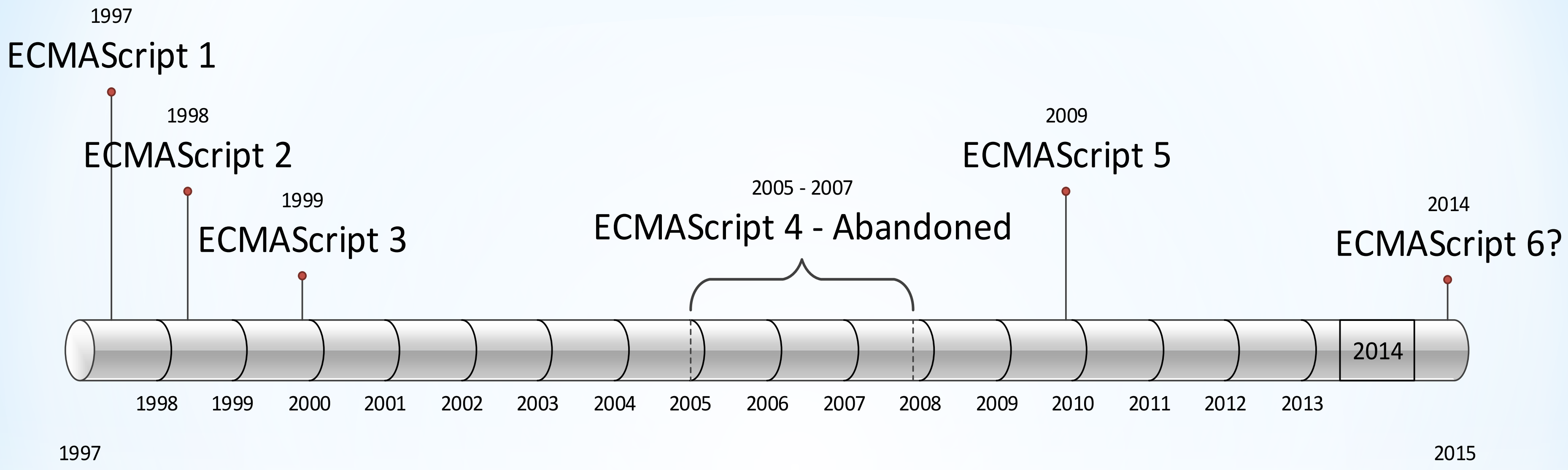
*ECMAScript 6

Ahmed Elashry
aelashry@outlook.com

*ECMAScript History



Language Features

- string
- BLOCK SCOPING
- Function
- Destructuring assignment
- Array
- Object
- Class
- Promise

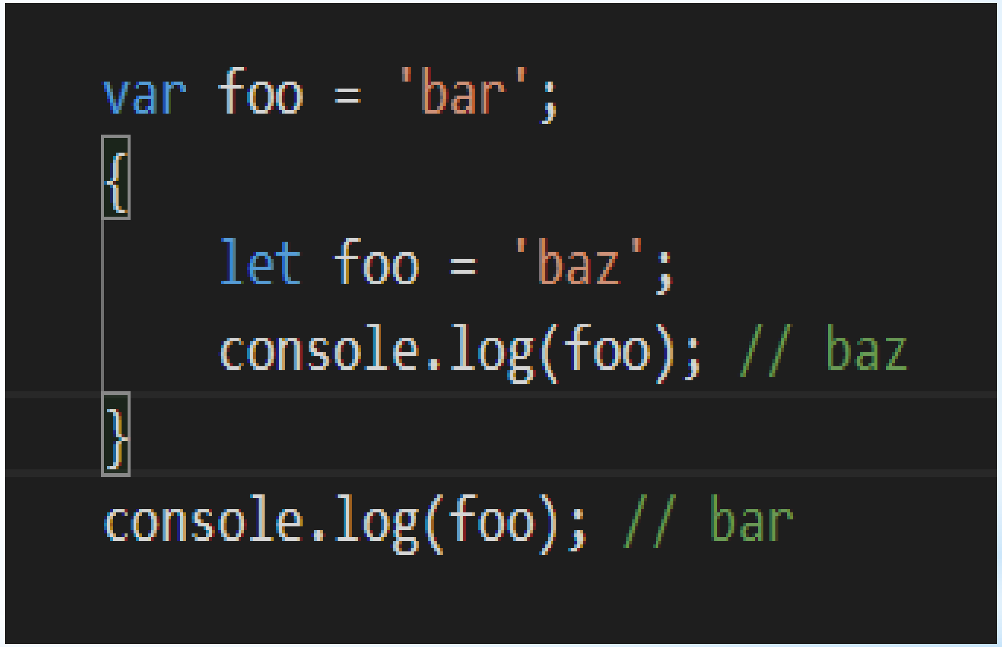
*ES6 Keywords

break

- case
- class
- catch
- const
- continue
- constructor
- debugger
- default
- delete
- do
- else
- export
- extends
- finally
- for
- function
- get
- if
- import
- in
- instanceof
- let
- new
- null
- return
- set
- super
- static
- switch
- this
- throw
- try
- typeof
- var

*Let and Block Scope

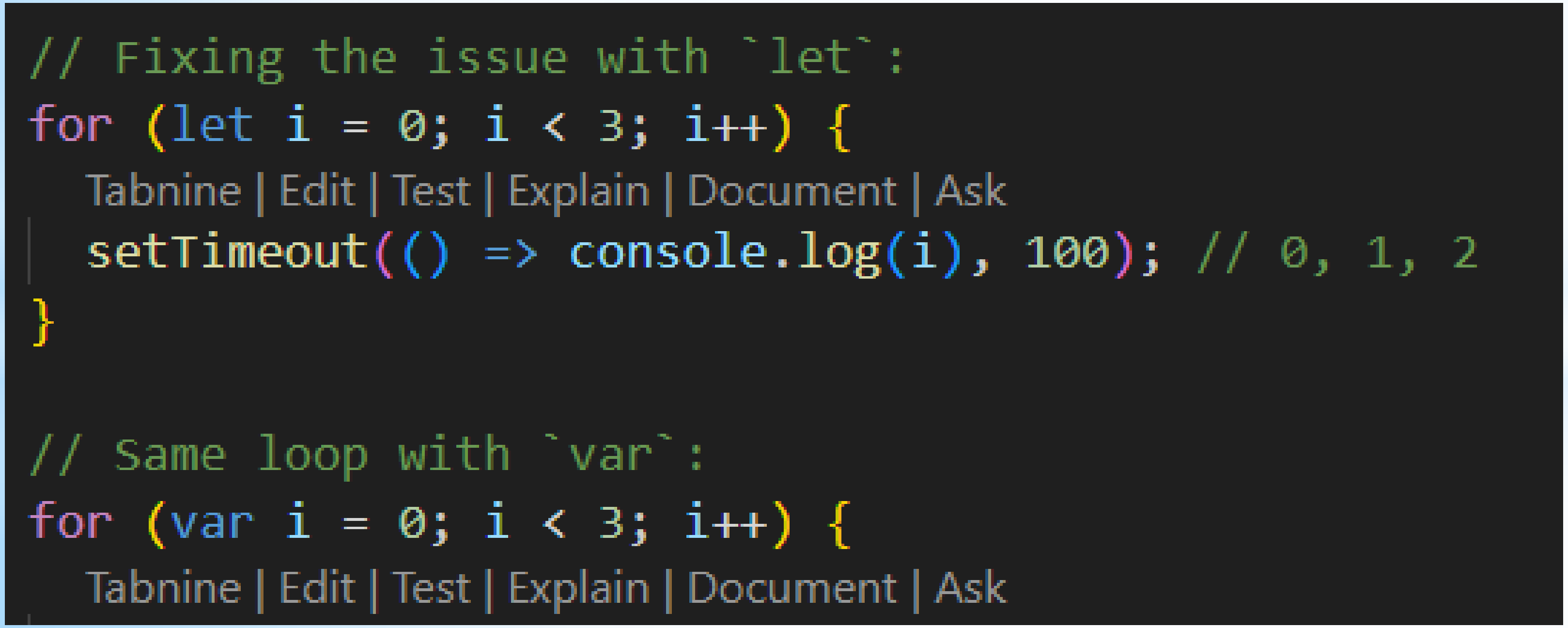
- The block scope restricts a variable’s access to the block in which it is declared. The **var** keyword assigns a function scope to the variable. Unlike the **var** keyword, the **let** keyword allows the script to restrict access to the variable to the nearest enclosing block.



```
var foo = 'bar';
{
    let foo = 'baz';
    console.log(foo); // baz
}
console.log(foo); // bar
```

