levi
nine
Technology Services

ECMASCRIPT

Beograd, 14.11.2019

AGENDA SLIDE

01 ECMAScript – what and why

02 Classes

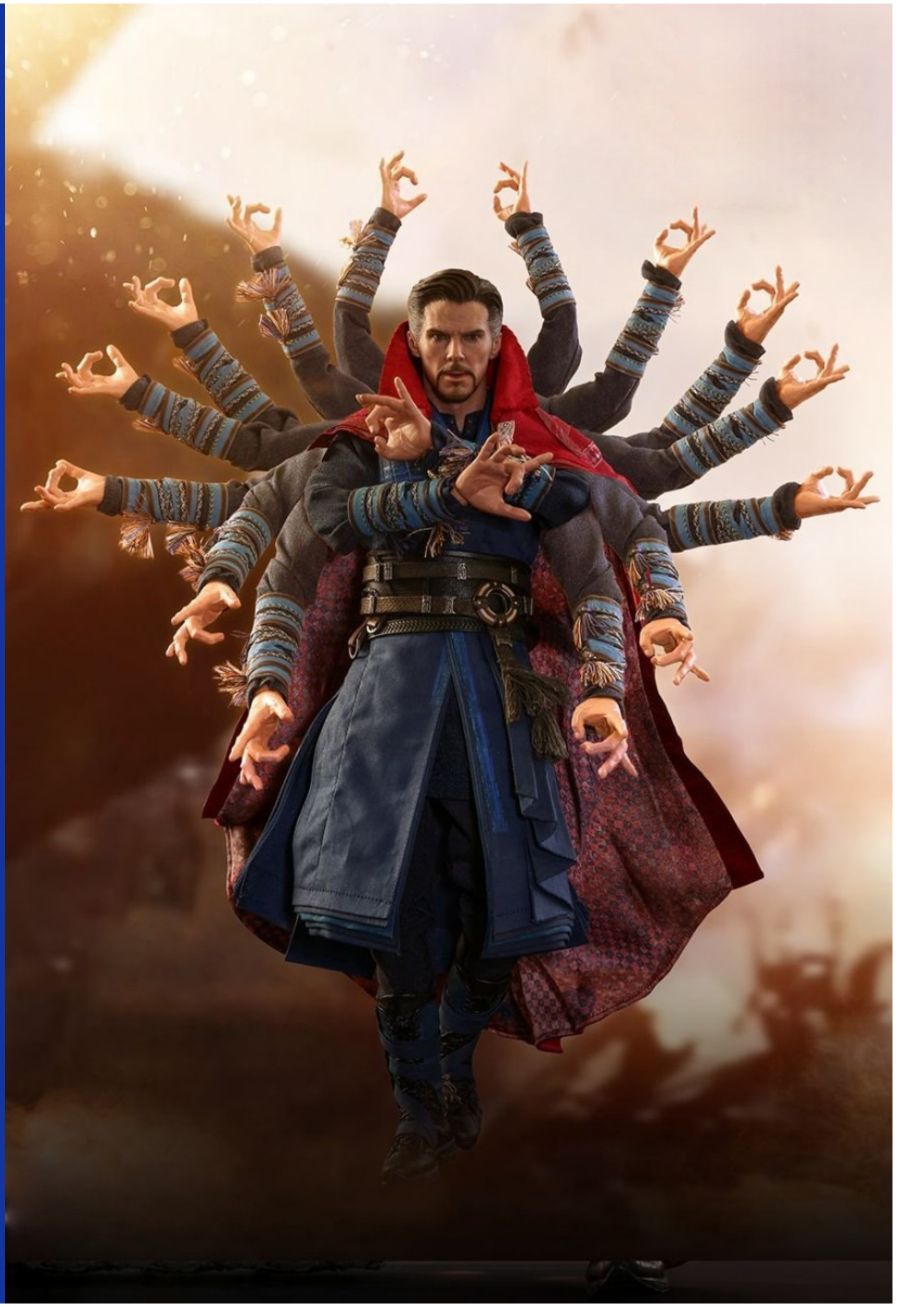
03 Collections

04 Arrow functions

01

ECMASCRIPT

What and why ?



