined arrays.

This method does not change the existing arrays.

```
array1.concat(array2, array3, ..., arrayX) //Syntax
```

concat( )

Example

```
let ary1 = [1, 2, 3];
let ary2 = [23, 43, 53];
let ary3 = [111, 12];

let ary_new = ary1.concat(ary2, ary3);

console.log(ary_new); // returns new array

// [1, 2, 3, 23, 43, 53, 111, 12]
```

»» Swipe

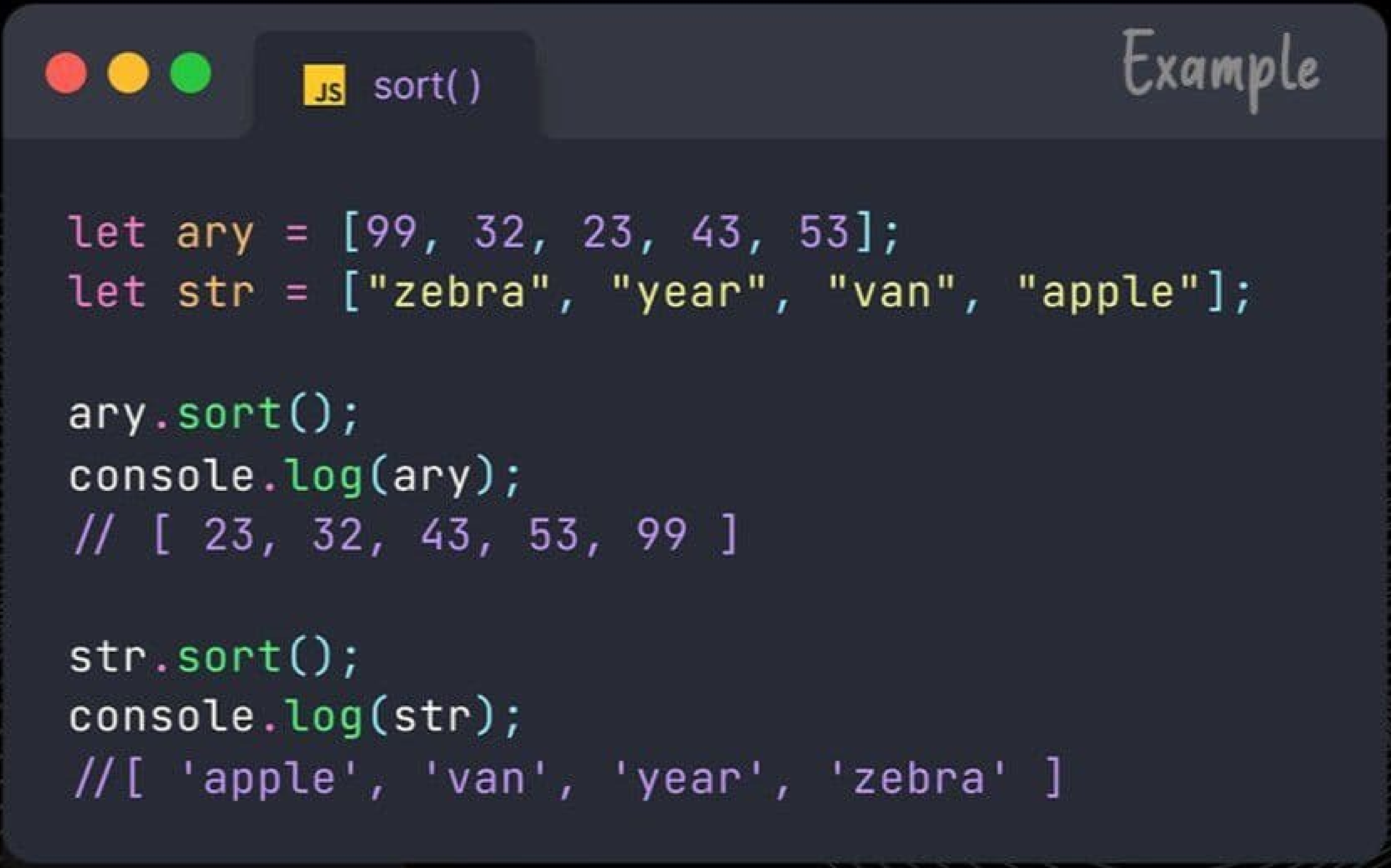
9. sort( )

The *sort( )* sorts the elements of an array, and *overwrites* the original array.

Sorts the elements as strings in *alphabetical* and ascending order.

```
array.sort(compareFunction);  //Syntax
```

sort( ) takes an *optional compare function*.



```
let ary = [99, 32, 23, 43, 53];
let str = ["zebra", "year", "van", "apple"];

ary.sort();
console.log(ary);
// [ 23, 32, 43, 53, 99 ]

str.sort();
console.log(str);
//[ 'apple', 'van', 'year', 'zebra' ]
```

Swipe

10. splice( )

The *splice( )* method adds and/or removes array elements.

The *splice( )* method overwrites the original array.

Syntax

```
array.splice(index, howmany, item1, ....., itemX)
```

Example

splice( )

```
let arry = [99, 32, 23, 43, 53, 100];

arry.splice(2, 3, "a", "b", "c");

console.log(arry);

// [ 99, 32, 'a', 'b', 'c', 100 ]
```

11. slice( )

The *slice( )* slices out a piece from an array, It creates a new array.

```
array.slice(start, end); //Syntaxt
```

slice( )

Example

```
let numbers = [1, 2, 3, 4, 5, 6];

let num2 = numbers.slice(1, 4);

console.log(num2);
//[2, 3, 4]
```