*Var VS let

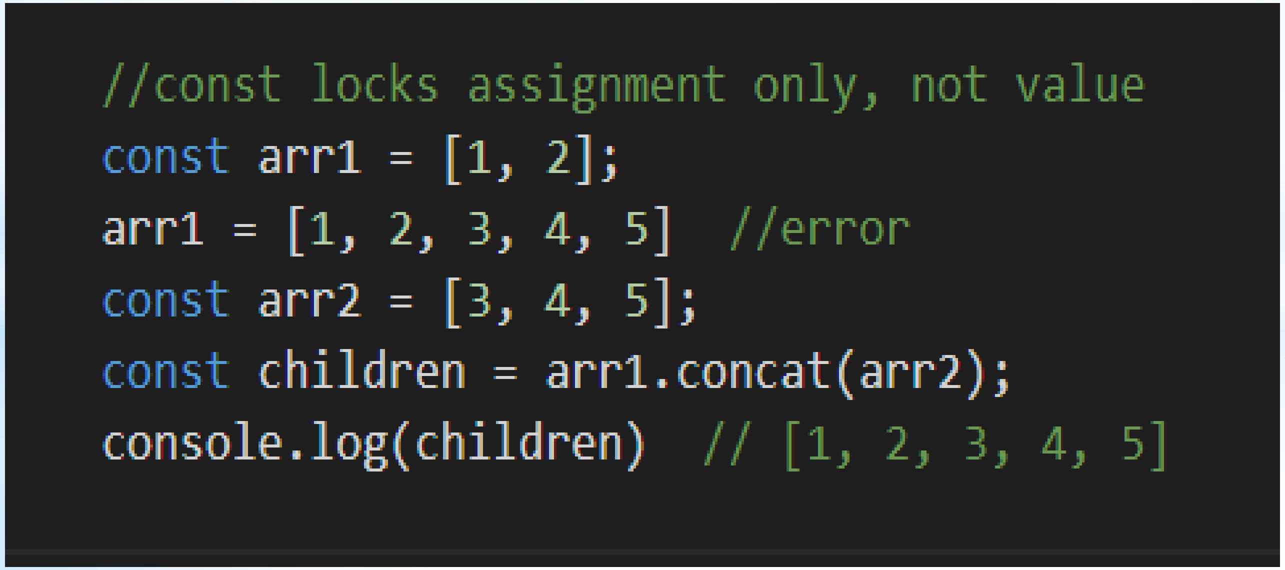
- let Scenario:
- Each iteration of the loop creates a new i variable, effectively capturing the current value of i within that specific iteration.
- When the callback functions execute after 100 milliseconds, they access their respective captured i values, resulting in the output 0, 1, 2.



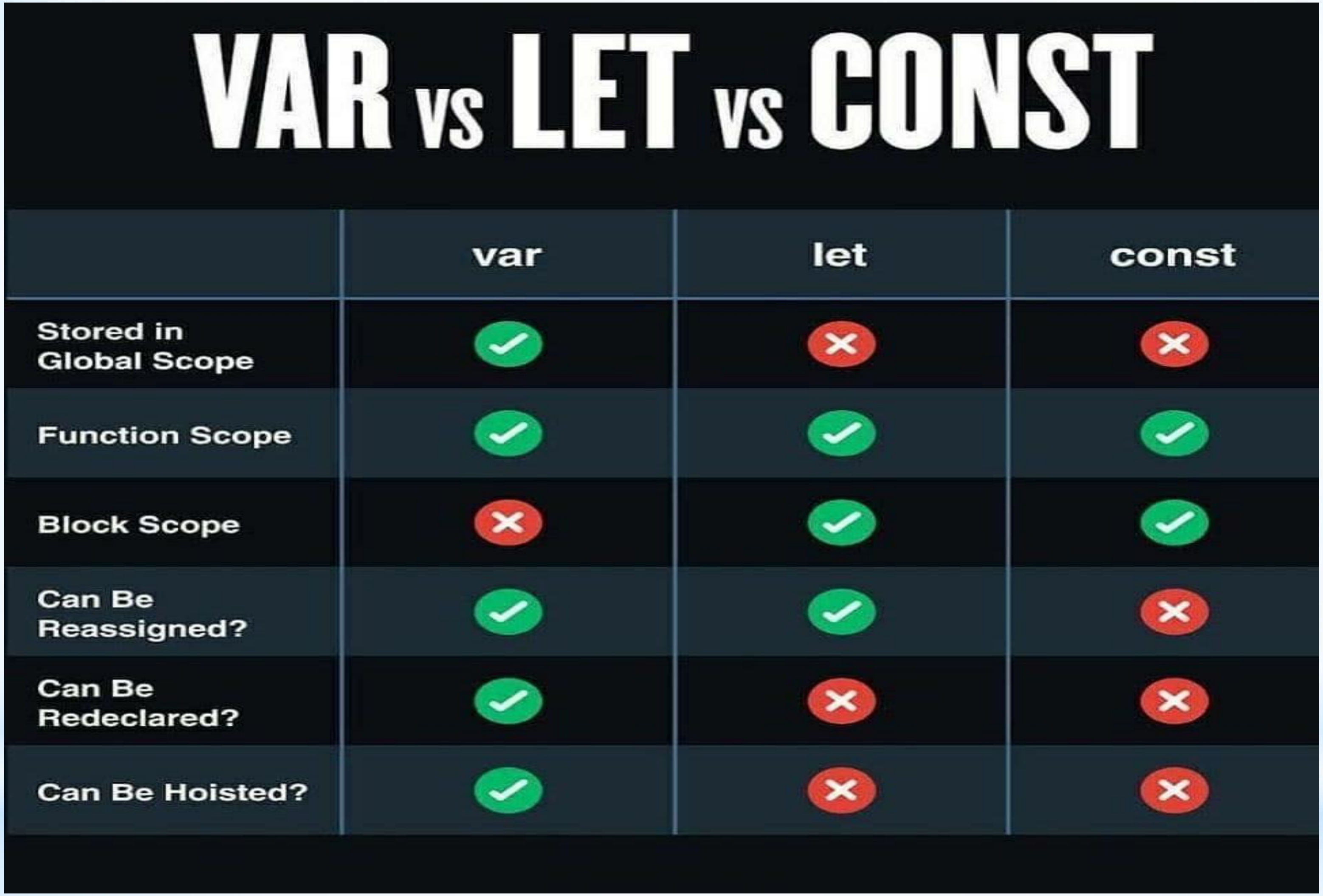
```
// Fixing the issue with `let`:
for (let i = 0; i < 3; i++) {
  Tabnine | Edit | Test | Explain | Document | Ask
  setTimeout(() => console.log(i), 100); // 0, 1, 2
}

// Same loop with `var`:
for (var i = 0; i < 3; i++) {
  Tabnine | Edit | Test | Explain | Document | Ask
```

The const

- This declaration creates a constant whose scope can be either global or local to the block in which it is declared. An initializer for a constant is required; that is, you must specify its value in the same statement in which it's declared (which makes sense, given that it can't be changed later).Constants cannot be reassigned a value.
  - A constant **cannot be re-declared**.
  - The value assigned to a **const** variable is immutable.

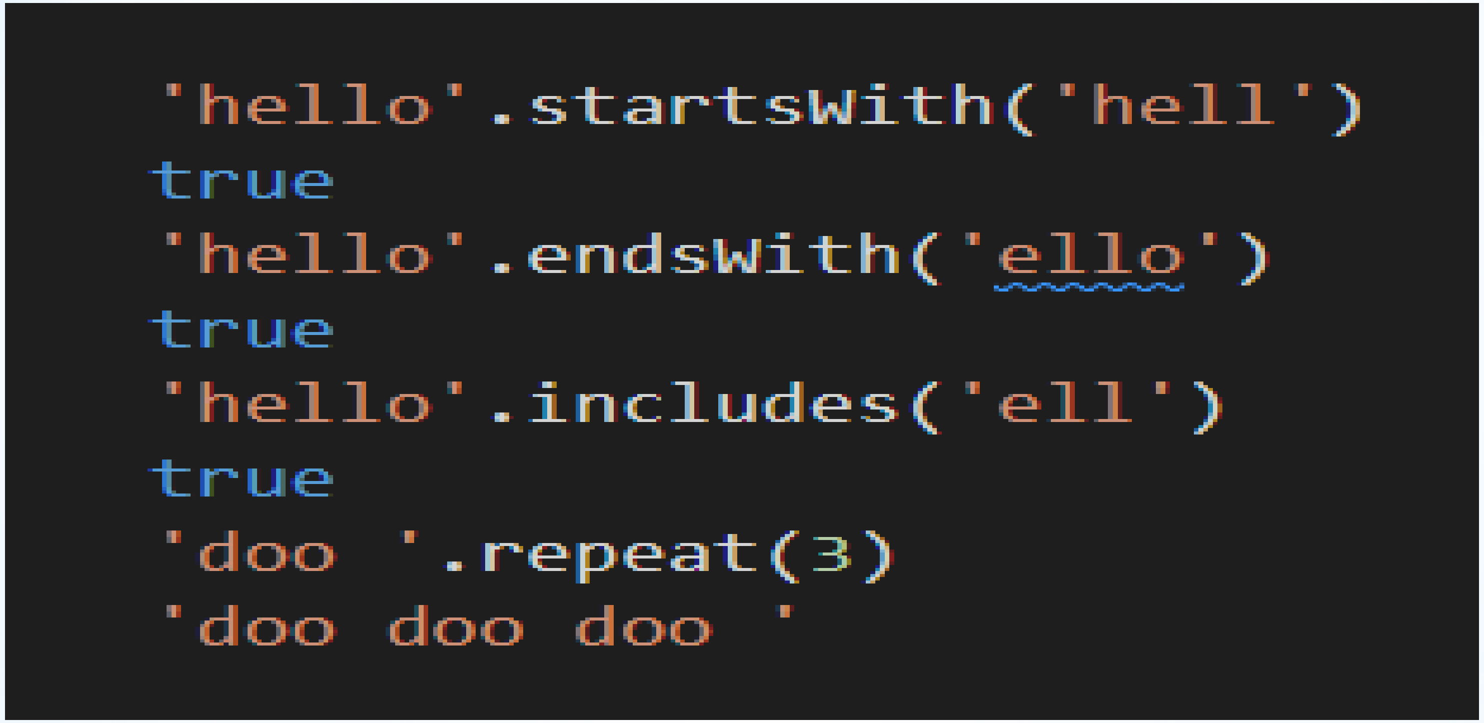


```
//const locks assignment only, not value
const arr1 = [1, 2];
arr1 = [1, 2, 3, 4, 5]  //error
const arr2 = [3, 4, 5];
const children = arr1.concat(arr2);
console.log(children)  // [1, 2, 3, 4, 5]
```