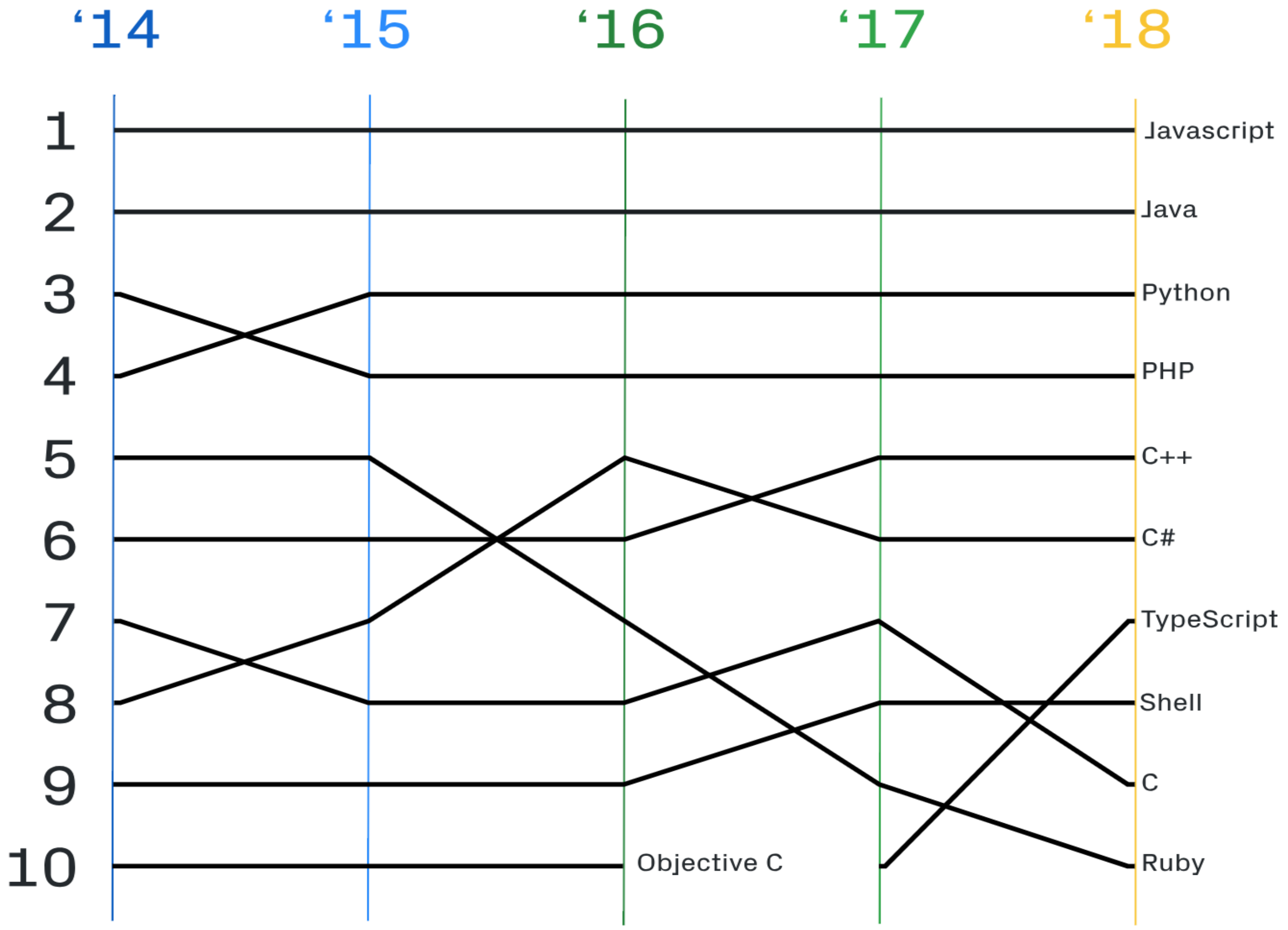
ECMASCRIPT

Netscape - Brendan Eich

Mocha < LiveScript < JavaScript

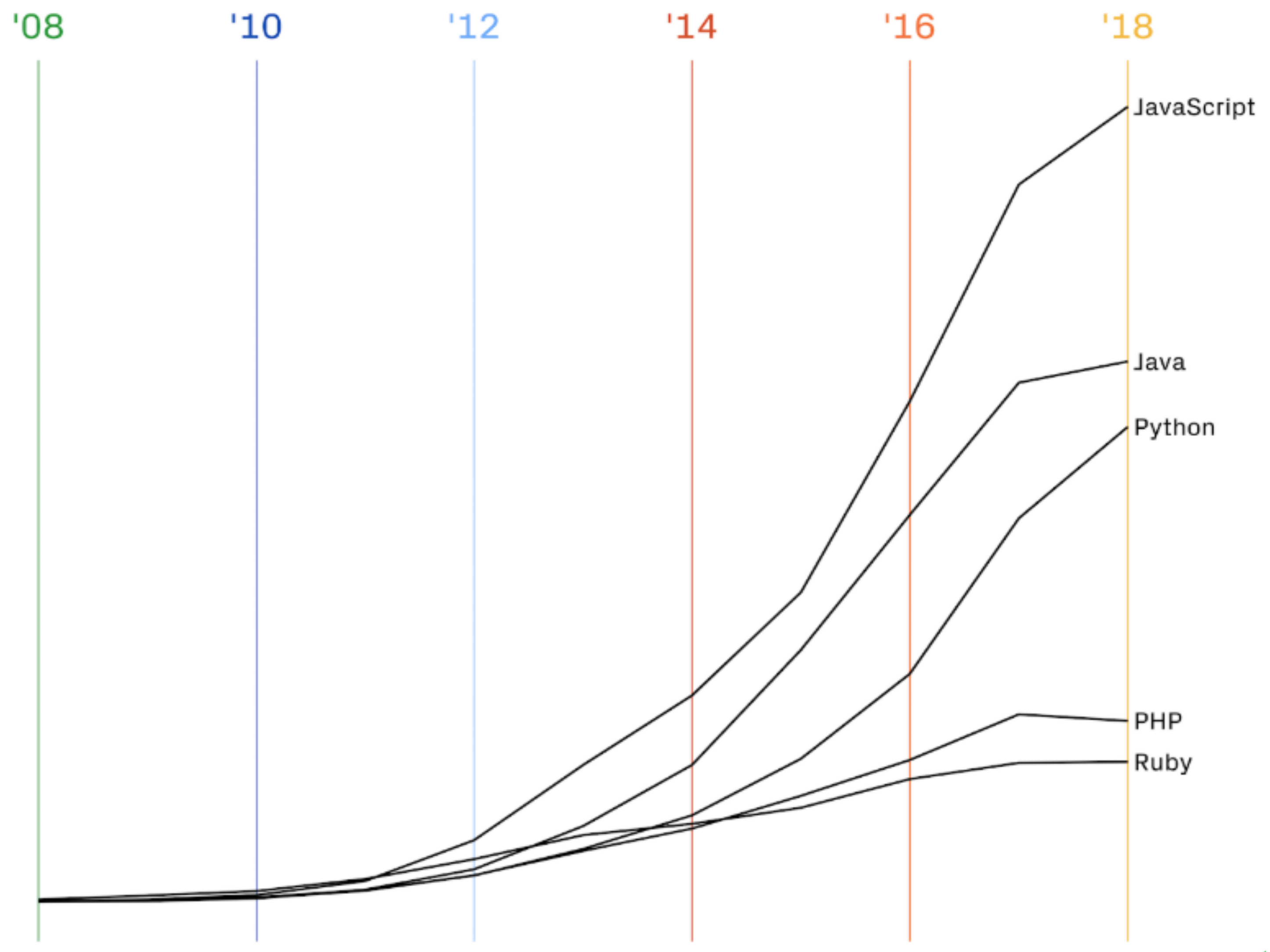
ECMASCRIPT

GitHub contributions



ECMASCRIPT

GitHub repositories



ECMASCRIPT

Standard / Guide the path of JavaScript

ES1, ES2, ES3 – early standards

ES4 – abandoned

ES5 – 2009

ES6/ES2015, ES7/ES2016, ES8/ES2017

ES.Next