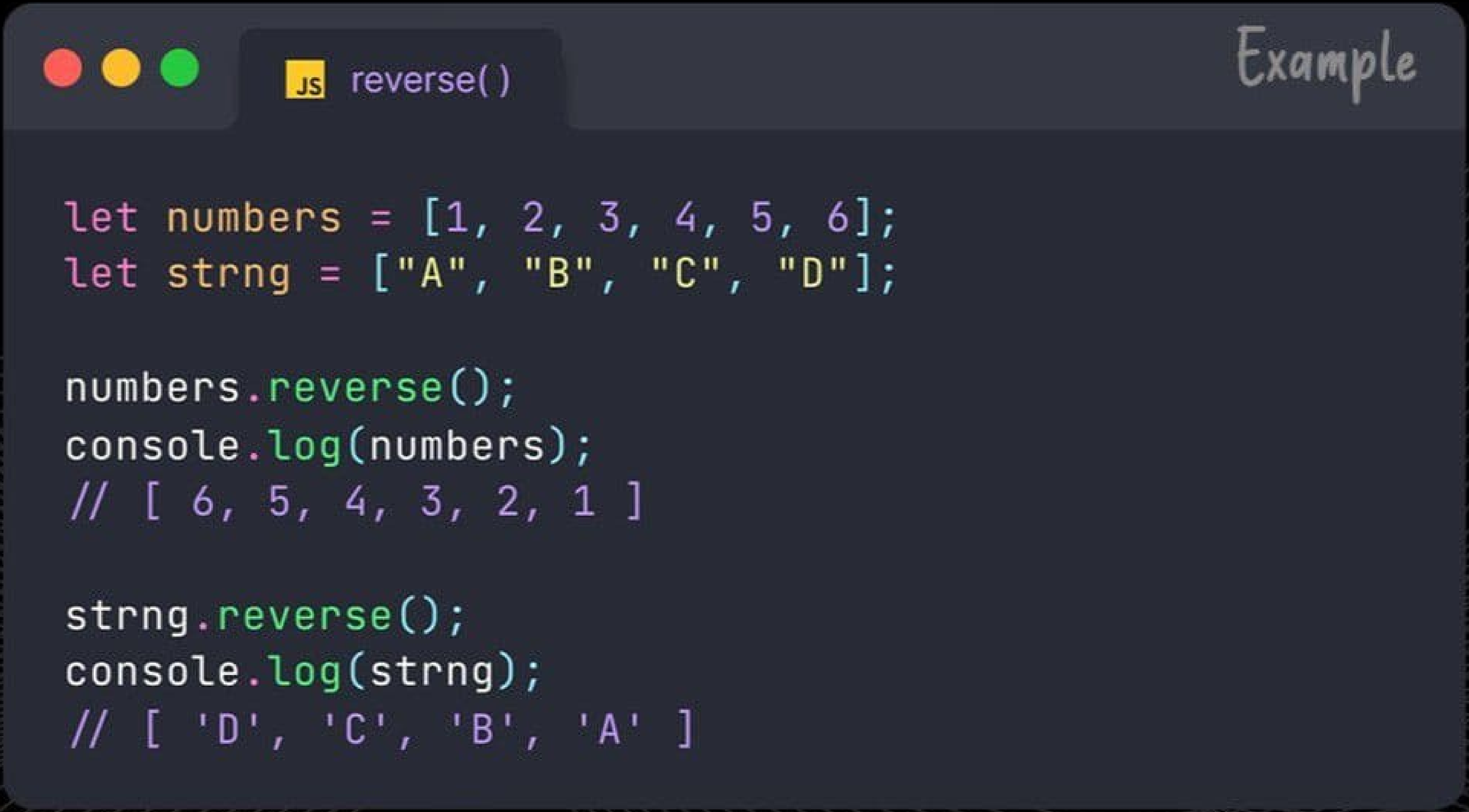
Swipe

12. reverse()

The *reverse( )* method reverses the order of the elements in an array.

This method overwrites the original array.

```
array.reverse(); //Synatx
```



```
let numbers = [1, 2, 3, 4, 5, 6];
let strng = ["A", "B", "C", "D"];

numbers.reverse();
console.log(numbers);
// [ 6, 5, 4, 3, 2, 1 ]

strng.reverse();
console.log(strng);
// [ 'D', 'C', 'B', 'A' ]
```

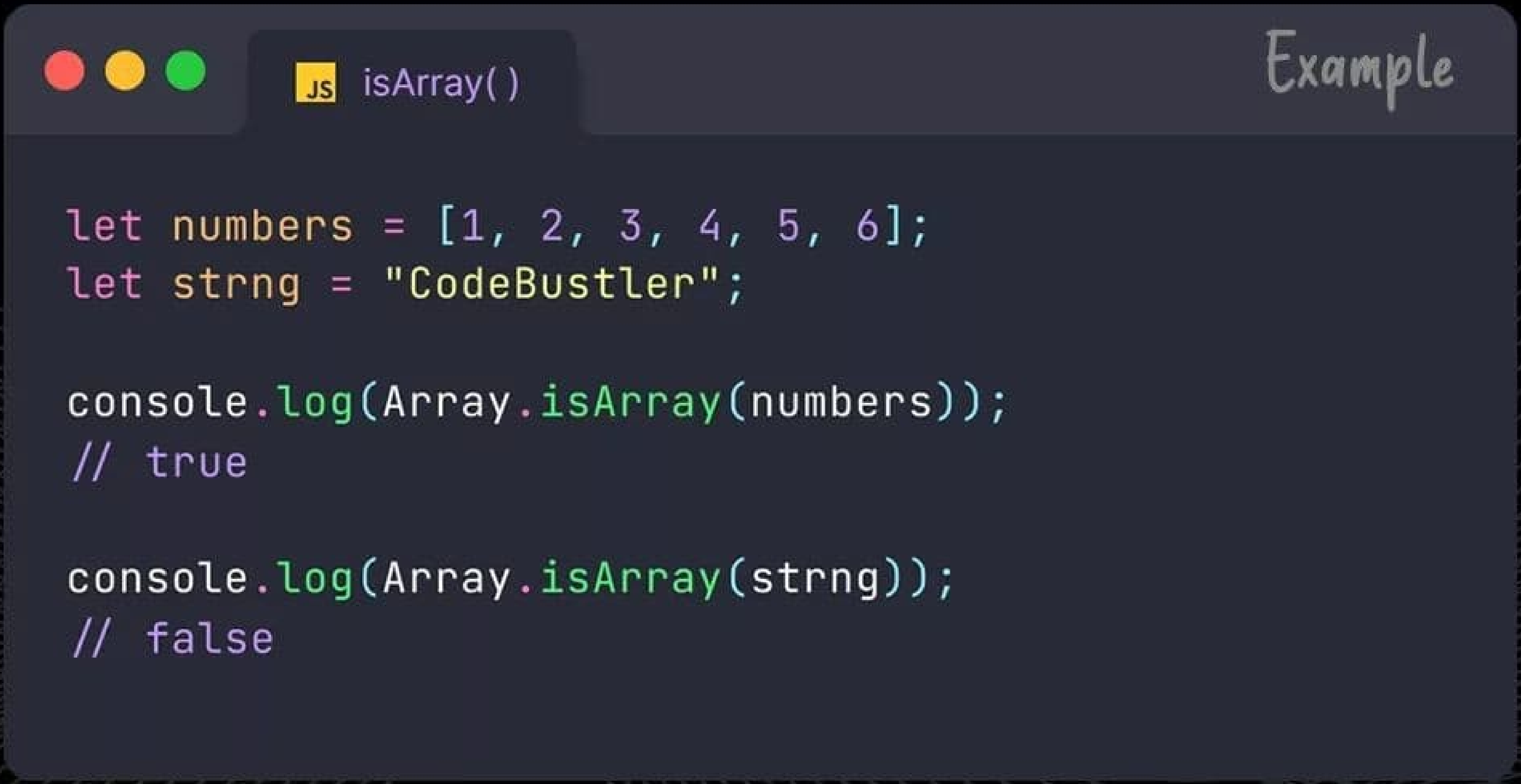
Swipe

13. isArray()

