*ECMAScript 6

Ahmed Elashry
aelashry@outlook.com

*Day2

Array Methods

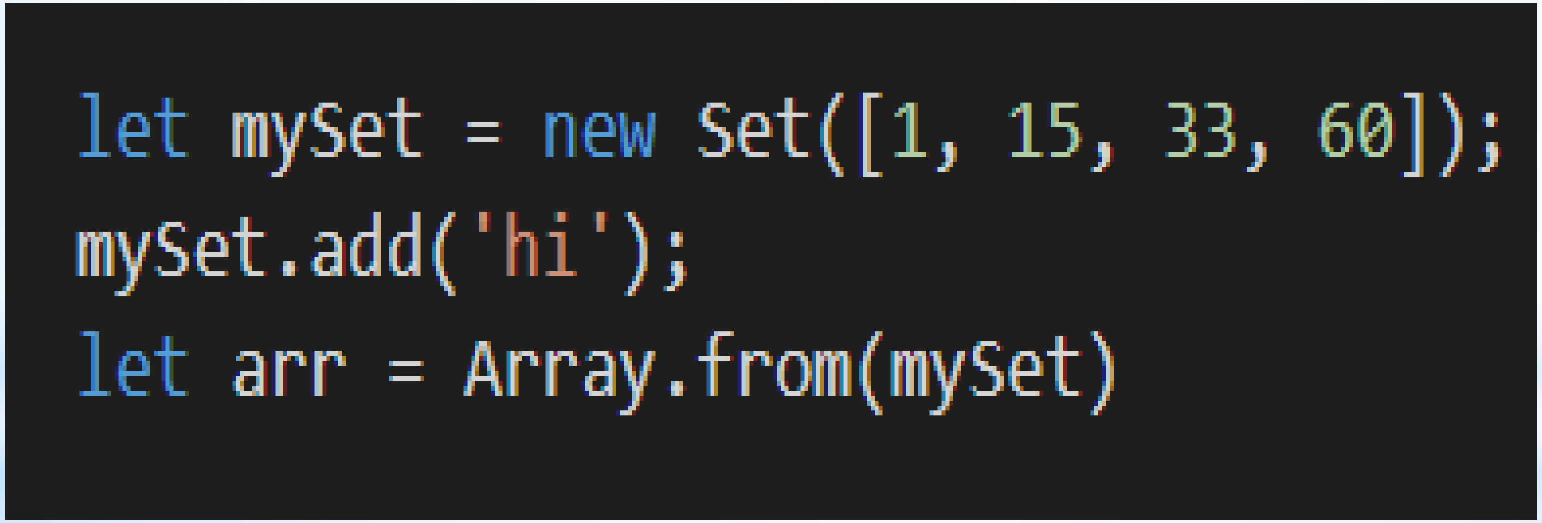
- find() method returns the value of the first element in the provided. If no values satisfy the testing function, undefined is returned.
- findIndex() returns the index of the first element in the. Otherwise, it returns -1, indicating that no element passed the test.
- filter() Creates a new array with all of the elements of this array for which the provided filtering function returns true.
- forEach() Calls a function for each element in the array.
- map() creates a new array populated with the results of calling a provided function on every element in the calling array.
- includes() This allows us to check if an element is present in an array:

- Every() tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value.
- some() tests whether at least one element in the array passes the test implemented by the provided function
- with() change the value of a given index. It returns a new array with the element at the given index replaced with the given value.
- from() method returns an Array object from any object with a length property or any iterable object.
- flat() method creates a new array by flattening a nested array.

Array Methods

Set

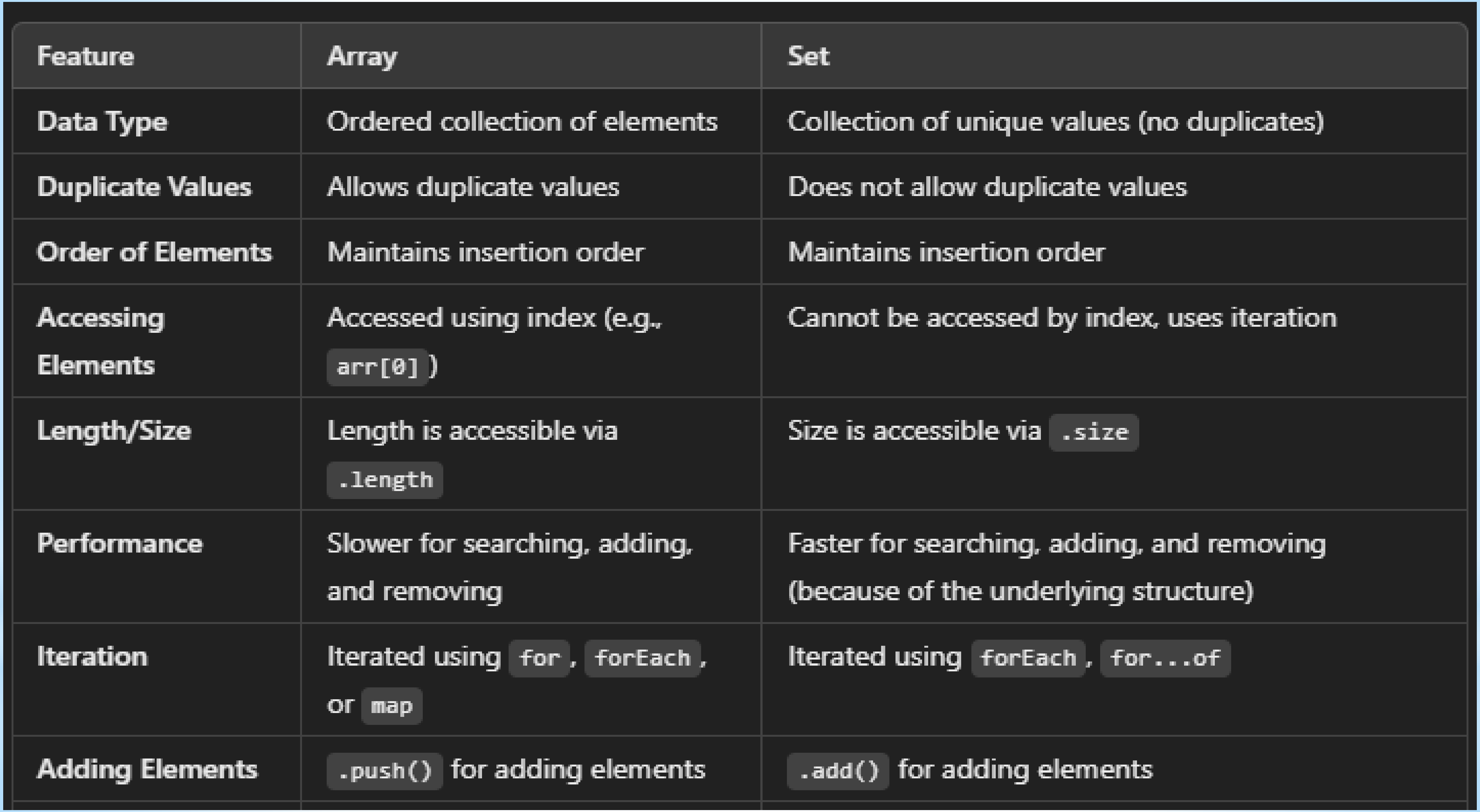
- Set objects are collections of values. You can iterate its elements in insertion order. A value in a Set may only occur once; it is unique in the Set's collection.
- Set constructor accepts to convert to Array in the other direction .



```
let mySet = new Set([1, 15, 33, 60]);
mySet.add('hi');
let arr = Array.from(mySet)
```

- Checking whether an element exists in a collection using indexOf for arrays is slow.
- Set objects let you delete elements by their value. With an array you would have to splice based on an element's index.
- The value NaN cannot be found with indexOf in an array.
- Set objects store unique values; you don't have to keep track of duplicates by yourself.