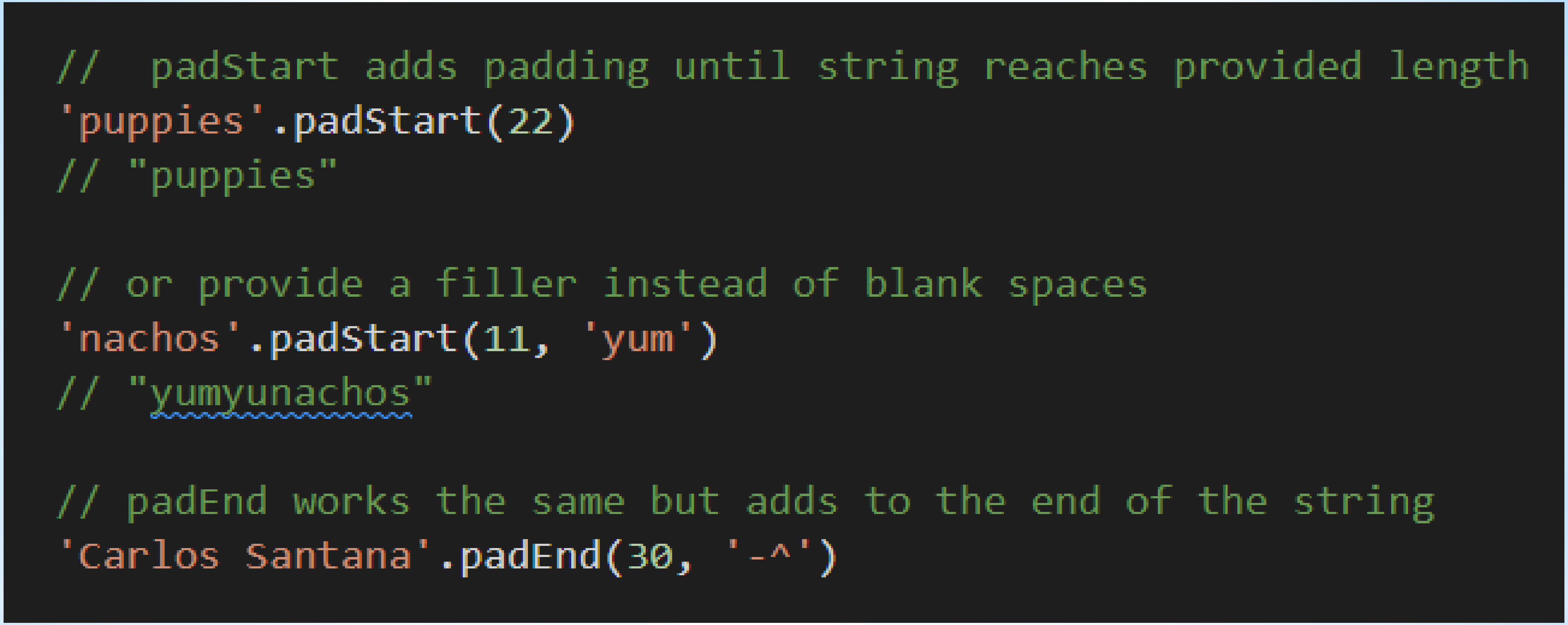
VAR vs LET vs CONST

	var	let	const
Stored in Global Scope	✓	✗	✗
Function Scope	✓	✓	✓
Block Scope	✗	✓	✓
Can Be Reassigned?	✓	✓	✗
Can Be Redeclared?	✓	✗	✗
Can Be Hoisted?	✓	✗	✗



```
'hello'.startsWith('hell')
true
'hello'.endsWith('ello')
true
'hello'.includes('ell')
true
'doo '.repeat(3)
'doo doo doo '
```

New string methods:



```
//  padStart adds padding until string reaches provided length
'puppies'.padStart(22)
// "puppies"

// or provide a filler instead of blank spaces
'nachos'.padStart(11, 'yum')
// "yumyunachos"

// padEnd works the same but adds to the end of the string
'Carlos Santana'.padEnd(30, '-^')
```

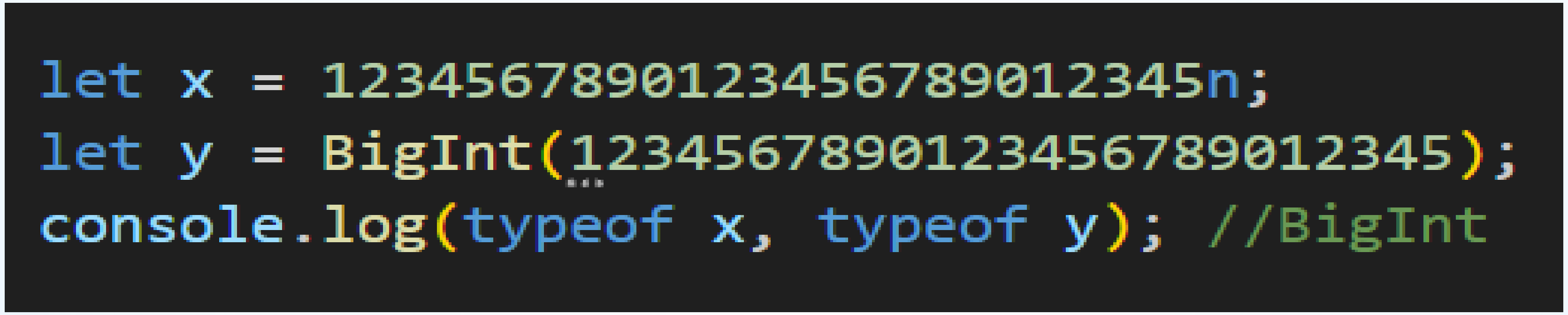
New string methods:

- Template literals are enclosed by the back-tick (` `) (grave accent) character instead of double or single quotes. Template literals can contain place holders. These are indicated by the Dollar sign and curly braces (${expression}).

*Multi-line template literals

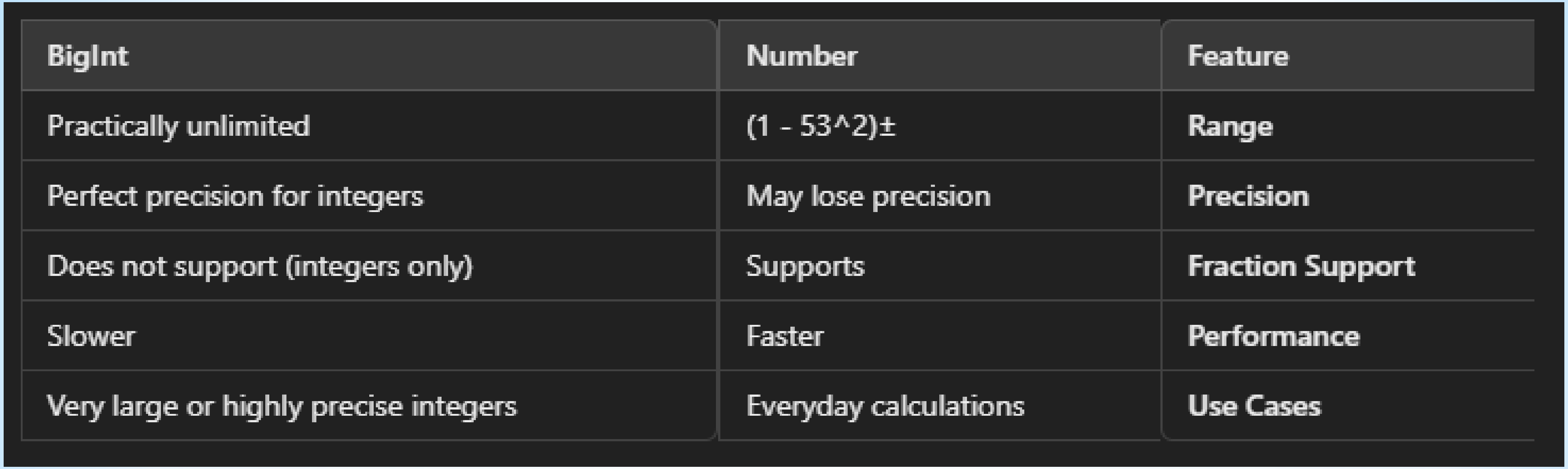
BigInt

- JavaScript BigInt variables are used to store big integer values that are too big to be represented by a a normal JavaScript Number.
- JavaScript integers are only accurate up to about 15 digits.



```
let x = 1234567890123456789012345n;
let y = BigInt(1234567890123456789012345);
console.log(typeof x, typeof y); //BigInt
```