ECMASCRIPT

Browser / NodeJS support / Why we need to transpile it

	ES2015	ES2016+
Node	98%	97%
Chrome	98%	97%
Firefox	98%	80%
Edge	96%	40%
Safari	99%	80%
IE	11%	0%

ECMASCRIPT

Browser / NodeJS support / Why we need to transpile it

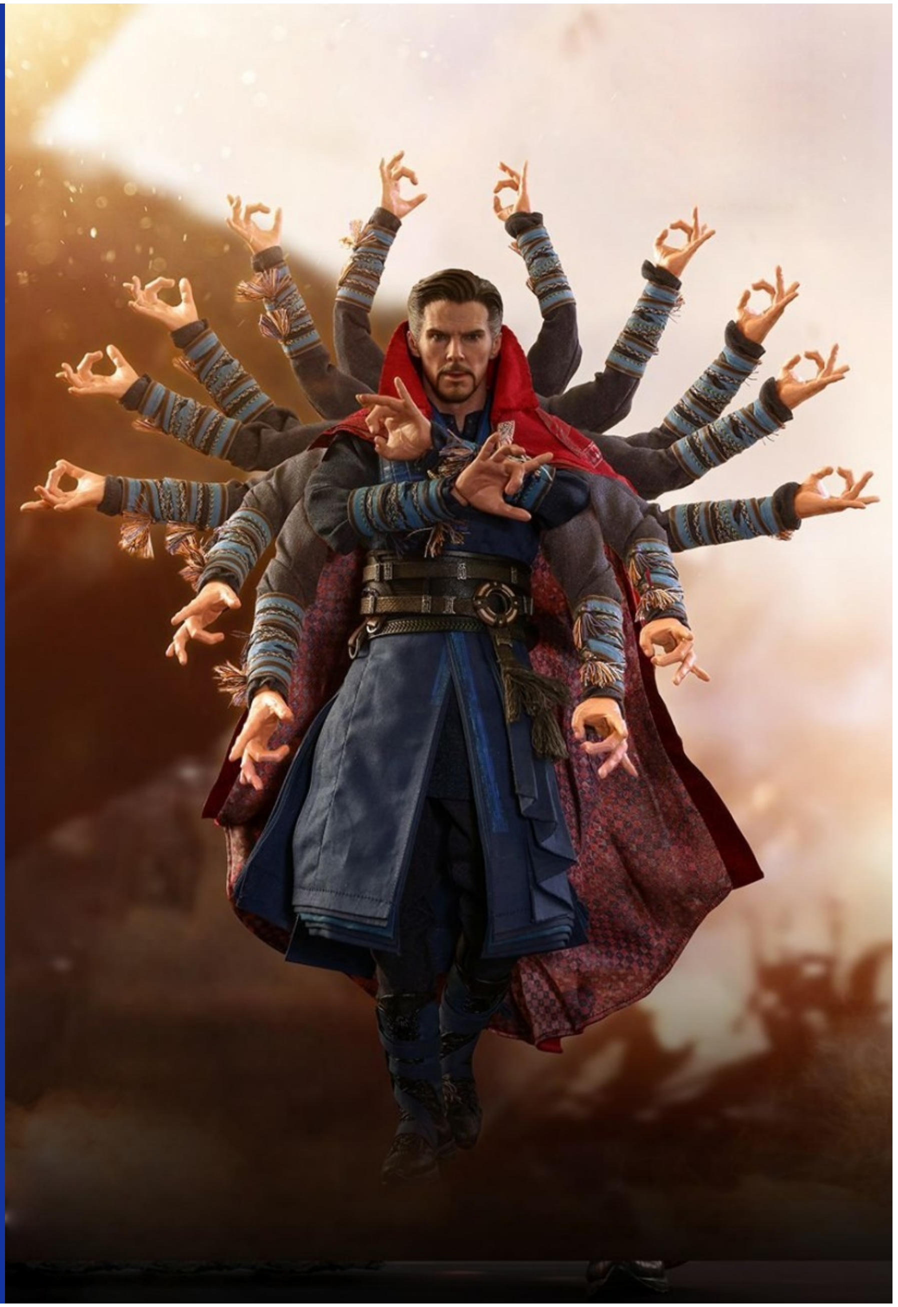
	ES2015	ES2016+
Node	98%	97%
Chrome	98%	97%
Firefox	98%	80%
Edge	96%	40%
Safari	99%	80%
IE	11%	0%