Array and Set compared



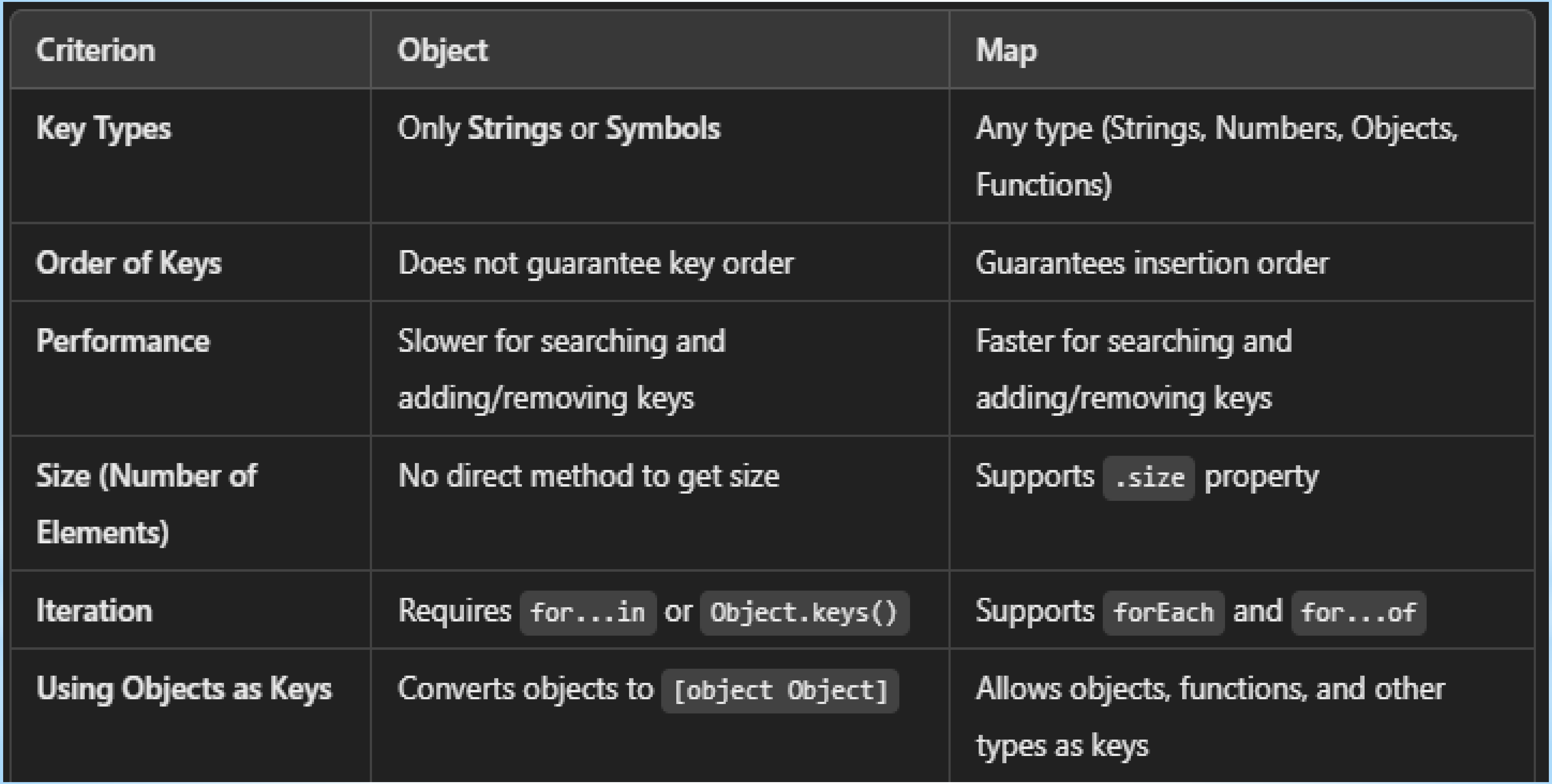
Feature	Array	Set
Data Type	Ordered collection of elements	Collection of unique values (no duplicates)
Duplicate Values	Allows duplicate values	Does not allow duplicate values
Order of Elements	Maintains insertion order	Maintains insertion order
Accessing Elements	Accessed using index (e.g., `arr[0]`)	Cannot be accessed by index, uses iteration
Length/Size	Length is accessible via `.length`	Size is accessible via `.size`
Performance	Slower for searching, adding, and removing	Faster for searching, adding, and removing (because of the underlying structure)
Iteration	Iterated using `for`, `forEach`, or `map`	Iterated using `forEach`, `for...of`
Adding Elements	`.push()` for adding elements	`.add()` for adding elements

Maps

- This data structure enables mapping a key to a value.
- The Map object is a simple key/value pair. Keys and values in a map may be primitive or objects.
- basic Map operations
  - The set() function sets the value for the key This function returns the Map object.
  - The has() function returns a boolean value indicating whether the specified key is found in the Map object. This function takes a key as parameter.
  - The get() function is used to retrieve the value corresponding to the specified key.
  - The clear() Removes all key/value pairs from the Map object.
  - The delete(key) Removes any value associated to the key
  - The entries() Returns a new Iterator object that contains **an array of** [key, value]
  - The keys() / values() Returns a new Iterator object that contains **an array of** [key, value] for each element

Object and Map compared

- The keys of an Object are Strings, where they can be of any value for a Map.
- You can get the size of a Map easily while you have to manually keep track of size for an Object.
- The iteration of maps is in insertion order of the elements.