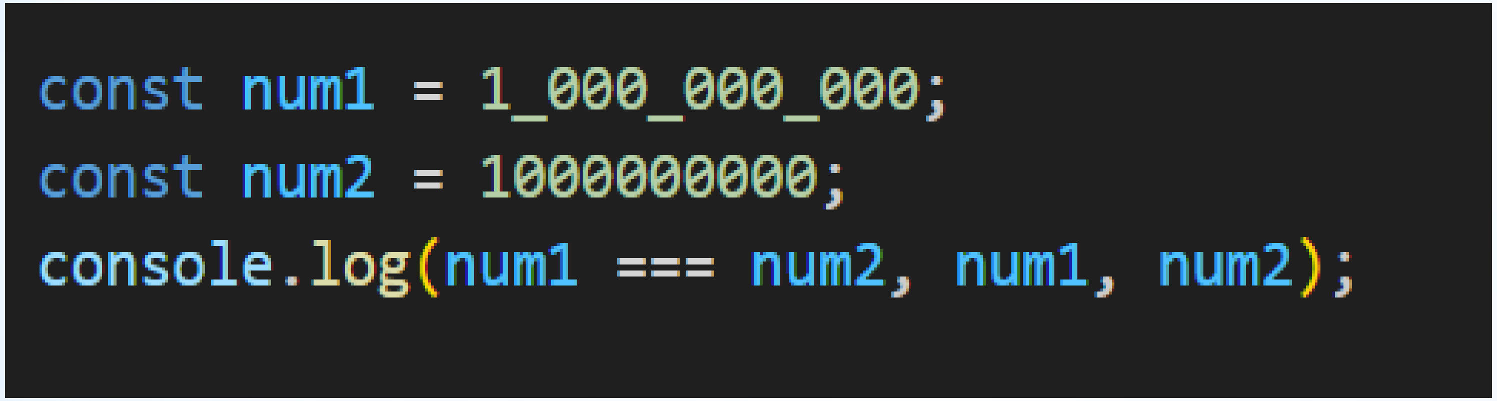
*BigInt VS Number



BigInt	Number	Feature
Practically unlimited	(1 - 53^2)±	**Range**
Perfect precision for integers	May lose precision	**Precision**
Does not support (integers only)	Supports	**Fraction Support**
Slower	Faster	**Performance**
Very large or highly precise integers	Everyday calculations	**Use Cases**

Numeric Separator (_)

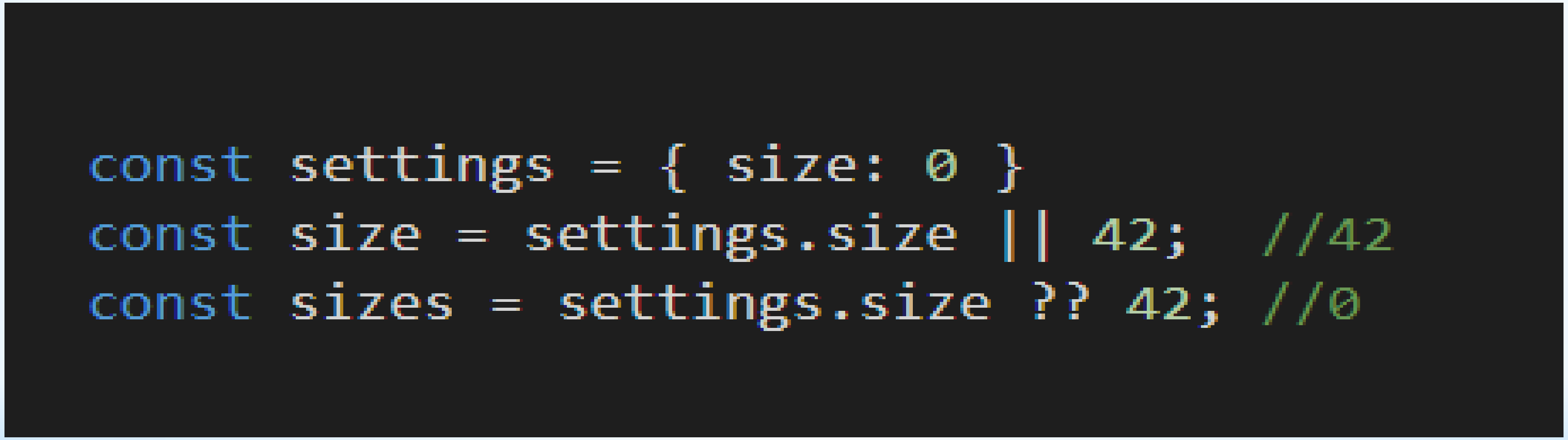
- numeric separator (_) to make numbers more readable:
- The numeric separator is not allowed at the beginning or at the end of a number.



```
const num1 = 1_000_000_000;
const num2 = 1000000000;
console.log(num1 === num2, num1, num2);
```

nullish operator

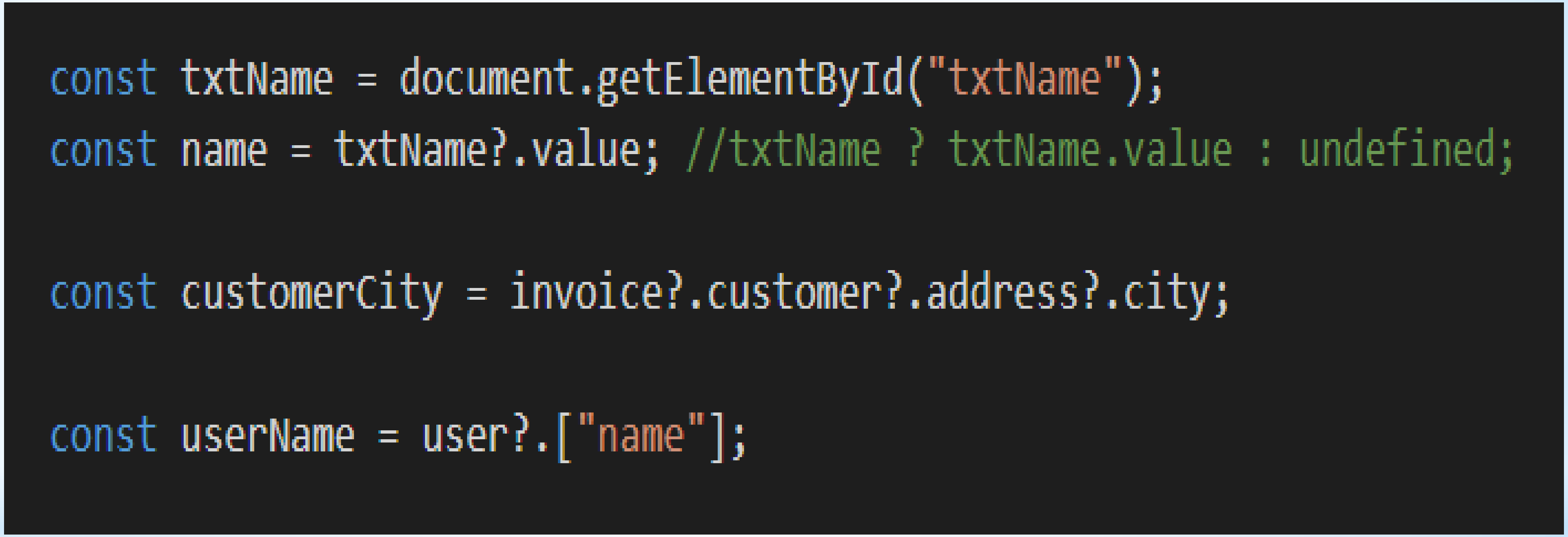
- Using the || operator is very common to assign a default value when the one you are attempting to assign is
- null or undefined => ??
- "" or false => ||



```
const settings = { size: 0 }
const size = settings.size || 42;  //42
const sizes = settings.size ?? 42; //0
```