BABEL

TypeScript

02

CLASSES



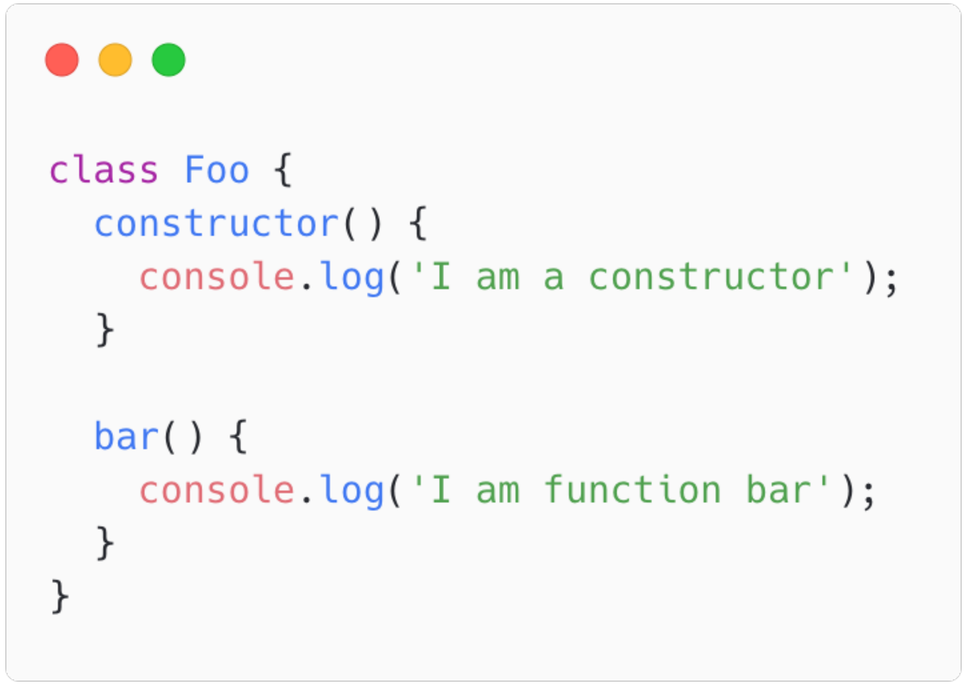
CLASSES

Syntactical sugar over prototype-based inheritance

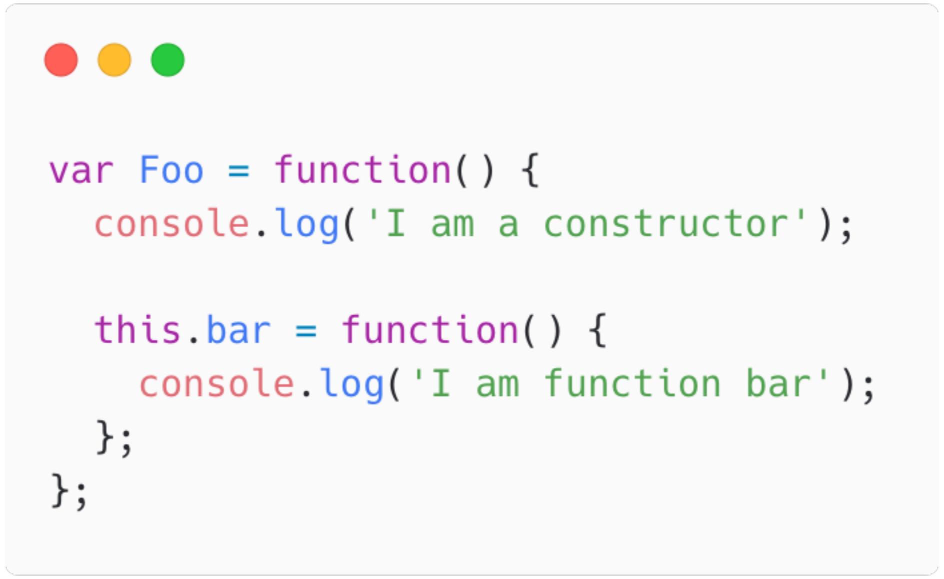
NOT a new object-oriented inheritance model

Constructor functions with a prototype property

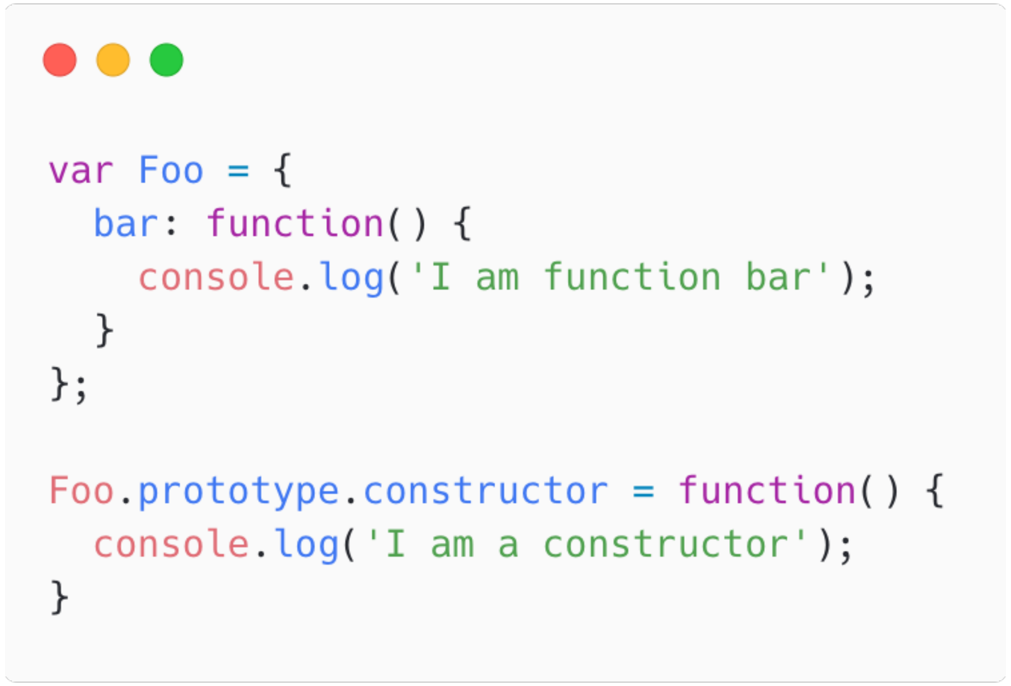
CLASSES


```
class Foo {
  constructor() {
    console.log('I am a constructor');
  }

  bar() {
    console.log('I am function bar');
  }
}
```