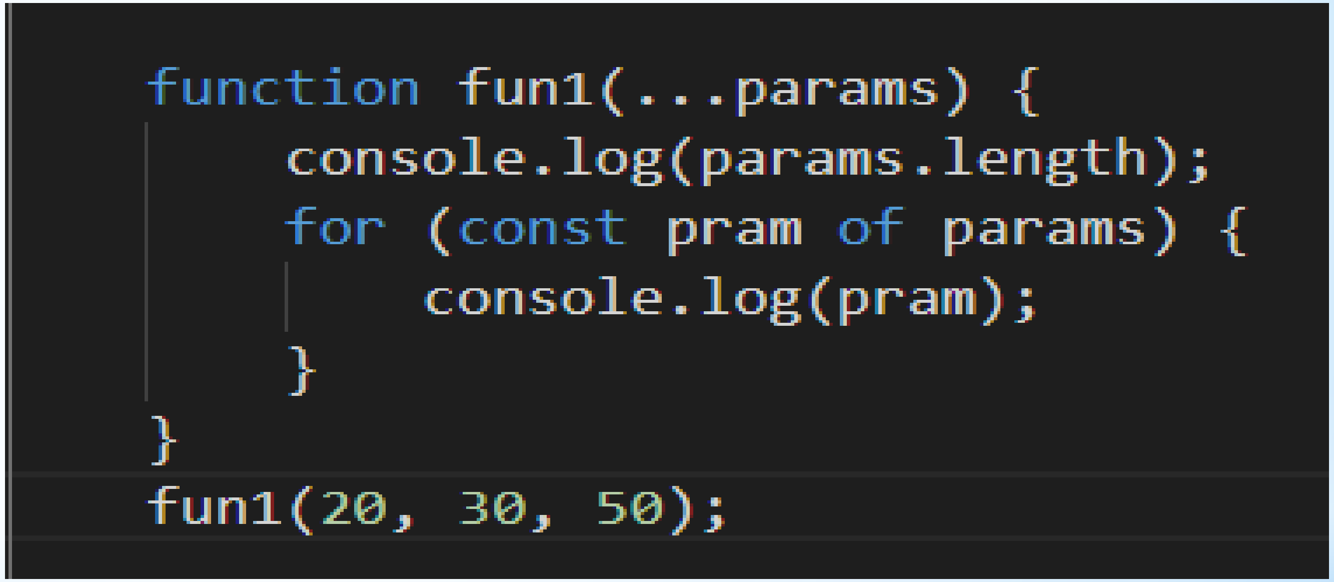
Criterion	Object	Map
Key Types	Only **Strings** or **Symbols**	Any type (Strings, Numbers, Objects, Functions)
Order of Keys	Does not guarantee key order	Guarantees insertion order
Performance	Slower for searching and adding/removing keys	Faster for searching and adding/removing keys
Size (Number of Elements)	No direct method to get size	Supports `.size` property
Iteration	Requires `for...in` or `Object.keys()`	Supports `forEach` and `for...of`
Using Objects as Keys	Converts objects to `[object Object]`	Allows objects, functions, and other types as keys

- Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure—a set of statements that performs a task or calculates a value. To use a function, you must define it somewhere in the scope from which you wish to call it.

Functions

Function parameters

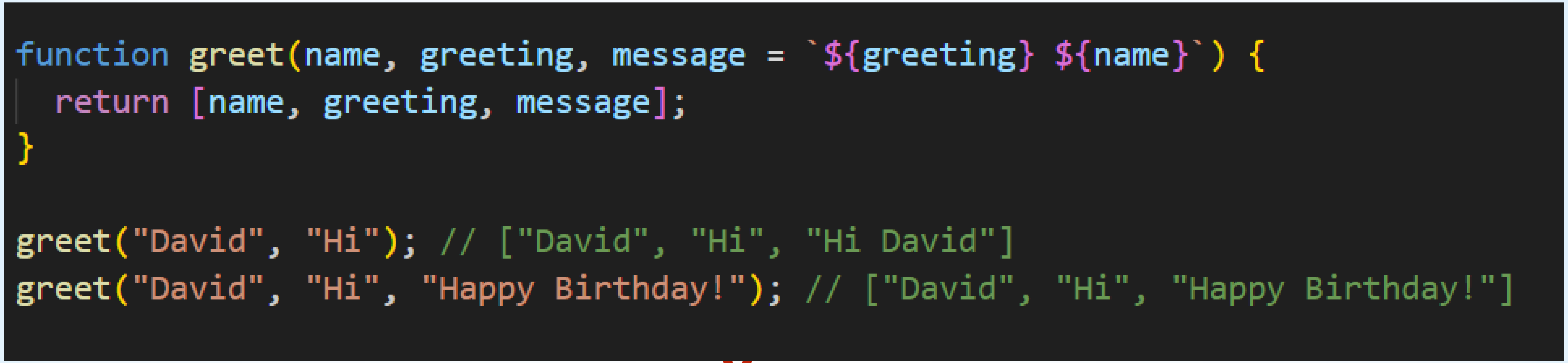
- function allows the parameters to be initialized with default values
- Rest parameters doesn’t restrict the number of values that you can pass to a function.



```
function fun1(...params) {
    console.log(params.length);
    for (const pram of params) {
        console.log(pram);
    }
}
fun1(20, 30, 50);
```