Optional chaining

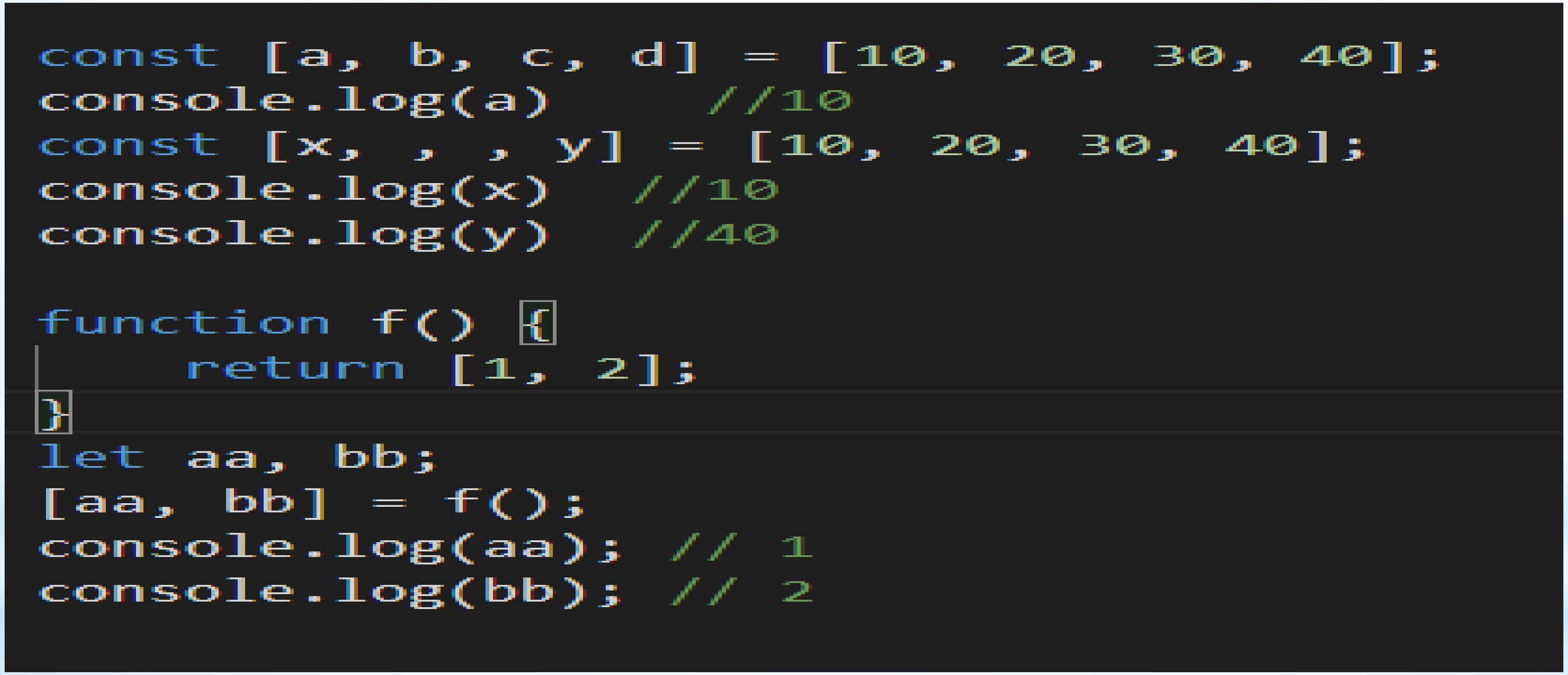
- The optional chaining operator (?.) allows you to have a more compact and readable code,



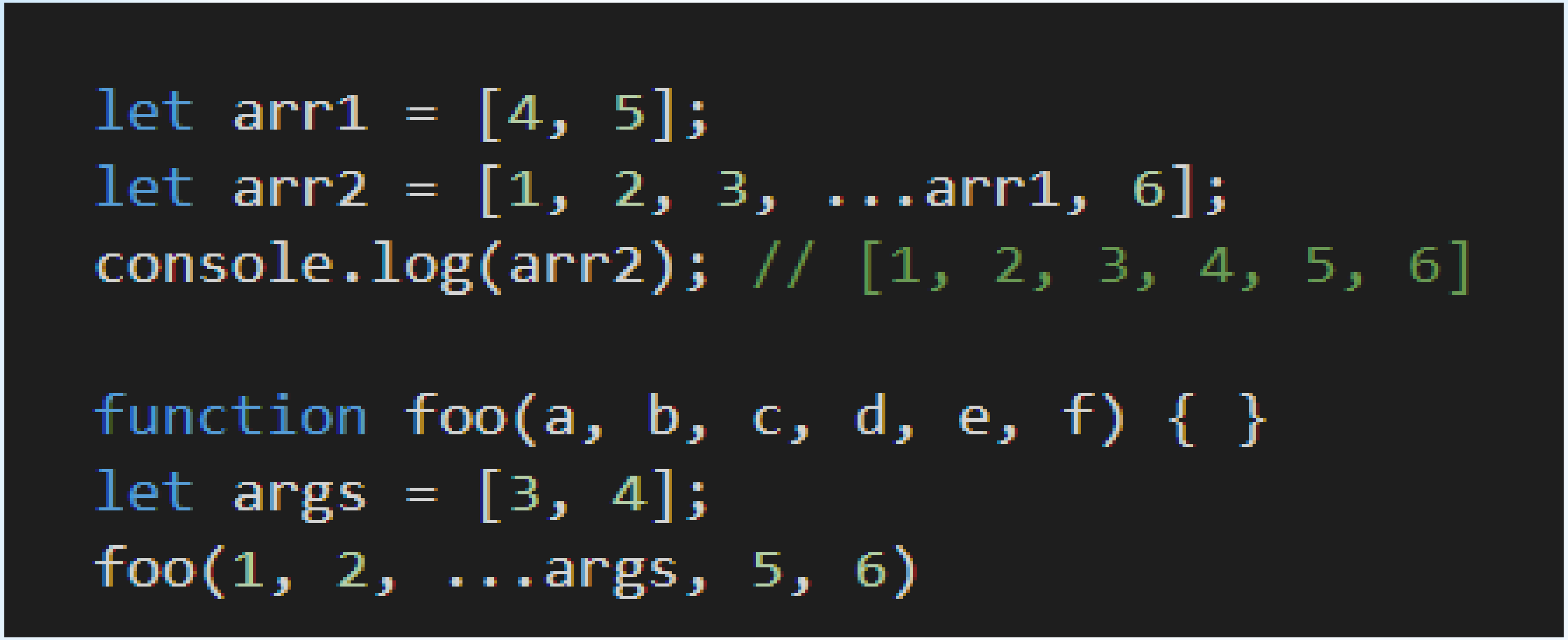
```
const txtName = document.getElementById("txtName");
const name = txtName?.value; //txtName ? txtName.value : undefined;

const customerCity = invoice?.customer?.address?.city;

const userName = user?.["name"];
```

*Destructuring assignment

*The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

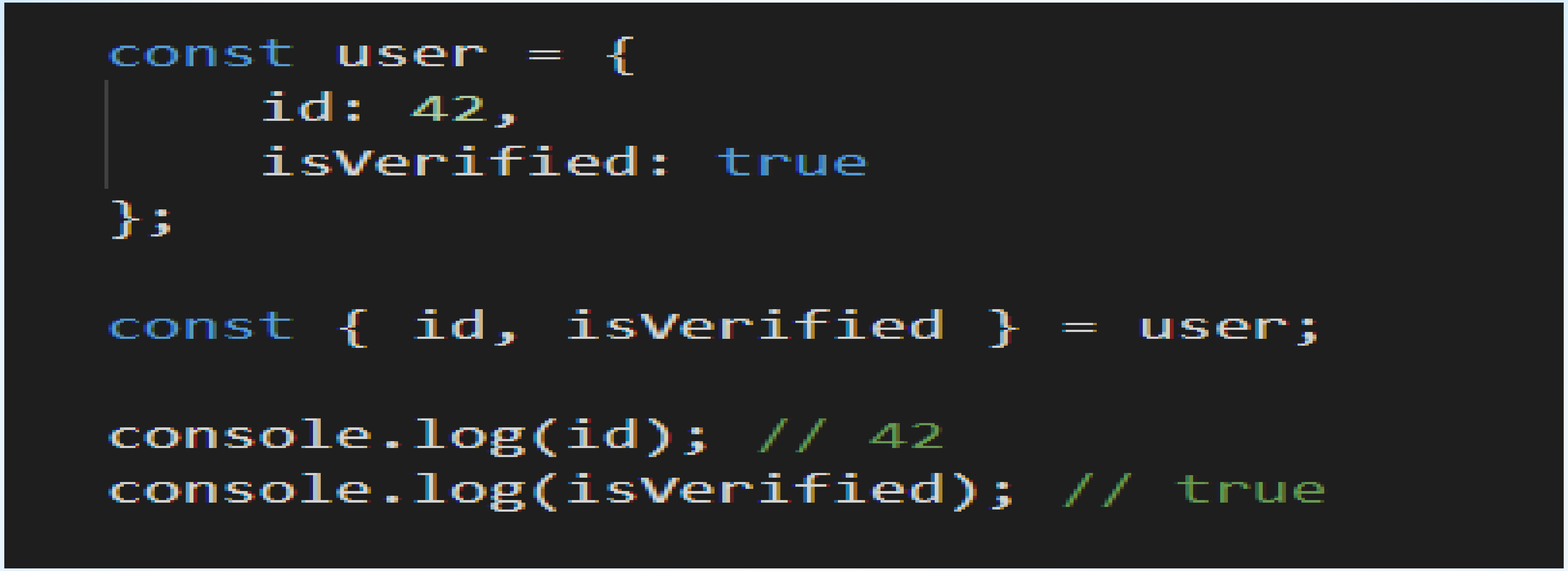


```
const [a, b, c, d] = [10, 20, 30, 40];
console.log(a)    //10
const [x, , , y] = [10, 20, 30, 40];
console.log(x)  //10
console.log(y)  //40

function f() {
    return [1, 2];
}
let aa, bb;
[aa, bb] = f();
console.log(aa); // 1
console.log(bb); // 2
```



```
let arr1 = [4, 5];
let arr2 = [1, 2, 3, ...arr1, 6];
console.log(arr2); // [1, 2, 3, 4, 5, 6]

function foo(a, b, c, d, e, f) { }
let args = [3, 4];
foo(1, 2, ...args, 5, 6)
```