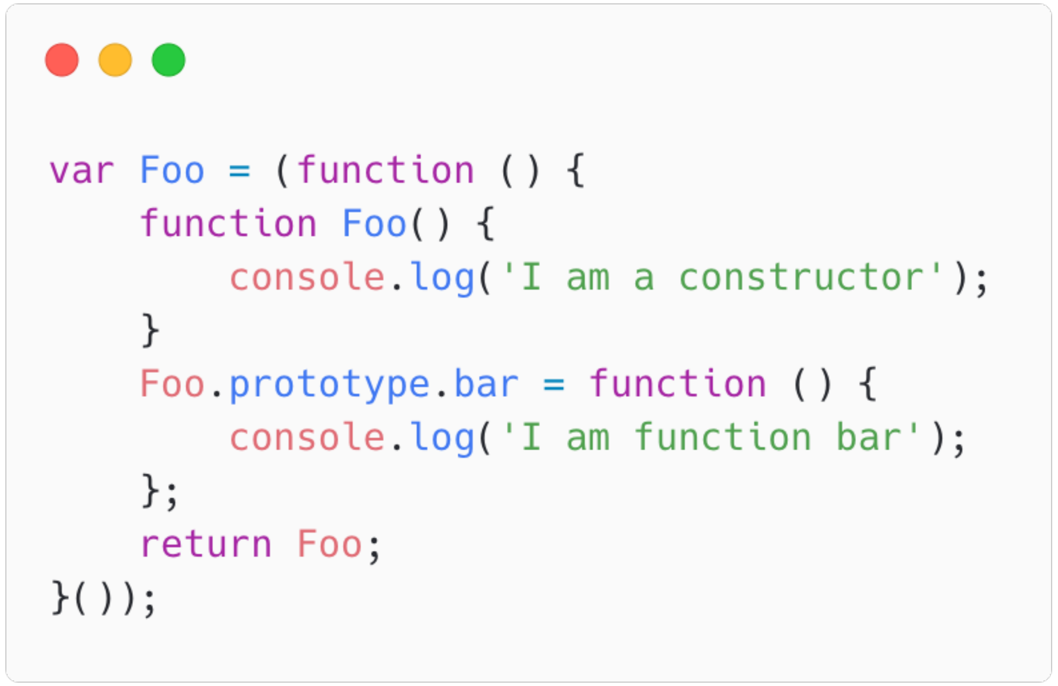
CLASSES



```
var Foo = function() {
  console.log('I am a constructor');

  this.bar = function() {
    console.log('I am function bar');
  };
};
```

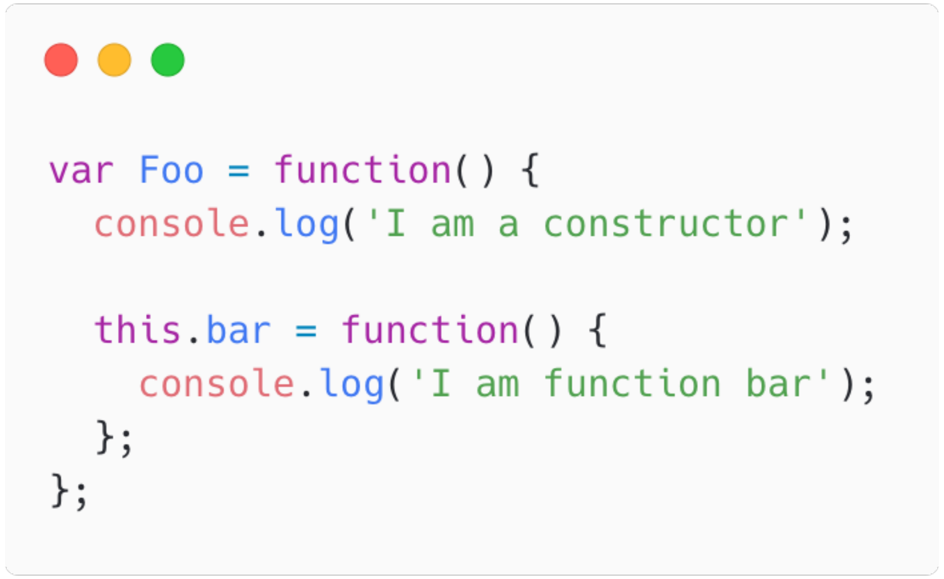


```
var Foo = {
  bar: function() {
    console.log('I am function bar');
  }
};

Foo.prototype.constructor = function() {
  console.log('I am a constructor');
}
```