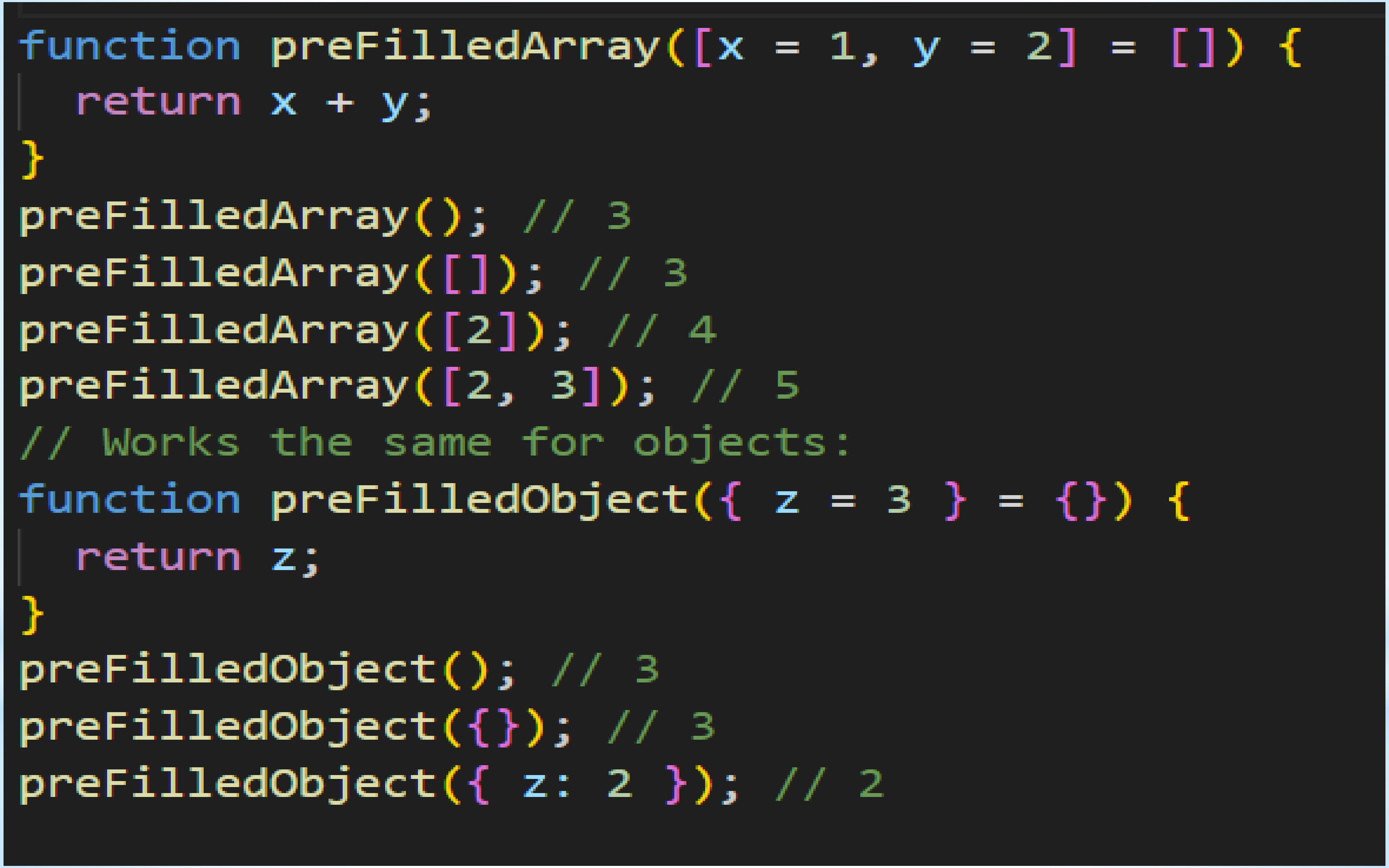
Default Parameter

- Using named parameters for optional settings makes it easier to understand how a function should be invoked.
- It’s okay to omit some options when invoking a function with named parameters.



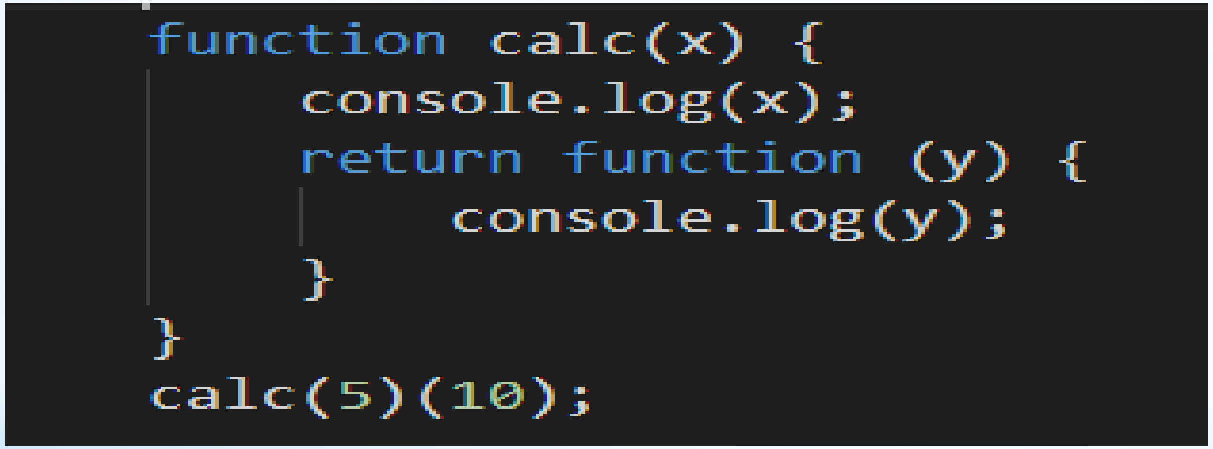
```
function greet(name, greeting, message = `${greeting} ${name}`) {
  return [name, greeting, message];
}

greet("David", "Hi"); // ["David", "Hi", "Hi David"]
greet("David", "Hi", "Happy Birthday!"); // ["David", "Hi", "Happy Birthday!"]
```



```
function preFilledArray([x = 1, y = 2] = []) {
  return x + y;
}
preFilledArray(); // 3
preFilledArray([]); // 3
preFilledArray([2]); // 4
preFilledArray([2, 3]); // 5
// Works the same for objects:
function preFilledObject({ z = 3 } = {}) {
  return z;
}
preFilledObject(); // 3
preFilledObject({}); // 3
preFilledObject({ z: 2 }); // 2
```

- You can nest a function within a function. The nested (inner) function is private to its containing (outer) function. It also forms a *closure*. A closure is an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).



```
function calc(x) {
    console.log(x);
    return function (y) {
        console.log(y);
    }
}
calc(5)(10);
```

Nested functions

- Immediately Invoked Function Expressions (IIFEs) can be used to avoid variable hoisting from within blocks. It allows public access to methods while retaining privacy for variables defined within the function. This pattern is called as a self-executing anonymous function

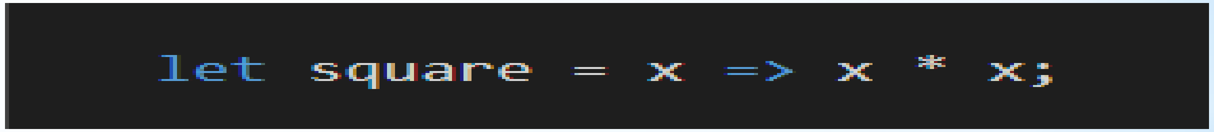
*Immediately Invoked Function Expression

- The Function() constructor expects any number of string arguments. The last argument is the body of the function
- The Function() constructor is not passed any argument that specifies a name for the function it creates.