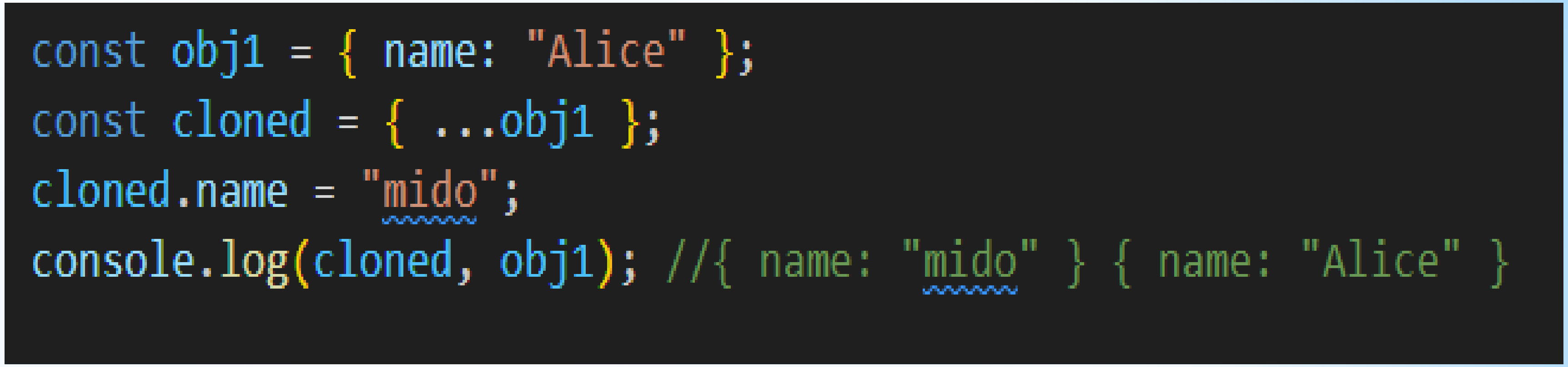
SPREAD OPERATOR



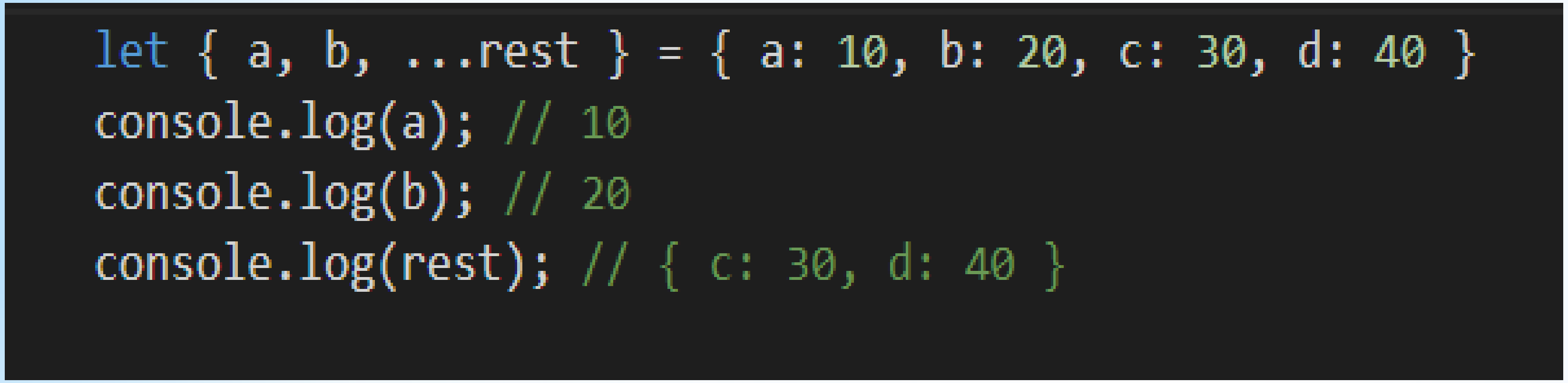
```javascript
const user = {
    id: 42,
    isVerified: true
};

const { id, isVerified } = user;

console.log(id); // 42
console.log(isVerified); // true
```

*Object destructuring

*Rest in Object Destructuring



```
const obj1 = { name: "Alice" };
const cloned = { ...obj1 };
cloned.name = "mido";
console.log(cloned, obj1); //{ name: "mido" } { name: "Alice" }
```



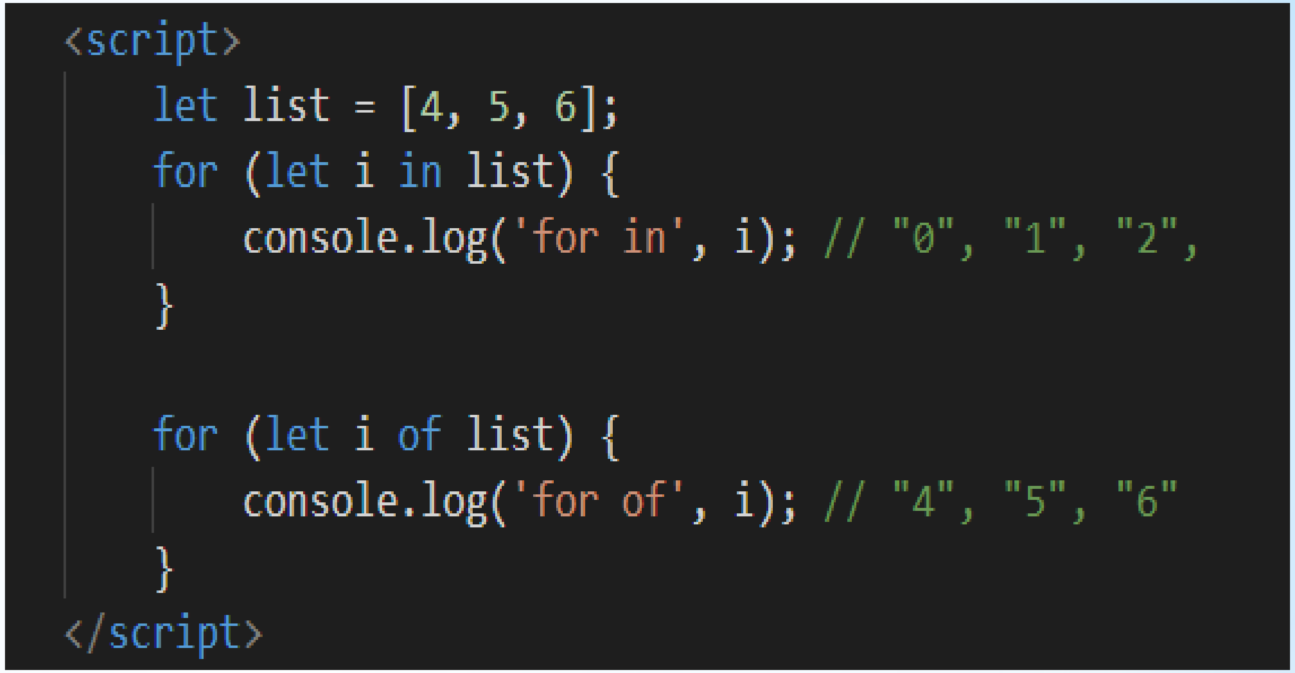
```
let { a, b, ...rest } = { a: 10, b: 20, c: 30, d: 40 }
console.log(a); // 10
console.log(b); // 20
console.log(rest); // { c: 30, d: 40 }
```