The *isArray( )* method returns *true* if an object is an array, otherwise *false.*

Check if an object is an array.

```
Array.isArray(obj); //Syntax
```


isArray( )

Example

```
let numbers = [1, 2, 3, 4, 5, 6];
let strng = "CodeBustler";

console.log(Array.isArray(numbers));
// true

console.log(Array.isArray(strng));
// false
```

Swipe

14. indexOf( )

The *indexOf( )* method returns the first index (position) of a specified value, returns -1 if the value is not found. and it searches from *left to right*.

Negative start values counts from the last element (but still searches from left to right)

```
array.indexOf(item, start); //Syntax
```

indexOf( )

Example

```
let elements = ["laptop", "HeadSet", "Mobile", "Router"];

console.log(elements.indexOf("Mobile", 0)); // 2

console.log(elements.indexOf("Mobile", 3)); // -1
```

Swipe

15. lastIndexOf( )

The *lastIndexOf( )* method returns the last index (position) of a specified value, returns *-1* if the value is not found. starts at a specified index and searches from *right to left*.

Negative start values counts from the last element (but still searches from right to left).

```
array.lastIndexOf(item, start); //Syntax
```

lastIndexOf( )

Example

```
let elements = ["laptop", "Mobile", "HeadSet", "Mobile", "Router"];

console.log(elements.indexOf("Mobile", 0)); // ☞ 1
// indexOf(): left to right

console.log(elements.lastIndexOf("Mobile", 4)); //☞ 3
// lastIndexOf() : right to left
```

Swipe

16. find( )

The *find( )* method returns the *first element* in the provided array that satisfies the provided testing function.

If no values satisfy the testing function, *undefined* is returned.

```
//Syntax (arrow function)

find((element) => {/* … */});
```

find( ) Example

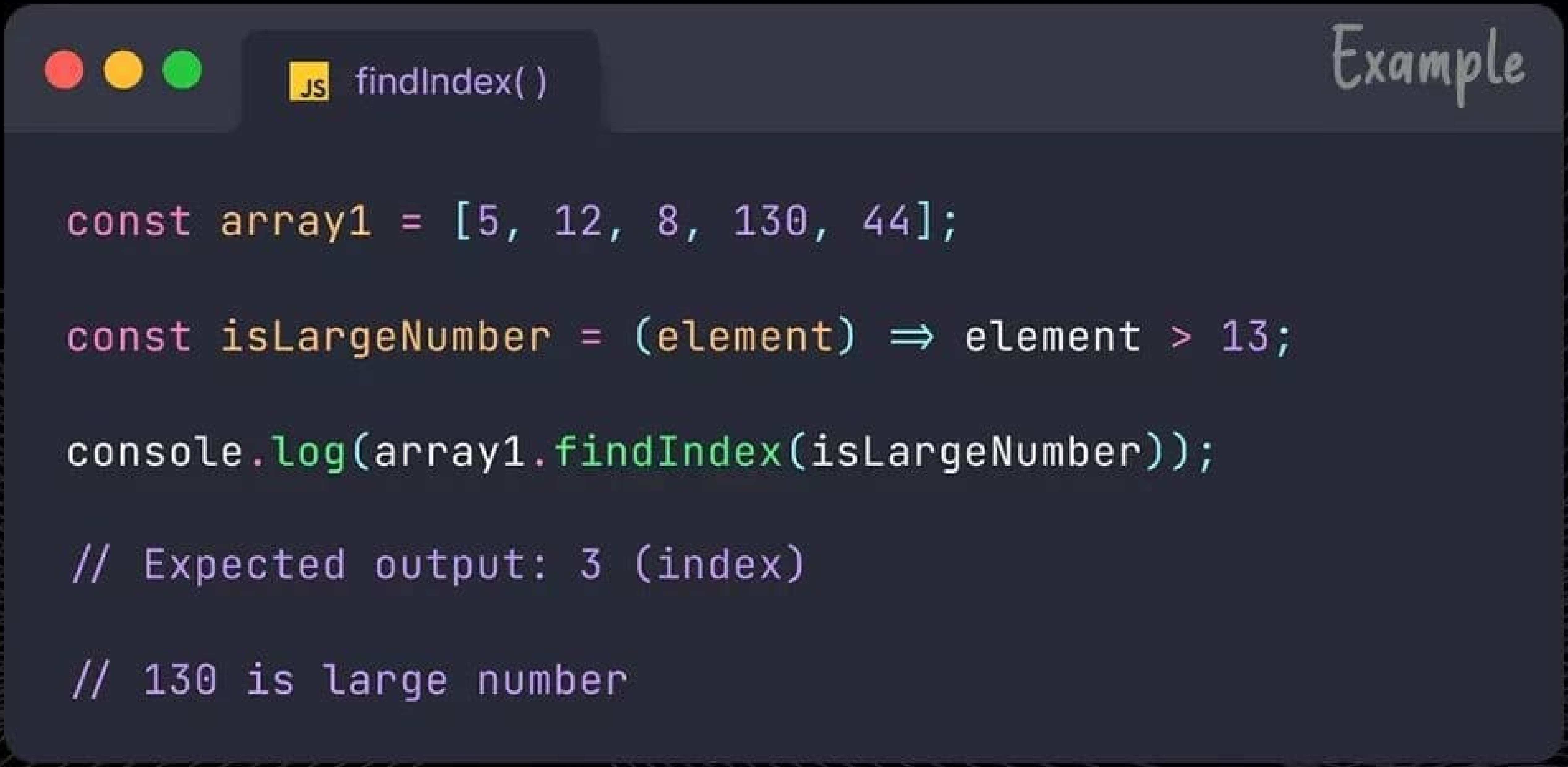
```
const arry = [5, 12, 8, 130, 44];

const found = arry.find((element) => element > 10);

console.log(found); // 12
```

Swipe

17. findIndex( )