Function Constructor

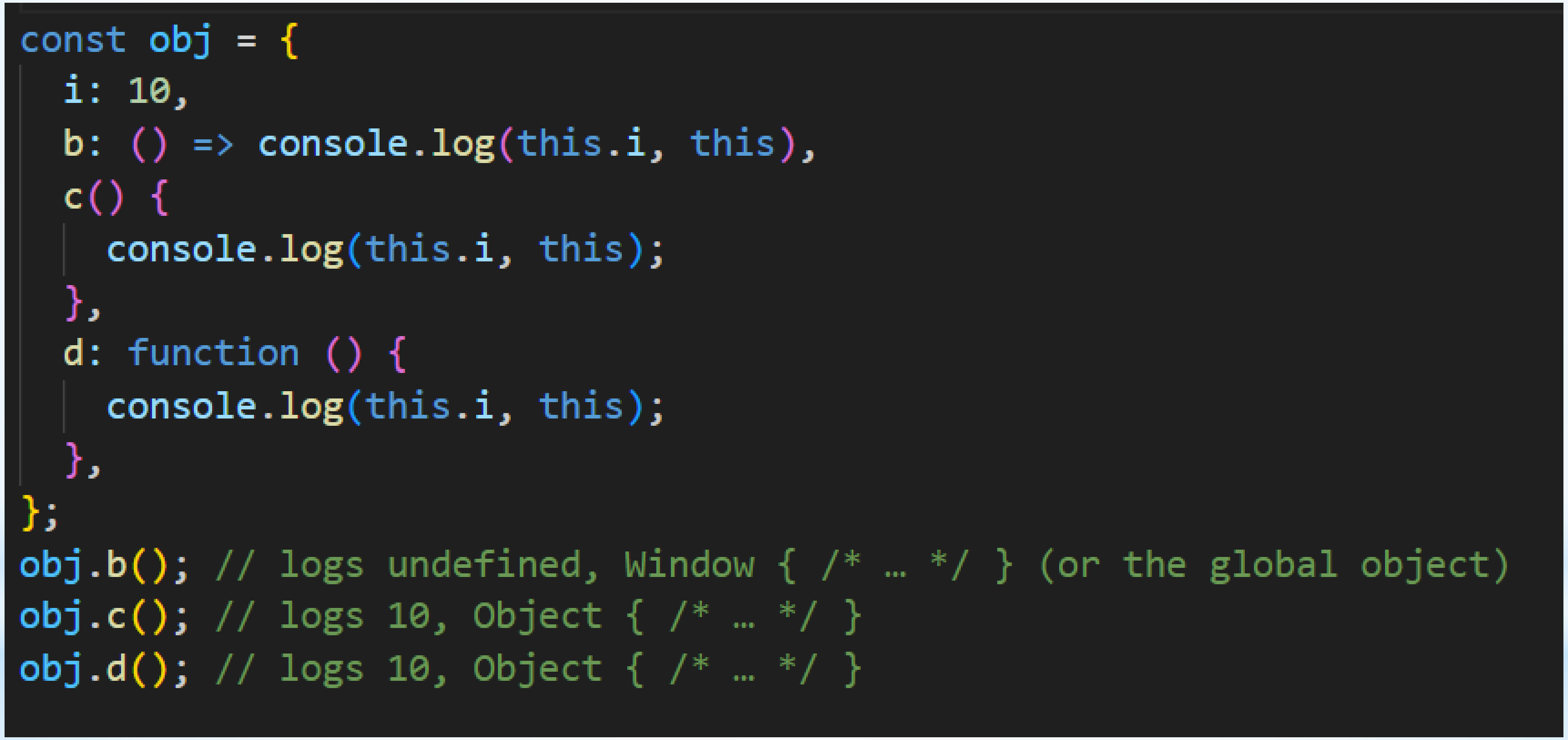
Lambda Functions(Arrow functions)

- Lambda refers to anonymous functions in programming. Lambda functions are a concise mechanism to represent anonymous functions. These functions are also called as Arrow functions.
- There are 3 parts to a Lambda function −
  - **Parameters** − A function may optionally have parameters.
  - The **fat arrow notation/lambda notation** (=>): It is also called as the **goes to** operator.
  - **Statements** − Represents the function’s instruction set.



```
let square = x => x * x;
```

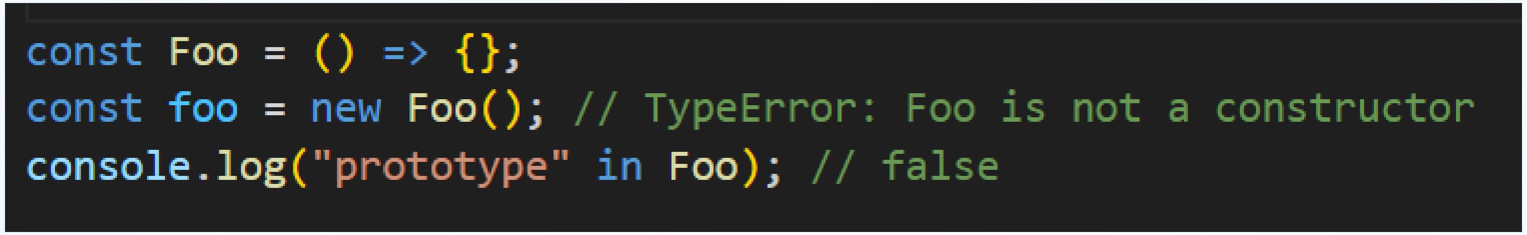
- Arrow function expressions should only be used for non-method functions because they do not have their own this. Let's see what happens when we try to use them as methods:



```
const obj = {
  i: 10,
  b: () => console.log(this.i, this),
  c() {
    console.log(this.i, this);
  },
  d: function () {
    console.log(this.i, this);
  },
};
obj.b(); // logs undefined, Window { /* … */ } (or the global object)
obj.c(); // logs 10, Object { /* … */ }
obj.d(); // logs 10, Object { /* … */ }
```

Lambda Functions(Arrow functions)

- Cannot be used as constructors
- Arrow functions cannot be used as constructors and will throw an error when called with new. They also do not have a prototype property.

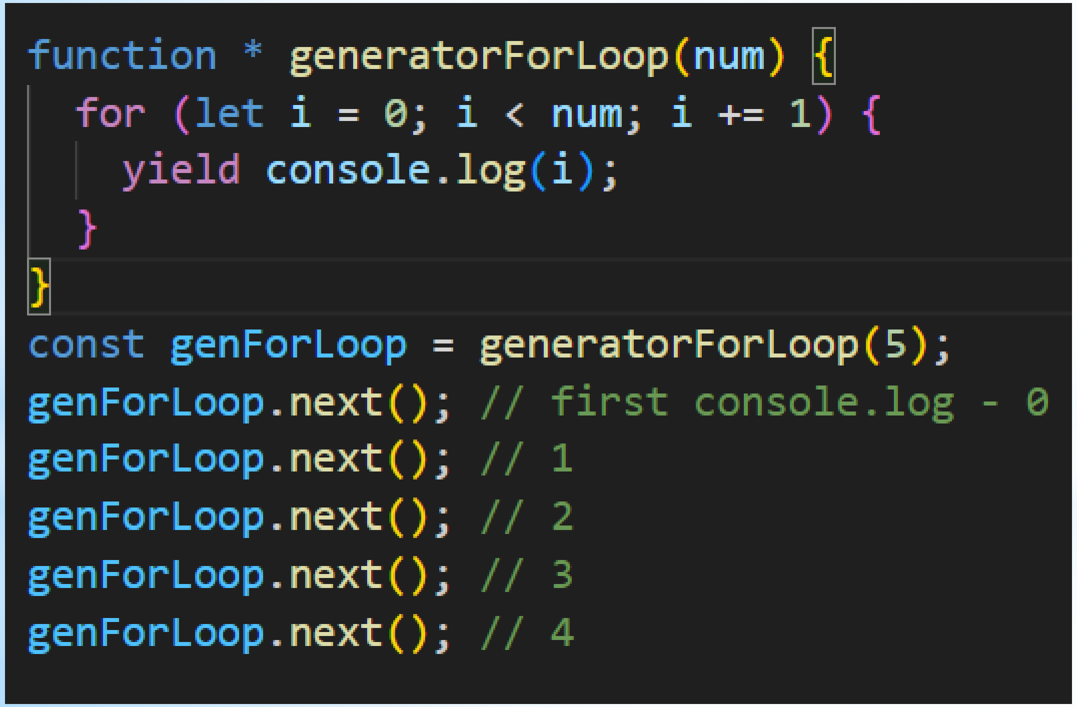


```
const Foo = () => {};
const foo = new Foo(); // TypeError: Foo is not a constructor
console.log("prototype" in Foo); // false
```