for...in statement

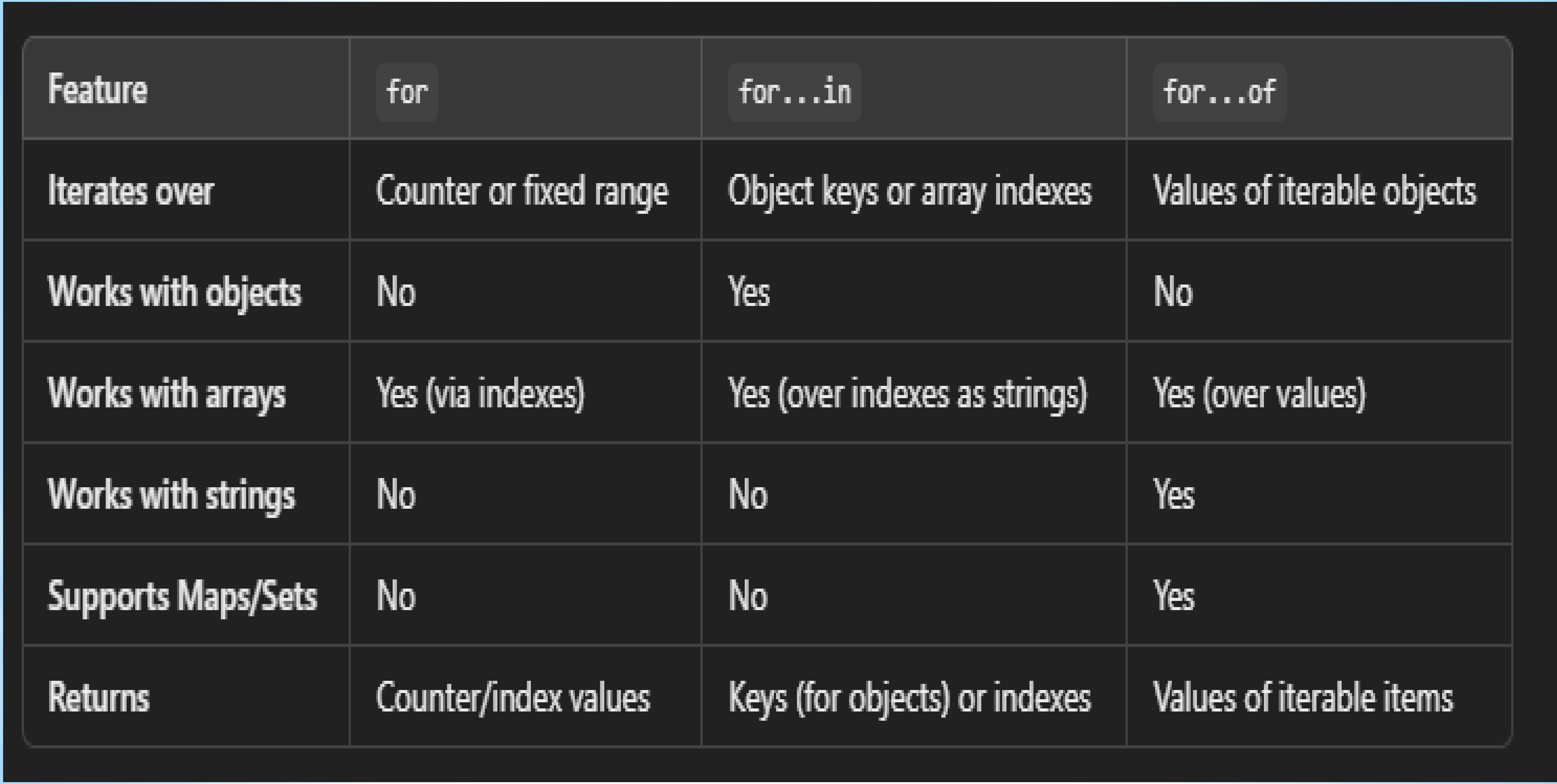
- The for...in statement iterates a specified variable over all the enumerable properties of an object. For each distinct property, JavaScript executes the specified statements.

*For in Vs For of

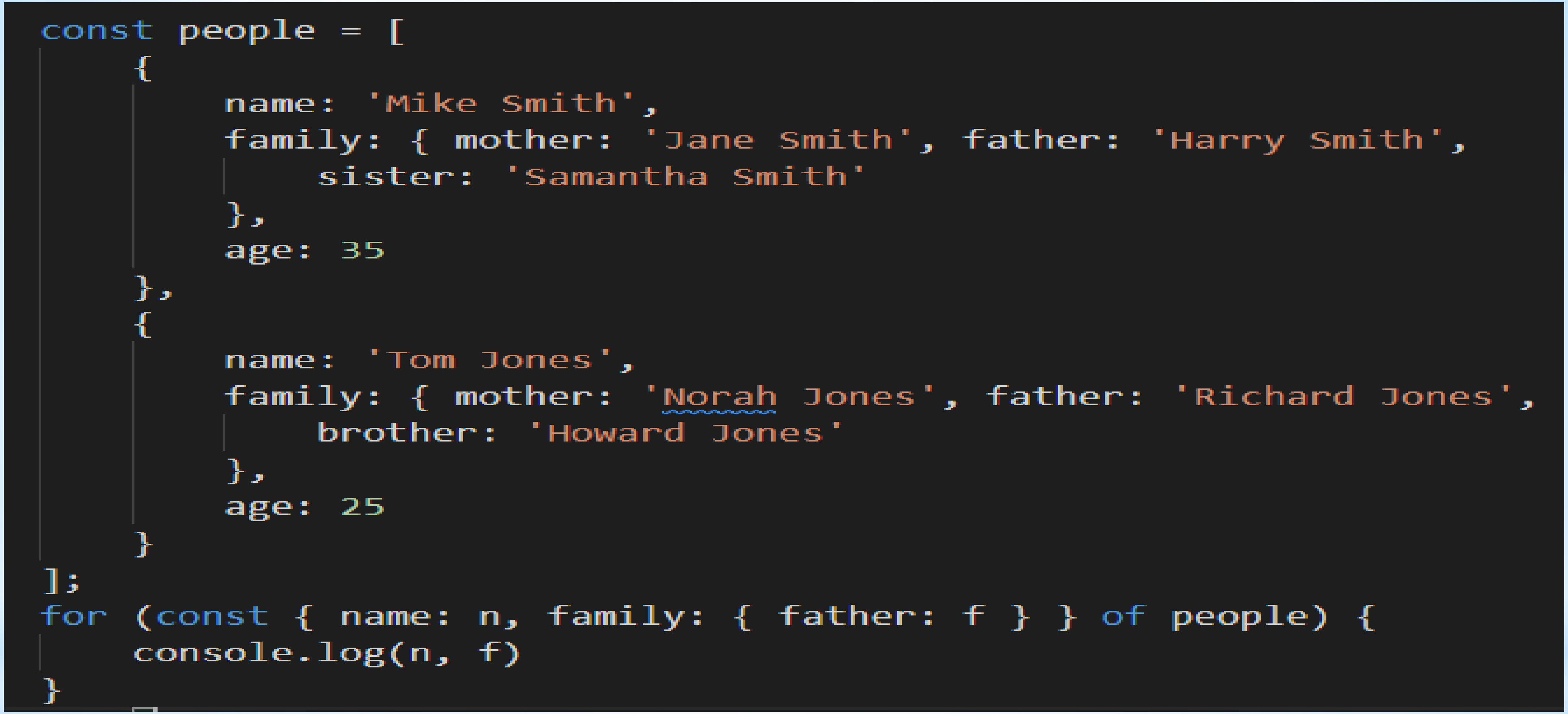
- *for..in iterates over all enumerable property keys of an object
- *for..of iterates over the values of an iterable object. Examples of iterable objects are arrays, strings, and NodeLists.



```
<script>
    let list = [4, 5, 6];
    for (let i in list) {
        console.log('for in', i); // "0", "1", "2",
    }

    for (let i of list) {
        console.log('for of', i); // "4", "5", "6"
    }
</script>
```



Feature	`for`	`for...in`	`for...of`
Iterates over	Counter or fixed range	Object keys or array indexes	Values of iterable objects
Works with objects	No	Yes	No
Works with arrays	Yes (via indexes)	Yes (over indexes as strings)	Yes (over values)
Works with strings	No	No	Yes
Supports Maps/Sets	No	No	Yes
Returns	Counter/index values	Keys (for objects) or indexes	Values of iterable items



```
const people = [
    {
        name: 'Mike Smith',
        family: { mother: 'Jane Smith', father: 'Harry Smith',
            sister: 'Samantha Smith'
        },
        age: 35
    },
    {
        name: 'Tom Jones',
        family: { mother: 'Norah Jones', father: 'Richard Jones',
            brother: 'Howard Jones'
        },
        age: 25
    }
];
for (const { name: n, family: { father: f } } of people) {
    console.log(n, f)
}
```