The *findIndex( )* method returns the index of the first element in an array that satisfies the provided testing function. If no elements satisfy the testing function, then -1 is returned.

```
//Syntax

array.findIndex(function(currentValue, index, arr), thisValue)
```



```
const array1 = [5, 12, 8, 130, 44];

const isLargeNumber = (element) => element > 13;

console.log(array1.findIndex(isLargeNumber));

// Expected output: 3 (index)

// 130 is large number
```

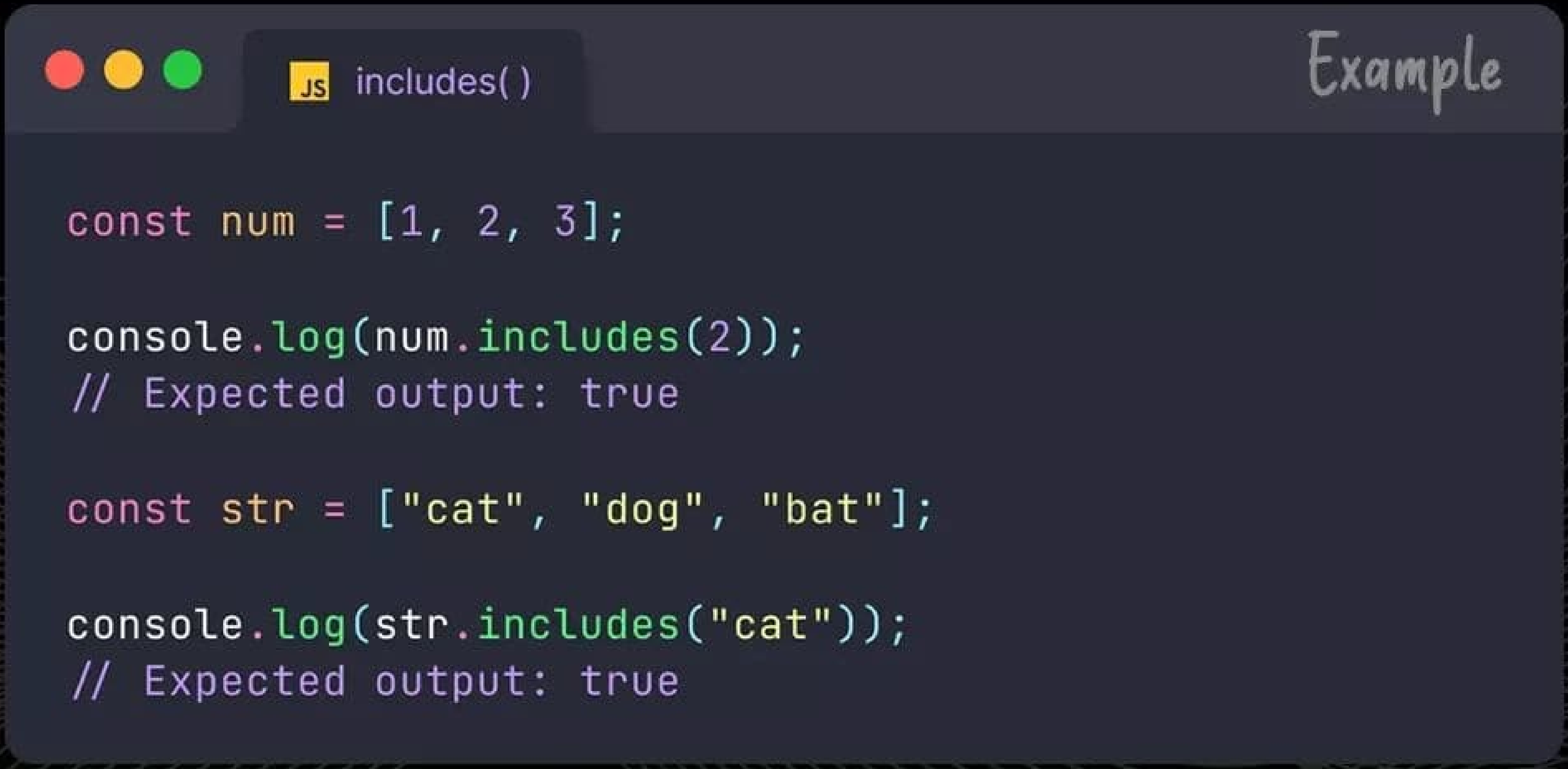
Swipe

18. includes( )

The *includes( )* method returns true if an array contains a specified value. *(case sensitive)*

if the value is not found returns false

```
array.includes(element, start); //Syntax
```


includes( )