Lambda Functions(Arrow functions

Generator Functions

- When a normal function is invoked, the control rests with the function called until it returns. With generators in ES6, the caller function can now control the execution of a called function. A generator is like a regular function



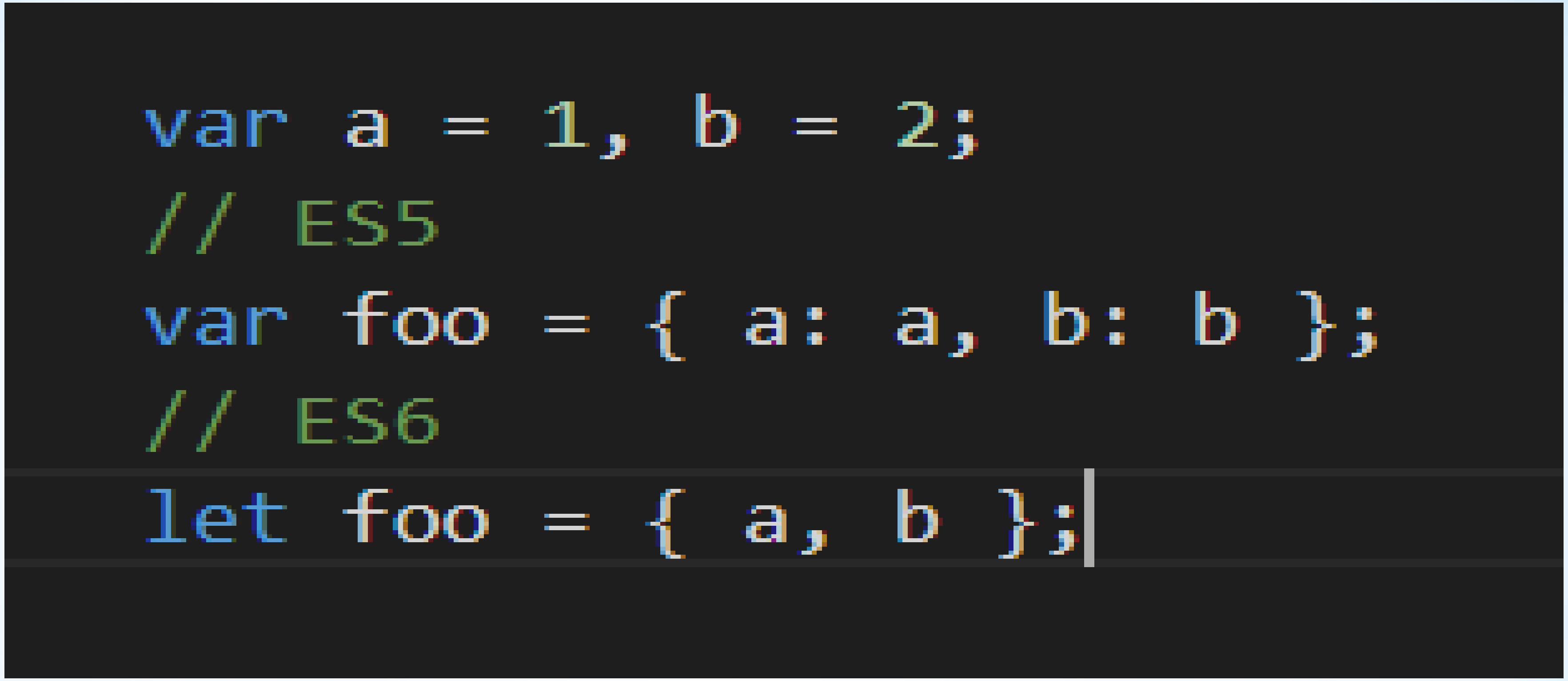
```
function * generatorForLoop(num) {
  for (let i = 0; i < num; i += 1) {
    yield console.log(i);
  }
}
const genForLoop = generatorForLoop(5);
genForLoop.next(); // first console.log - 0
genForLoop.next(); // 1
genForLoop.next(); // 2
genForLoop.next(); // 3
genForLoop.next(); // 4
```

*Objects

- An object is an instance which contains a set of key value pairs. Unlike primitive data types, objects can represent multiple or complex values and can change over their life time. The values can be scalar values or functions or even array of other objects.

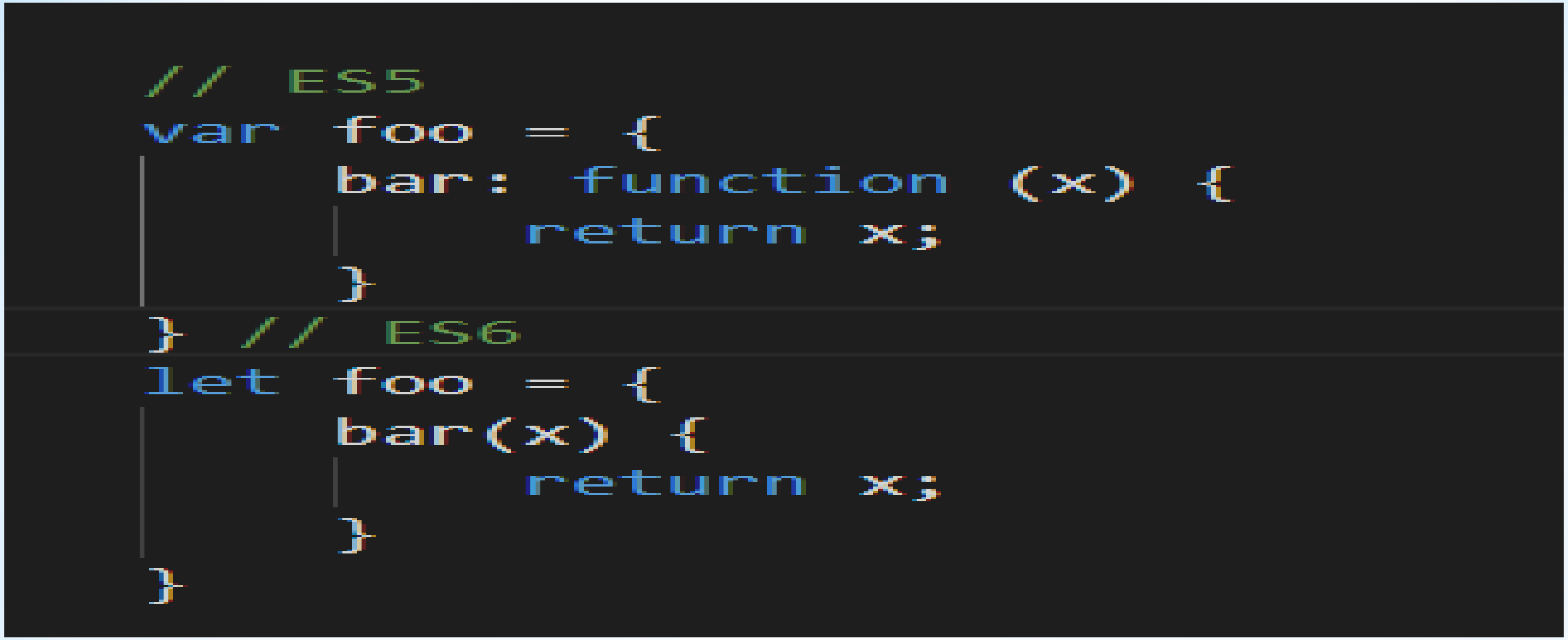
- The contents of an object are called **properties** (or members), and properties consist of a **name** (or key) and **value**. Property names must be strings or symbols, and values can be any type (including other objects).