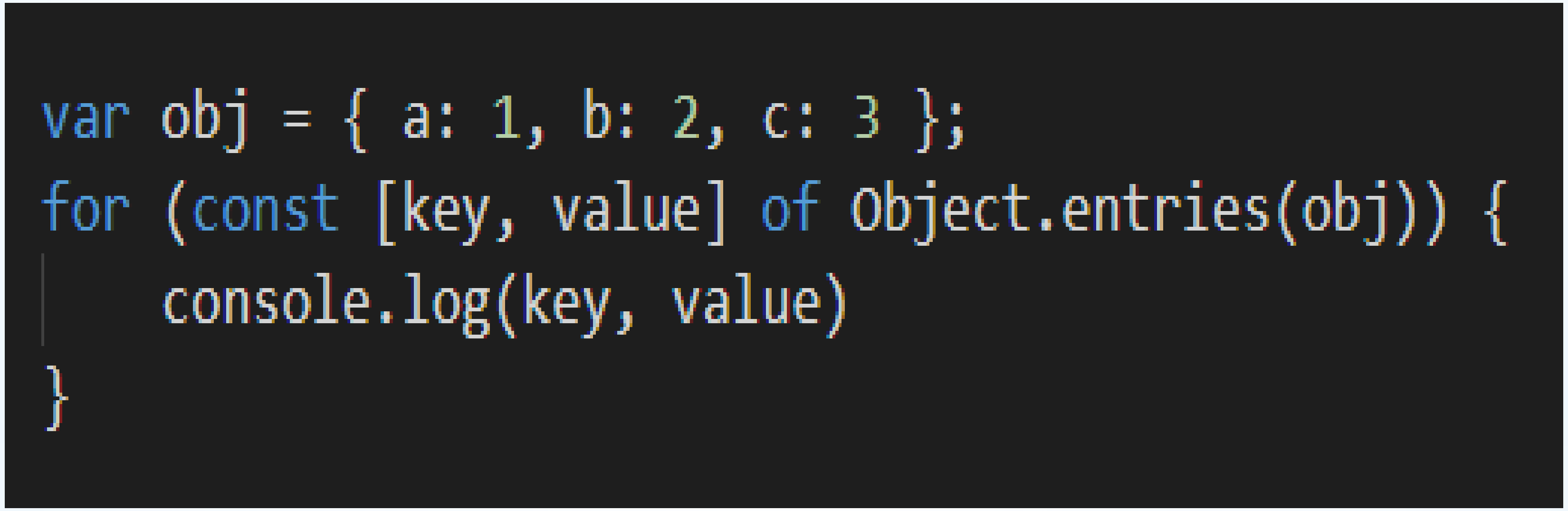
For of iteration and destructuring

- Object.keys(o) This method returns an array with all the own (not in the prototype chain) enumerable properties' names ("keys") of an object o.
- Object.getOwnPropertyNames(o) This method returns an array containing all own properties' names (enumerable or not) of an object o.

Enumerate the properties of an object

Object.entries

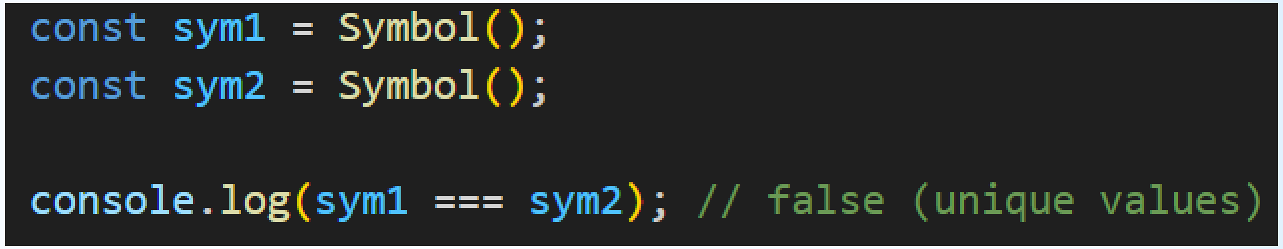
- The Object.entries() method returns an array of a given object's own enumerable string-keyed property [key, value] pairs.



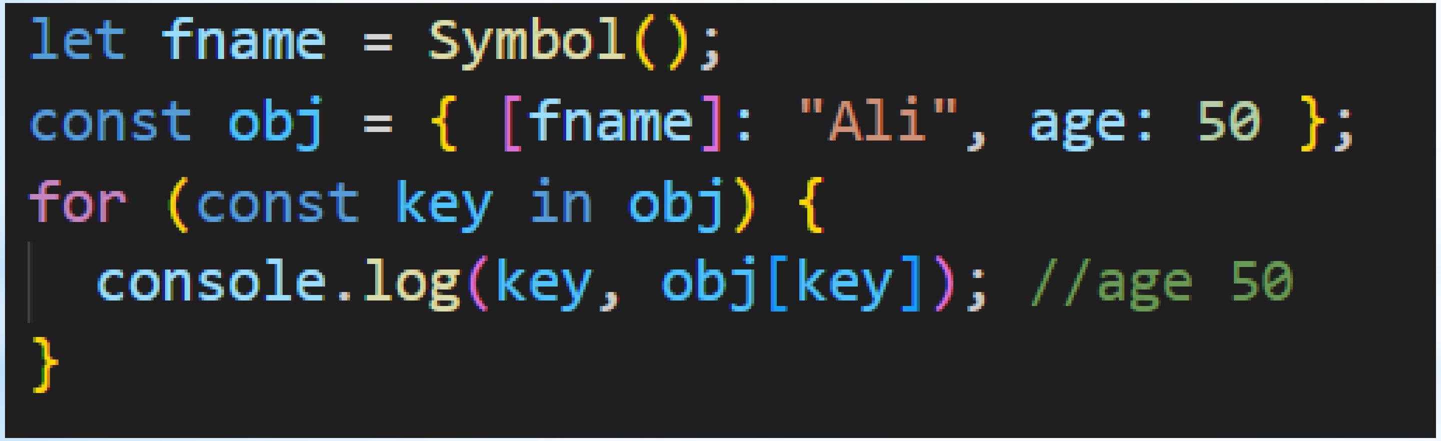
```
var obj = { a: 1, b: 2, c: 3 };
for (const [key, value] of Object.entries(obj)) {
    console.log(key, value)
}
```

Symbol

- What is it?
- Symbol is a new primitive data type introduced in ES6.
- It creates a unique value, primarily used as object keys to avoid naming collisions.



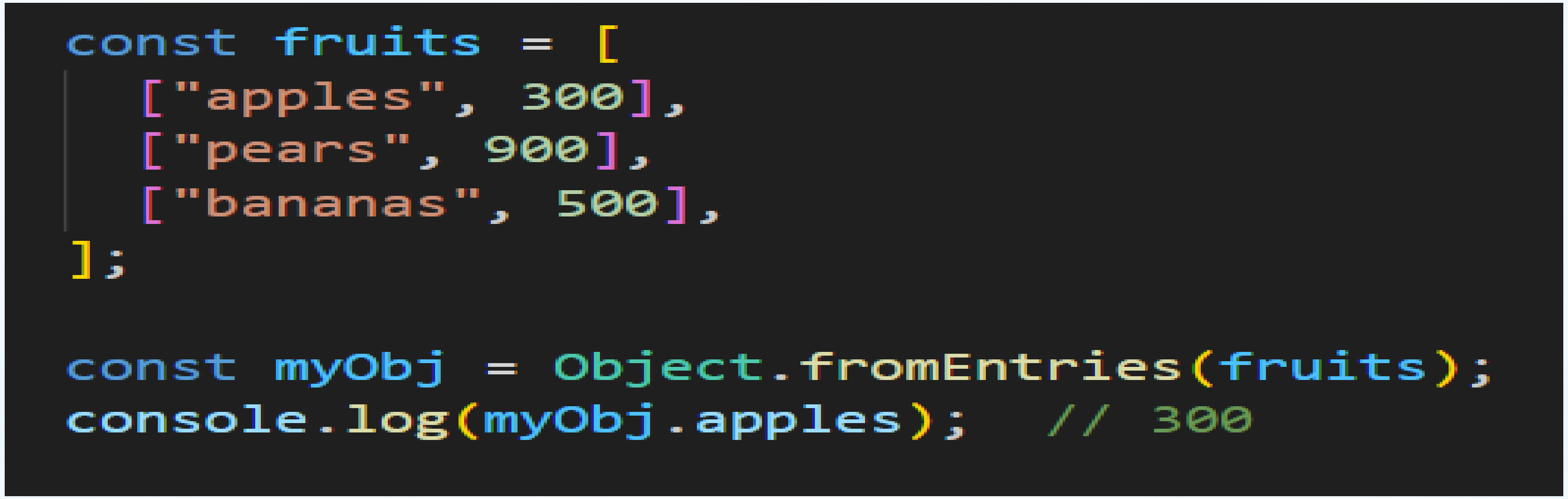
```
const sym1 = Symbol();
const sym2 = Symbol();

console.log(sym1 === sym2); // false (unique values)
```



```
let fname = Symbol();
const obj = { [fname]: "Ali", age: 50 };
for (const key in obj) {
  console.log(key, obj[key]); //age 50
}
```

Object fromEntries()

- The fromEntries() method creates an object from iterable key / value pairs.



```
const fruits = [
  ["apples", 300],
  ["pears", 900],
  ["bananas", 500],
];

const myObj = Object.fromEntries(fruits);
console.log(myObj.apples);   // 300
```