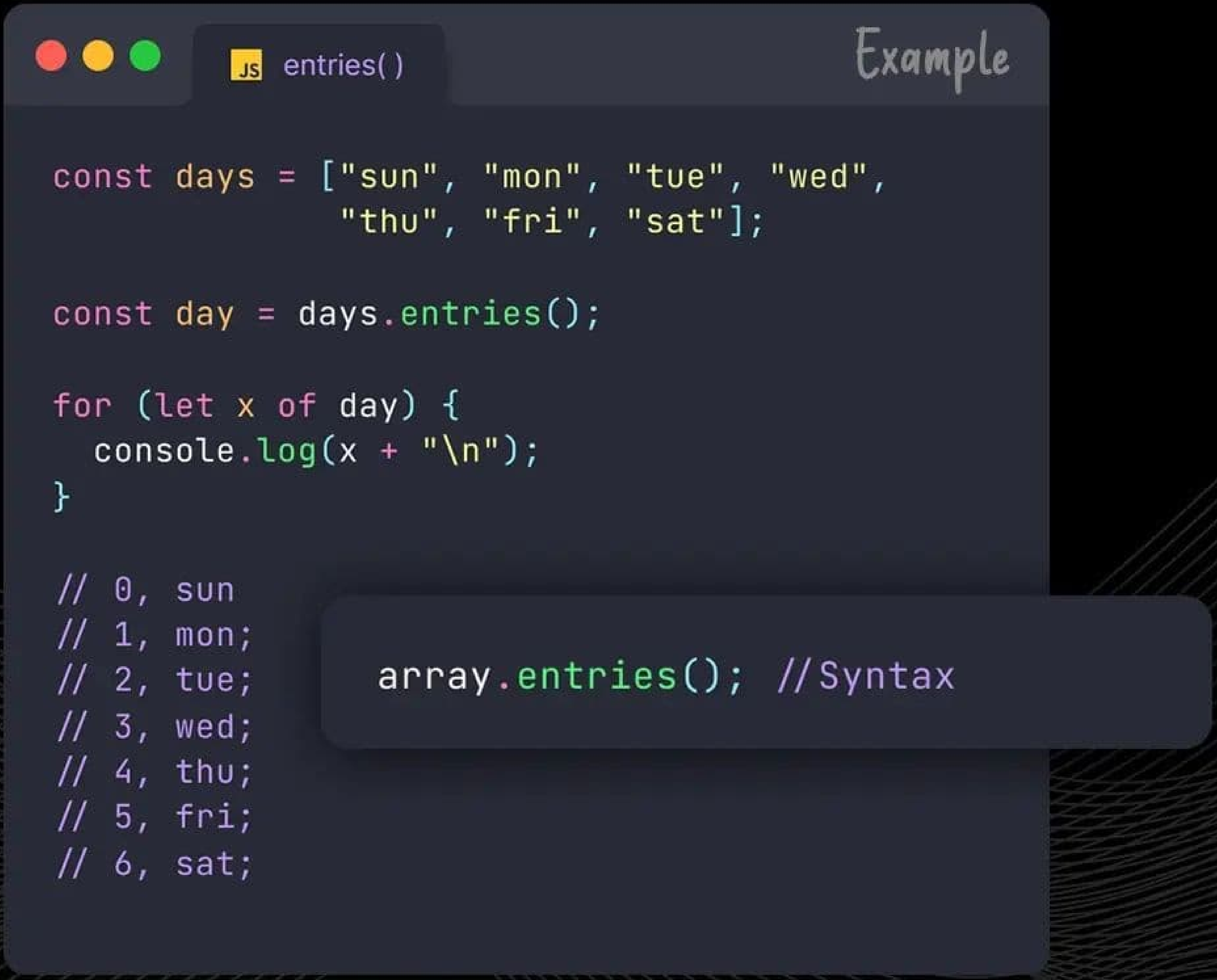
Example

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

console.log(num.includes(2));
// Expected output: true

const str = ["cat", "dog", "bat"];

console.log(str.includes("cat"));
// Expected output: true
```

Swipe

19. entries( )

The *entries( )* method returns an Array Iterator object with key/value pairs. And this method does not change the original array.



```
const days = ["sun", "mon", "tue", "wed",
              "thu", "fri", "sat"];

const day = days.entries();

for (let x of day) {
  console.log(x + "\n");
}

// 0, sun
// 1, mon;
// 2, tue;
// 3, wed;
// 4, thu;
// 5, fri;
// 6, sat;
```

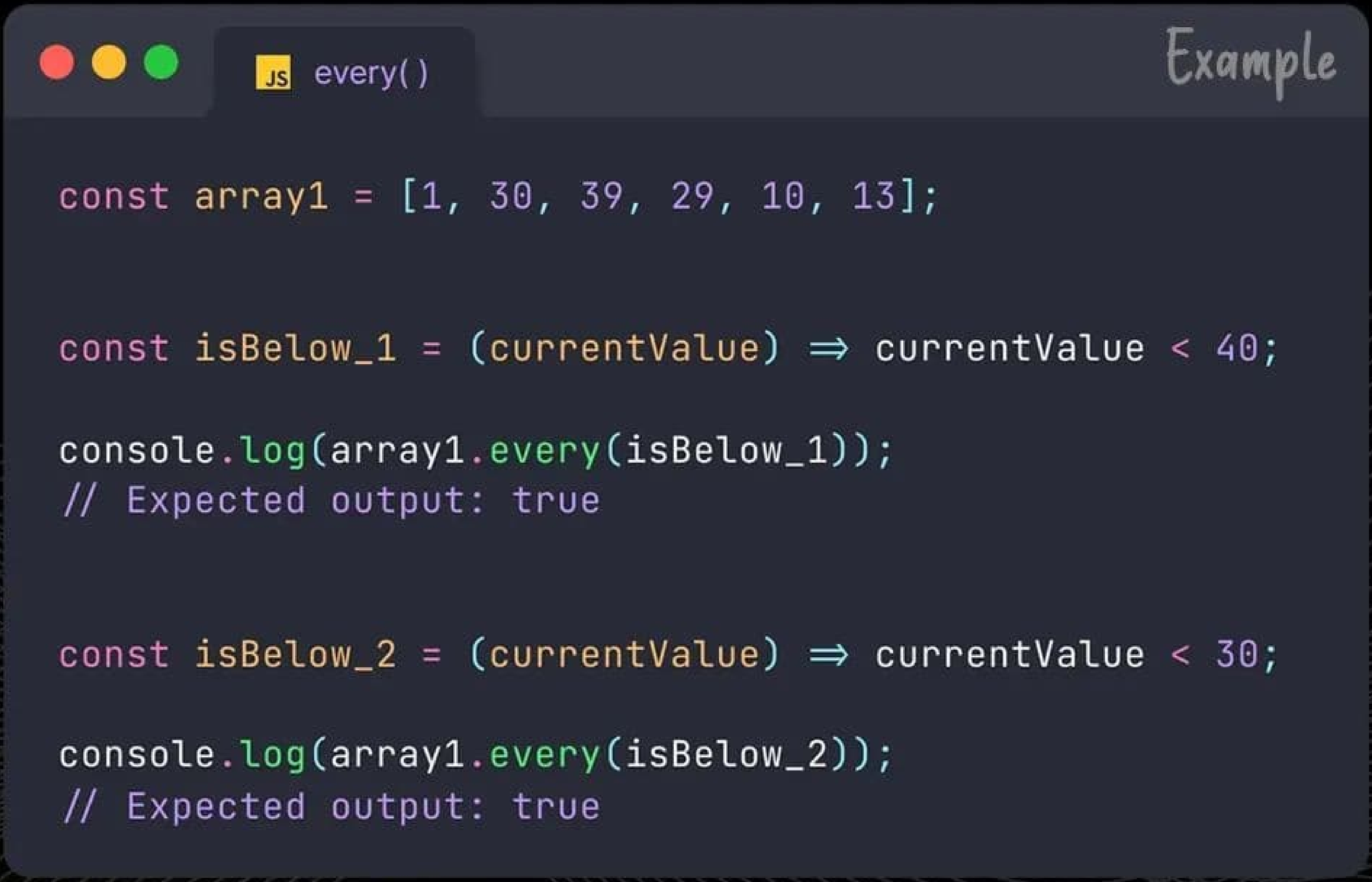
```
array.entries(); //Syntax
```

Swipe

20. every()

The *every( )* method tests whether all elements in the array pass the test implemented by the provided function. It returns a *Boolean value*.