- Unassigned properties of an object are undefined (and not null).



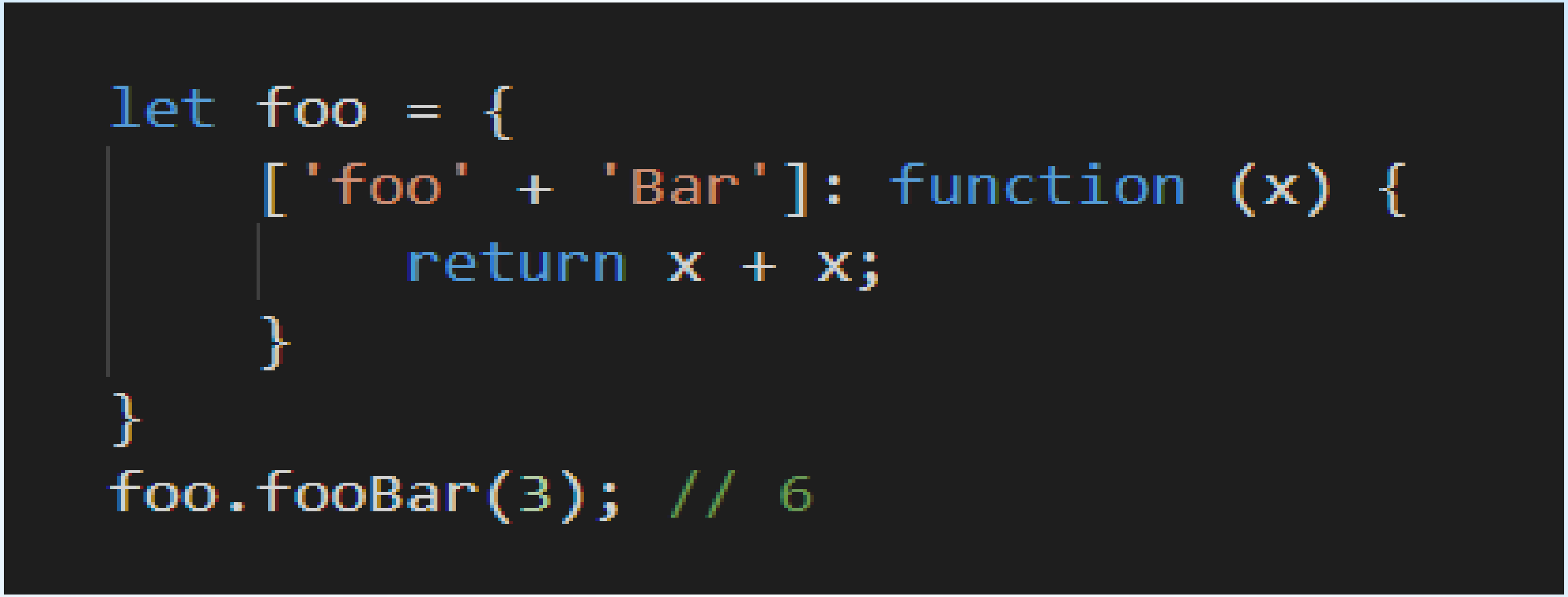
```
var a = 1, b = 2;
// ES5
var foo = { a: a, b: b };
// ES6
let foo = { a, b };
```

*Object Property shorthand



```
// ES5
var foo = {
    bar: function (x) {
        return x;
    }
} // ES6
let foo = {
    bar(x) {
        return x;
    }
}
```

*Object Compact method syntax

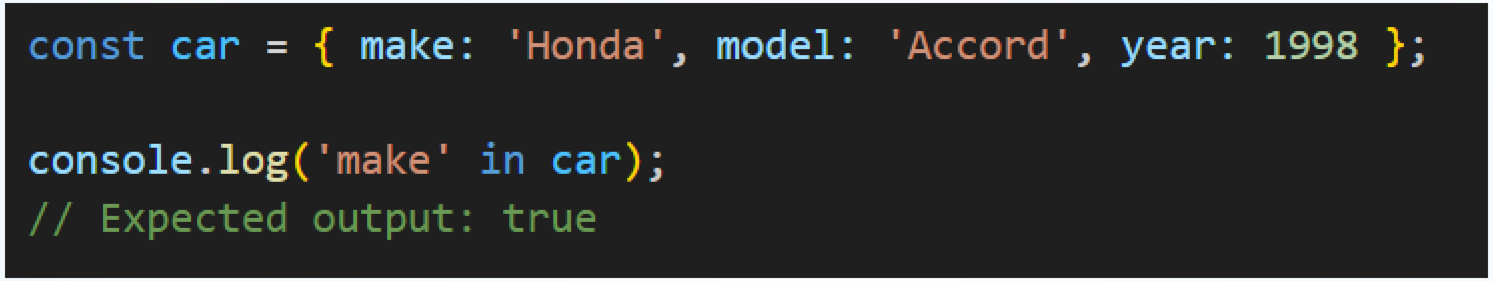


```
let foo = {
    ['foo' + 'Bar']: function (x) {
        return x + x;
    }
}
foo.fooBar(3); // 6
```

*Object Computed properties

in

- The in operator returns true if the specified property is in the specified object or its prototype chain.



```
const car = { make: 'Honda', model: 'Accord', year: 1998 };

console.log('make' in car);
// Expected output: true
```

- The Object.freeze() method freezes an object. A frozen object can no longer be changed; freezing an object prevents new properties from being added to it, existing properties from being removed, prevents changing the enumerability, configurability, or writability of existing properties, and prevents the values of existing properties from being changed

Object.freeze() Function

- The Object.seal() method seals an object, preventing new properties from being added to it Values of present properties can still be changed as long as they are writable.