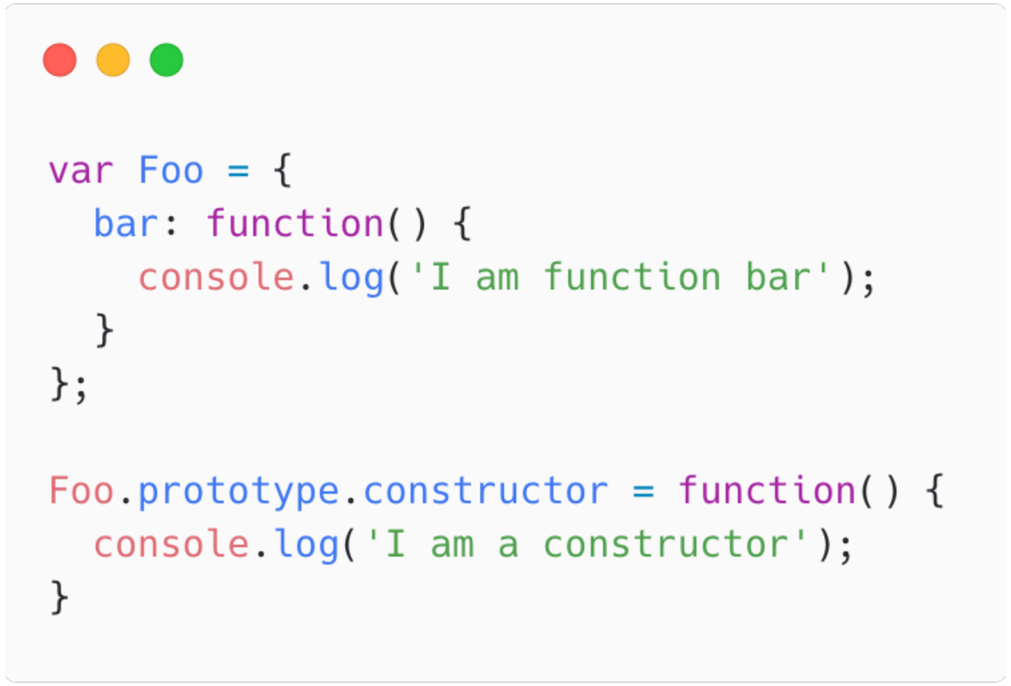


```
var Foo = (function () {
    function Foo() {
        console.log('I am a constructor');
    }
    Foo.prototype.bar = function () {
        console.log('I am function bar');
    };
    return Foo;
}());
```

CLASSES

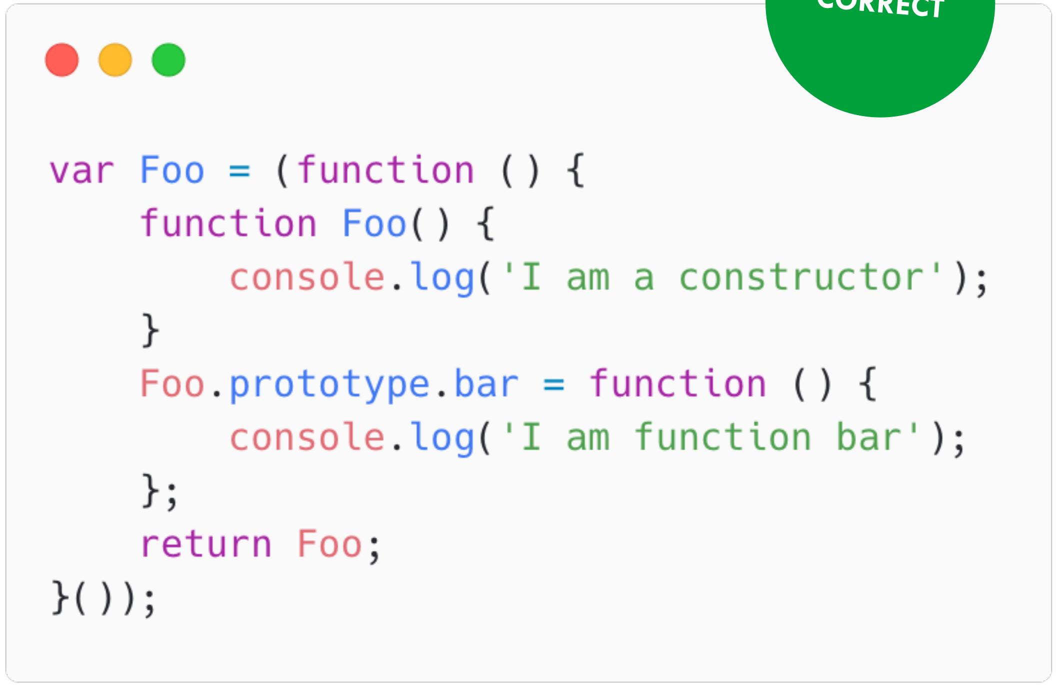


```
var Foo = function() {
  console.log('I am a constructor');

  this.bar = function() {
    console.log('I am function bar');
  };
};
```



```
var Foo = {
  bar: function() {
    console.log('I am function bar');
  }
};

Foo.prototype.constructor = function() {
  console.log('I am a constructor');
}
```