```
every((element) => {/* code */}); // Arrow function Syntax
```



every( )

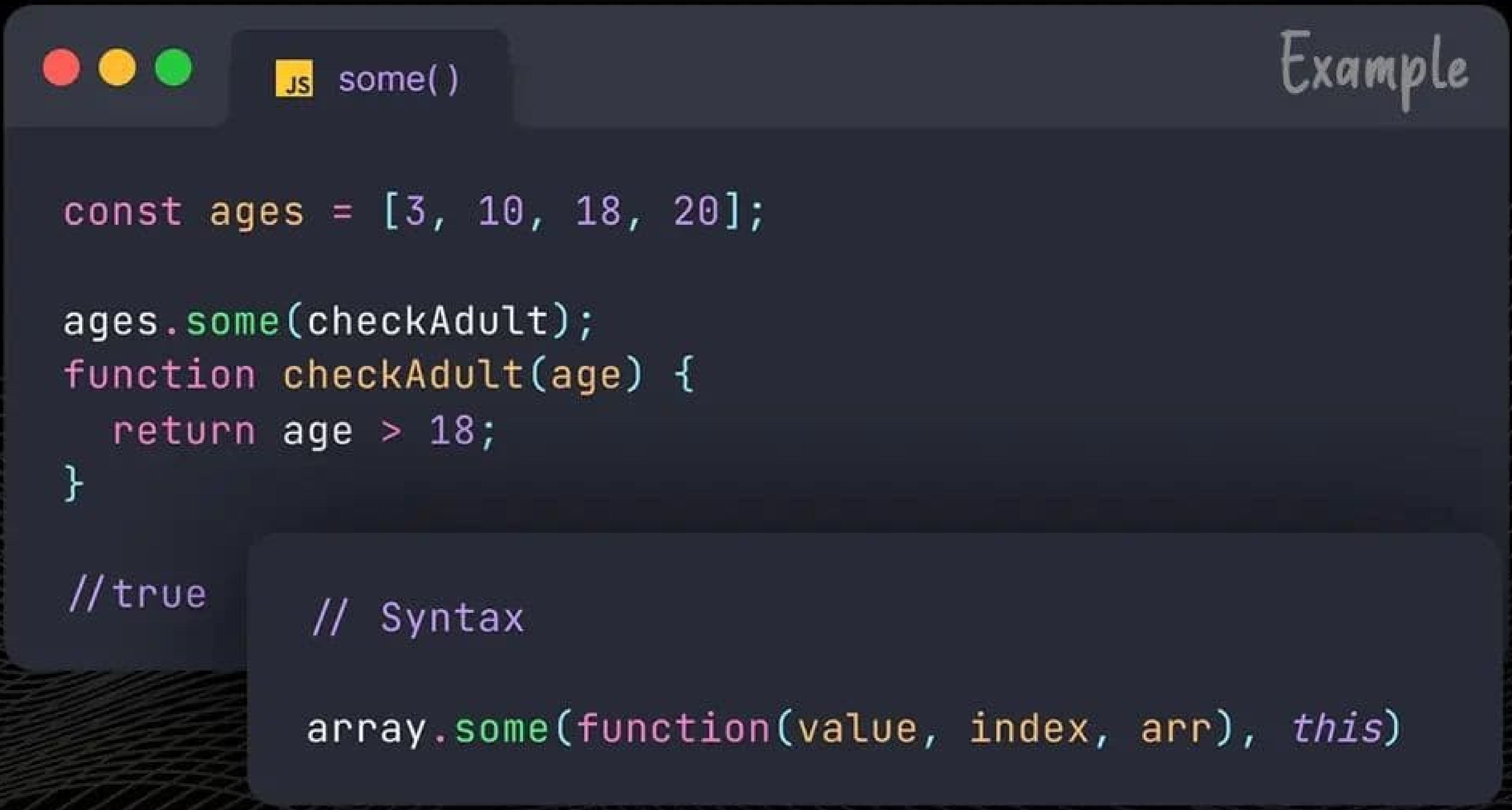
Example

```
const array1 = [1, 30, 39, 29, 10, 13];


const isBelow_1 = (currentValue) => currentValue < 40;

console.log(array1.every(isBelow_1));
// Expected output: true


const isBelow_2 = (currentValue) => currentValue < 30;

console.log(array1.every(isBelow_2));
// Expected output: true
```

21. some( )

The *some( )* method tests whether at least one el-ement in the array passes the test implemented by the provided function.

It returns *true* if, in the array, it finds an element for which the provided function returns true; otherwise it returns *false*. It doesn't modify the array.



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

ages.some(checkAdult);
function checkAdult(age) {
  return age > 18;
}

//true
```

```javascript
// Syntax

array.some(function(value, index, arr), this)
```