*Object.seal() Function

- A getter is a method that gets the value of a specific property. A setter is a method that sets the value of a specific property.

Defining getters and setters

- Configurable true if the property may be deleted from the corresponding object. Defaults to false.
- Enumerable true if and only if this property shows up during enumeration of the properties on the corresponding object. Defaults to false.
- Value Can be any valid JavaScript value (number, object, function, etc). Defaults to undefined.
- Writable true if the value associated with the property may be changed with an assignment operator. Defaults to false.

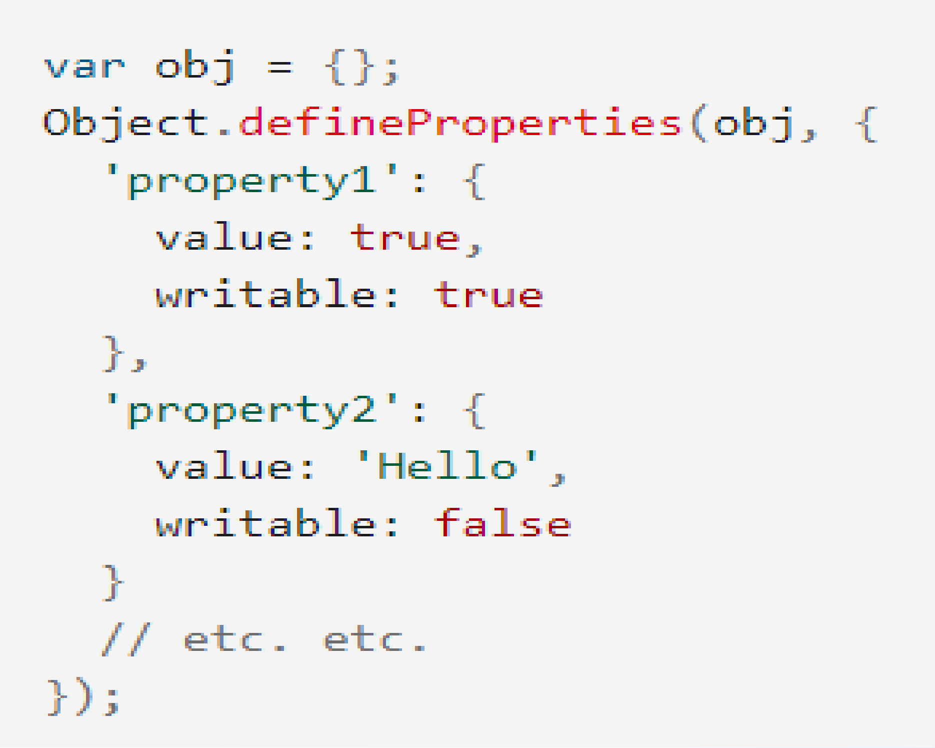
*Object.defineProperty()

- Get The return value will be used as the value of the property. Defaults to undefined.
- Set with this set to the object through which the property is assigned. Defaults to undefined.

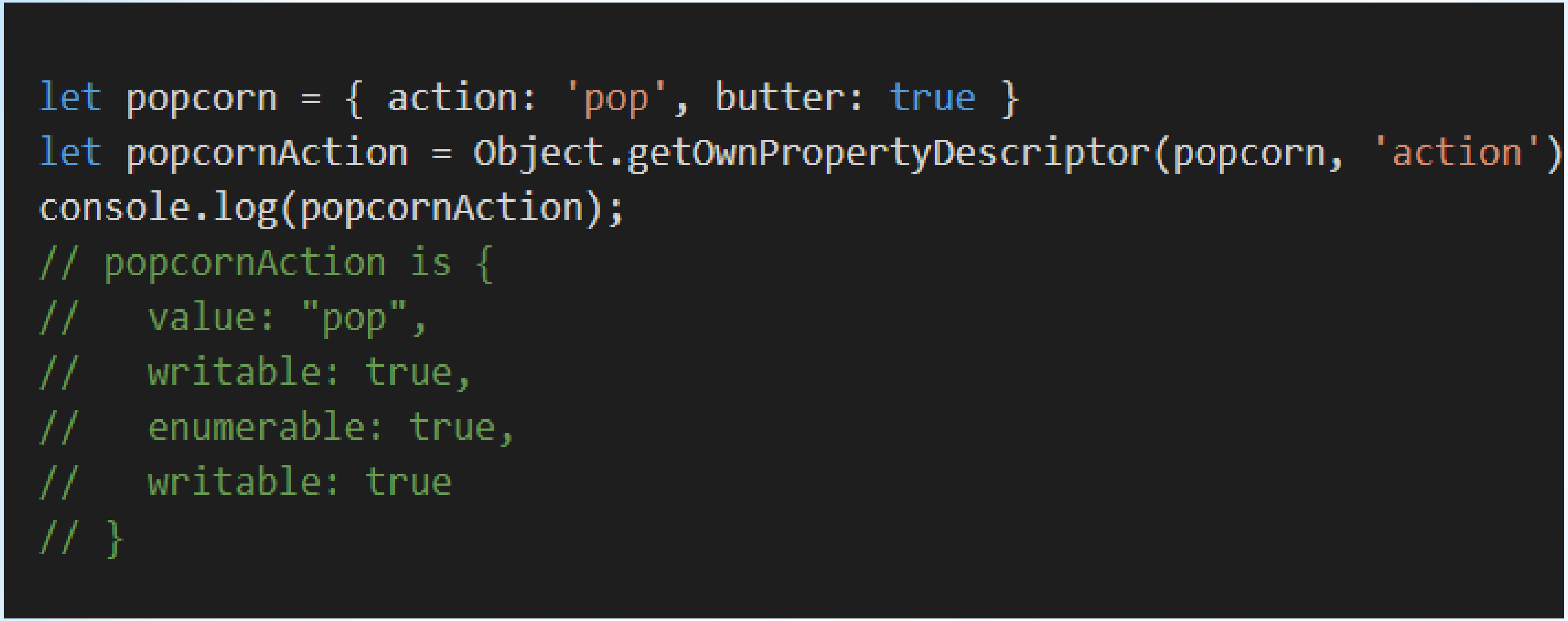
Object.defineProperty()

*Object.defineProperties()

*The Object.defineProperties() method defines new or modifies existing properties directly on an object, returning the object.



```
var obj = {};
Object.defineProperties(obj, {
  'property1': {
    value: true,
    writable: true
  },
  'property2': {
    value: 'Hello',
    writable: false
  }
  // etc. etc.
});
```



```
let popcorn = { action: 'pop', butter: true }
let popcornAction = Object.getOwnPropertyDescriptor(popcorn, 'action')
console.log(popcornAction);
// popcornAction is {
//   value: "pop",
//   writable: true,
//   enumerable: true,
//   writable: true
// }
```

Object.getOwnPropertyDescriptor