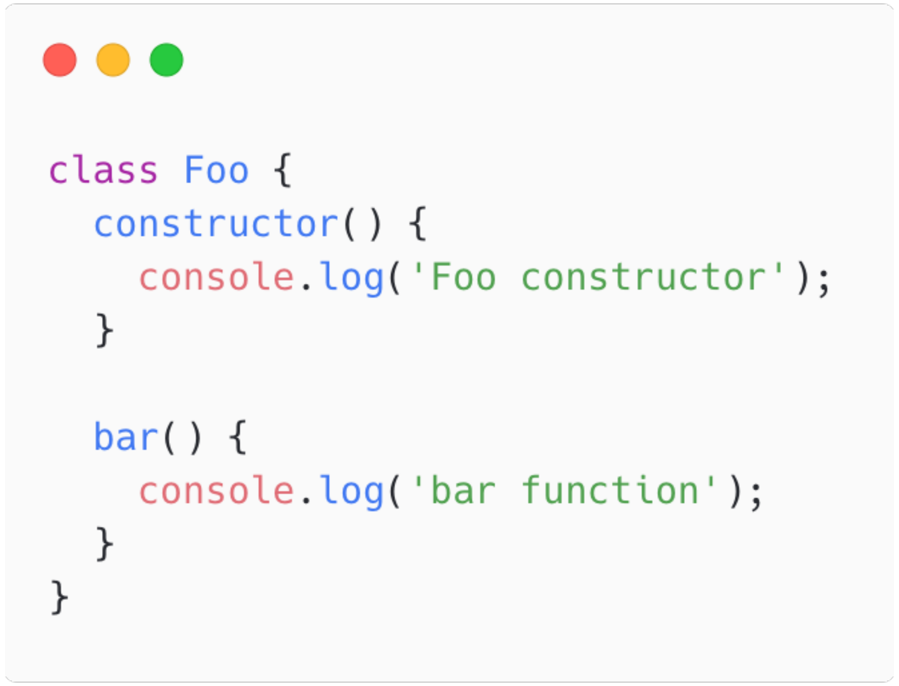
CORRECT

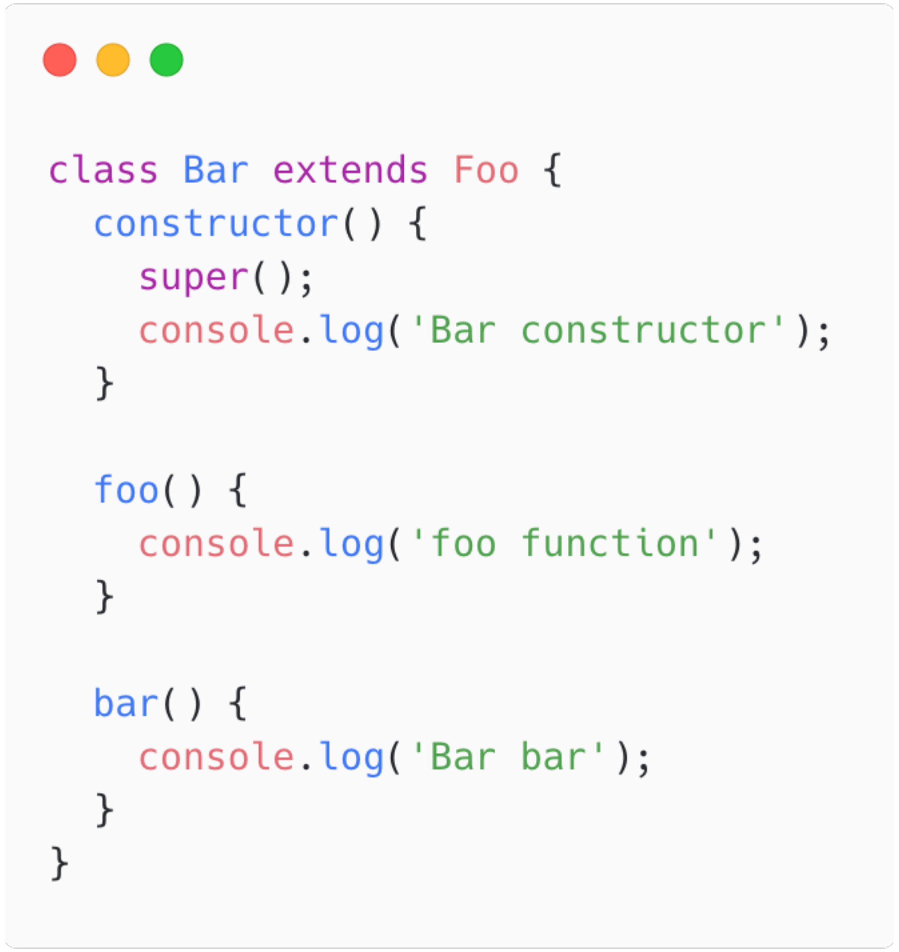


```
var Foo = (function () {
    function Foo() {
        console.log('I am a constructor');
    }
    Foo.prototype.bar = function () {
        console.log('I am function bar');
    };
    return Foo;
}());
```

CLASSES

Extending


```
class Foo {
  constructor() {
    console.log('Foo constructor');
  }

  bar() {
    console.log('bar function');
  }
}
```


```
class Bar extends Foo {
  constructor() {
    super();
    console.log('Bar constructor');
  }

  foo() {
    console.log('foo function');
  }

  bar() {
    console.log('Bar bar');
  }
}
```