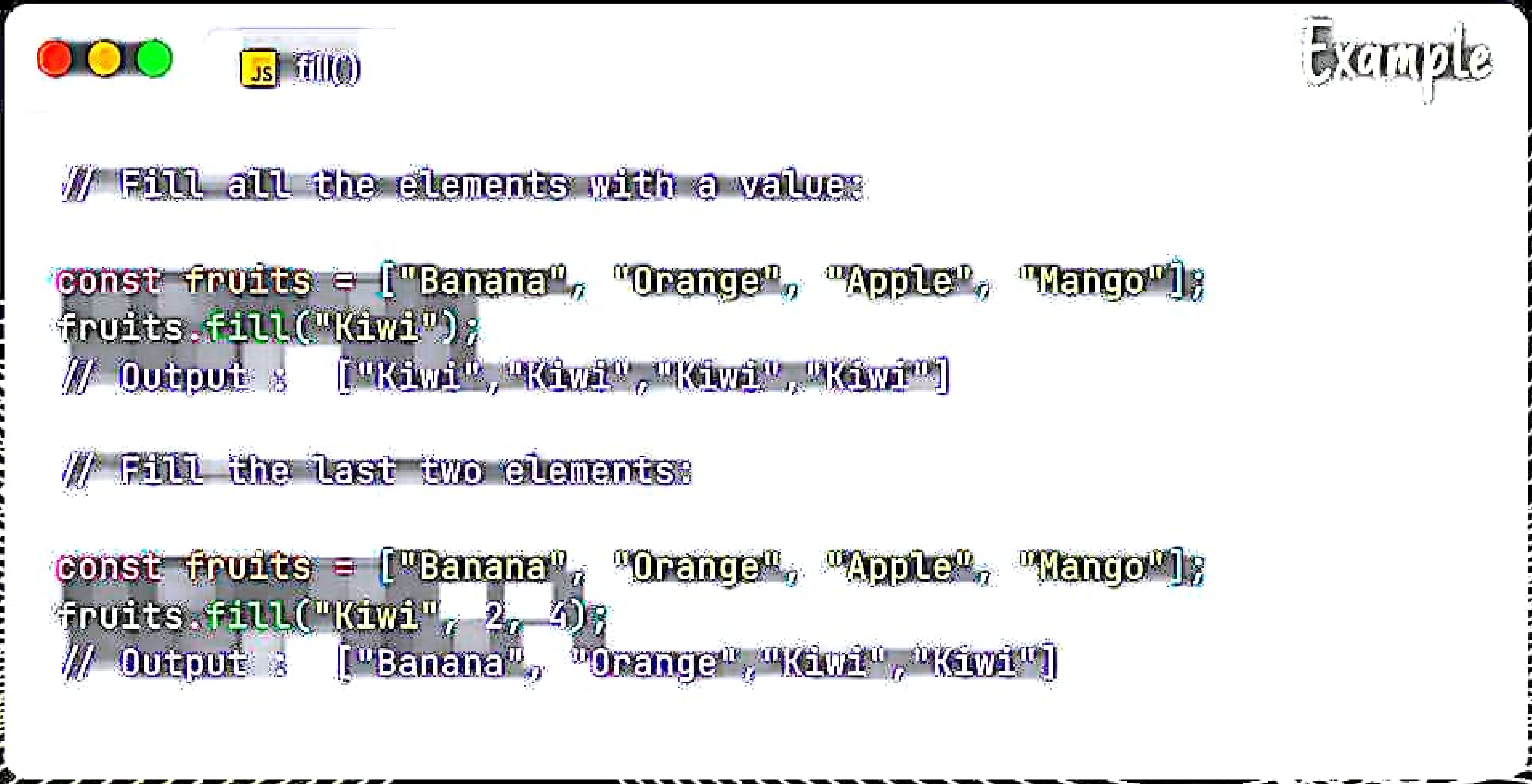
Swipe

22. fill( )

The *fill( )* method fills specified elements in an array with a value. method overwrites the original array.

Start and end position can be specified. If not, all *elements will be filled.*

```
array.fill(value, start, end); // Syntax
```


fill()

Example

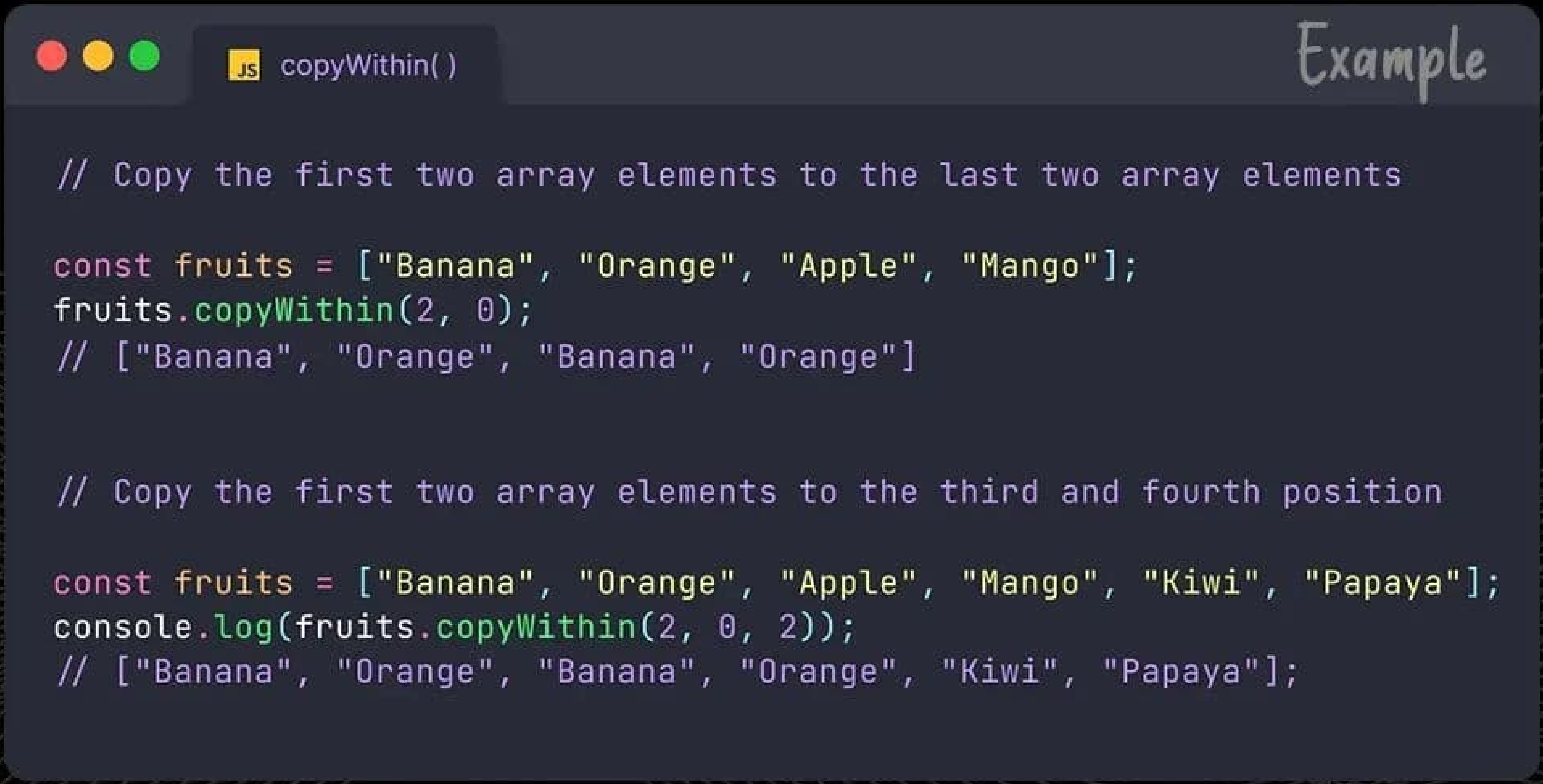
```
// Fill all the elements with a value:

const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.fill("Kiwi");
// Output :   ["Kiwi","Kiwi","Kiwi","Kiwi"]

// Fill the last two elements:

const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.fill("Kiwi", 2, 4);
// Output :   ["Banana", "Orange","Kiwi","Kiwi"]
```

23. copyWithin( )

The *copyWithin( )* method copies array elements to another position in the array, and this method *overwrites* the existing values.

The copyWithin() method does not add items to the array.

```
array.copyWithin(target, start, end); // Syntax
```



```
// Copy the first two array elements to the last two array elements

const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.copyWithin(2, 0);
// ["Banana", "Orange", "Banana", "Orange"]


// Copy the first two array elements to the third and fourth position

const fruits = ["Banana", "Orange", "Apple", "Mango", "Kiwi", "Papaya"];
console.log(fruits.copyWithin(2, 0, 2));
// ["Banana", "Orange", "Banana", "Orange", "Kiwi", "Papaya"];
```

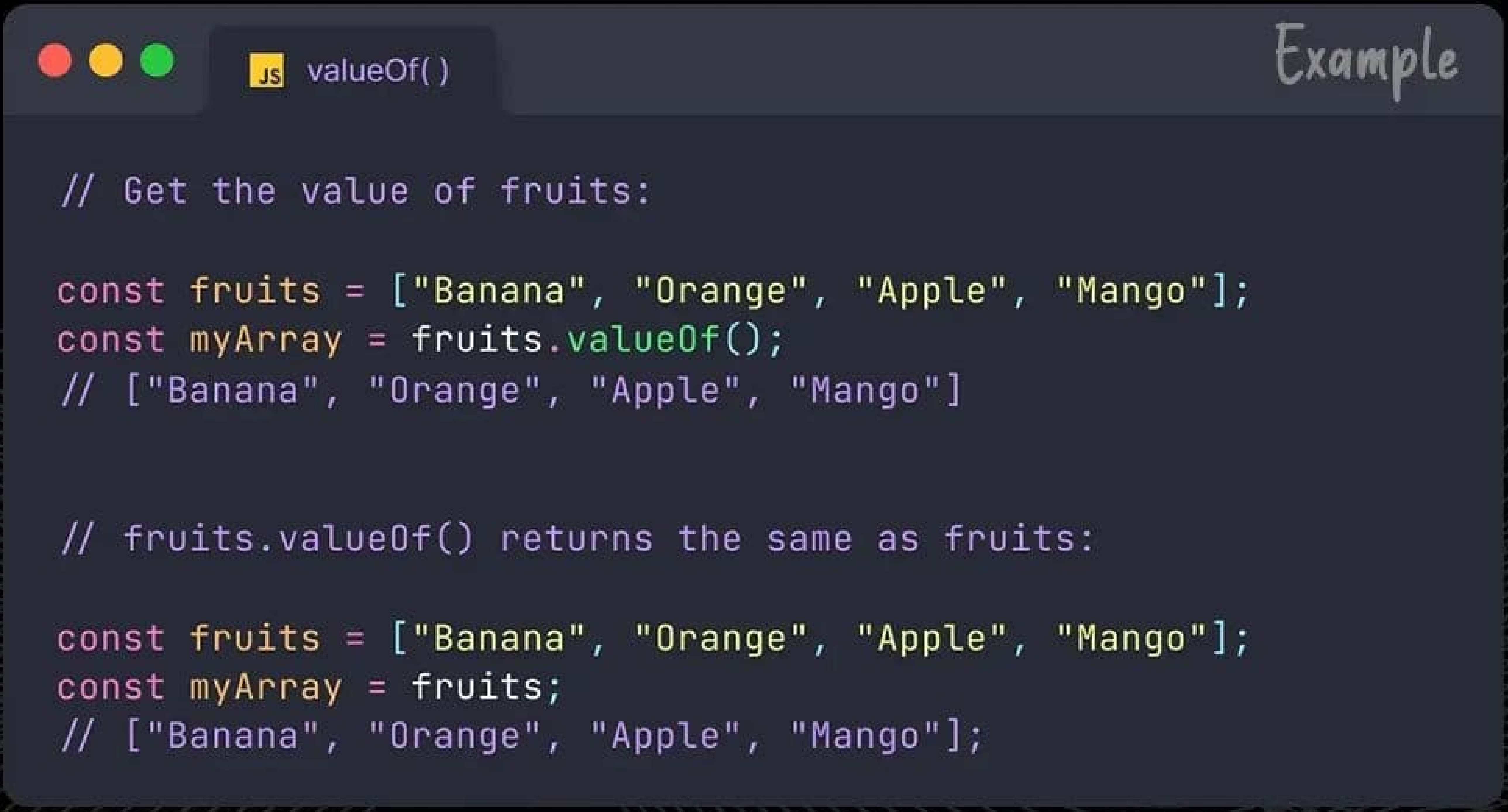
Swipe

24. valueOf( )

The *valueOf( )* method returns the array itself.and this method does not change the original array.

fruits.valueOf() returns the same as fruits.í?

```
array.valueOf(); // Syntax
```



valueOf( )

Example

```
// Get the value of fruits:

const fruits = ["Banana", "Orange", "Apple", "Mango"];
const myArray = fruits.valueOf();
// ["Banana", "Orange", "Apple", "Mango"]


// fruits.valueOf() returns the same as fruits:

const fruits = ["Banana", "Orange", "Apple", "Mango"];
const myArray = fruits;
// ["Banana", "Orange", "Apple", "Mango"];
```

Swipe

25. forEach( )

The *forEach( )* method executes a provided function once for each array element. And this method is not executed for empty elements.

```
// Syntax

array.forEach(function(currentValue, index, arr), thisValue)
```

forEach()

Example

```
// Multiply each element:

const numbers = [65, 44, 12, 4];
numbers.forEach(myFunction);

function myFunction(item, index, arr) {
  arr[index] = item * 10;
}

//650, 440, 120, 40
```

Swipe

27. filter( )

The *filter( )* method creates a new array filled with elements that pass a test provided by a function.

And this method does not change the original array.

```
// Syntax

array.filter(function(currentValue, index, arr), thisValue)
```

filter( )

Example

```
const ages = [32, 33, 16, 40];
const result = ages.filter(checkAdult);

function checkAdult(age) {
  return age > = 18;
}

// 32, 33, 40
```

Swipe

28. reduce()

The *reduce( )* method executes a *reducer function* for array element. and this method returns a single value: the function's accumulated result.

```
// Syntax

array.reduce(function(total, currentValue,
currentIndex, arr), initialValue);
```

JS reduce()

Example

```
const numbers = [175, 50, 25];

numbers.reduce(myFunc);

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

// 24
```

Swipe

29. reduceRight( )

The *reduce( )* method executes a *reducer function* for array element. and this method returns a single value: the function's accumulated result.

> works from right to left.

JS reduceRight( )

Example

```
const numbers = [175, 50, 25];

numbers.reduceRight(myFunc);

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

// -200
```

```
// Syntax
array.reduceRight(function(total, currentValue,
 currentIndex, arr), initialValue);
```

Swipe

30. from( )

The *Array.from( )* method returns an array from any object with a length property.

And this method returns an array from any iterable object.

```
// Syntax

Array.from(object, mapFunction, thisValue);
```

JS from( ) Example

```
// Create an array from a string:

console.log(Array.from("Code"));

// Array ["C", "o", "d", "e"]
```