CLASSES

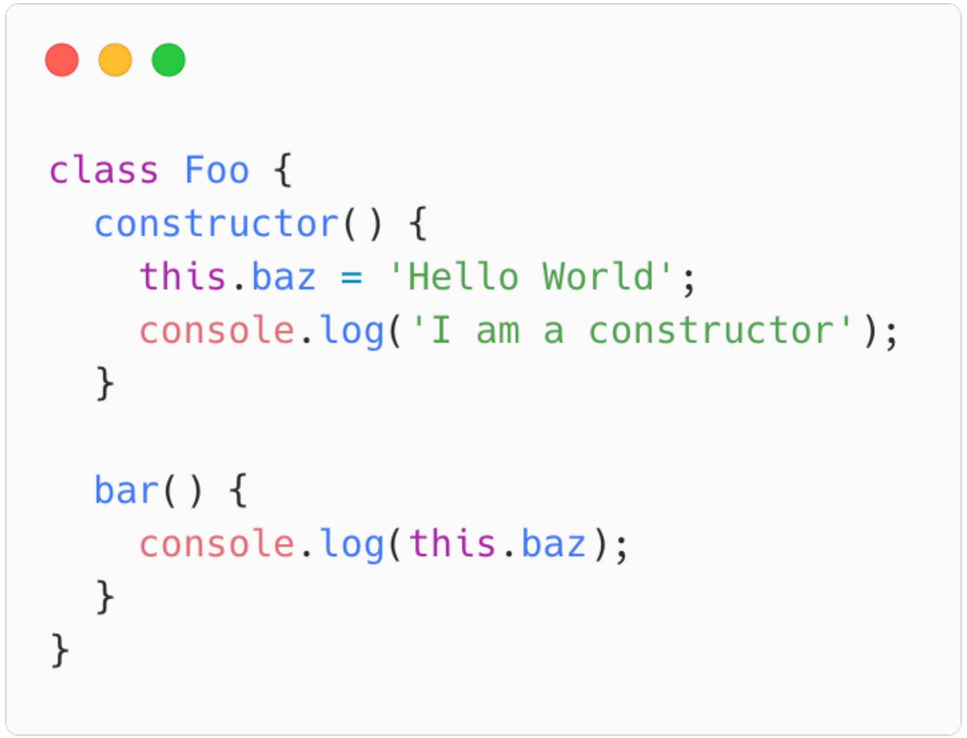
Multiple class inheritance not allowed

Extending requires to call the parent’s constructor (super()) in the constructor function

super() needs to be called before calling “this” in the constructor function

CLASSES

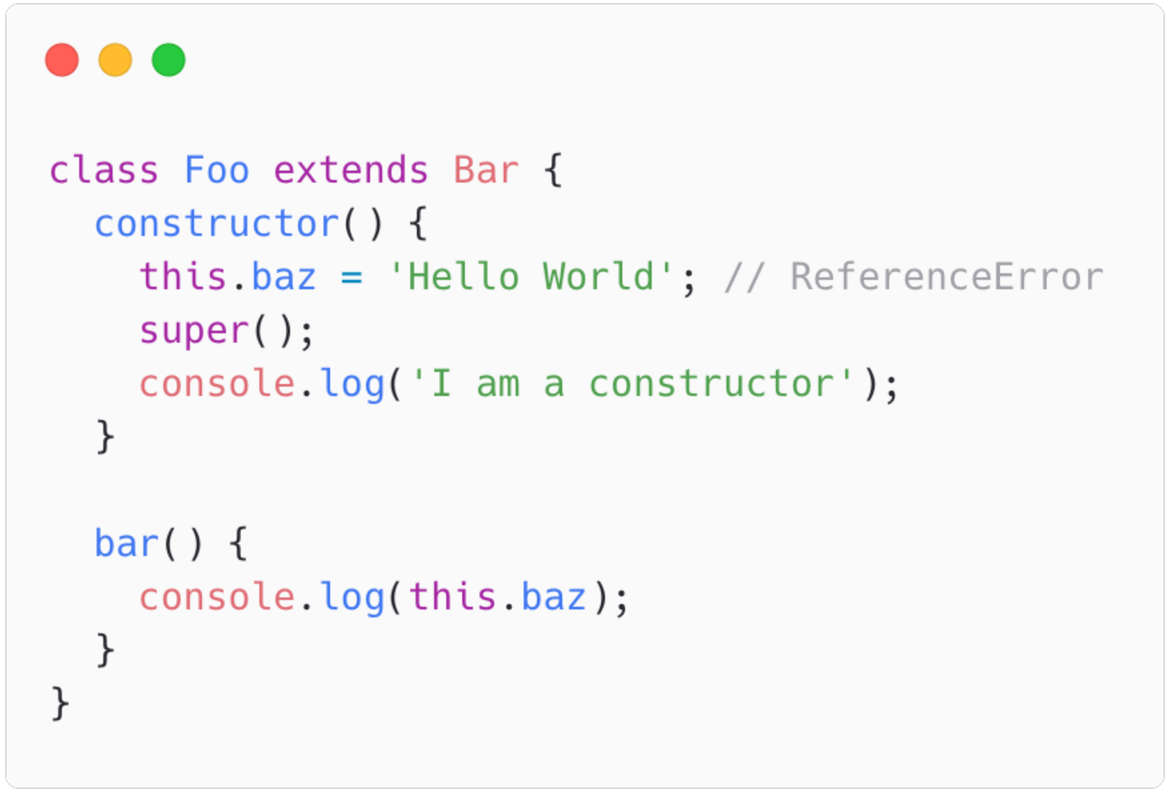
Variables



```
class Foo {
  constructor() {
    this.baz = 'Hello World';
    console.log('I am a constructor');
  }

  bar() {
    console.log(this.baz);
  }
}
```

CLASSES

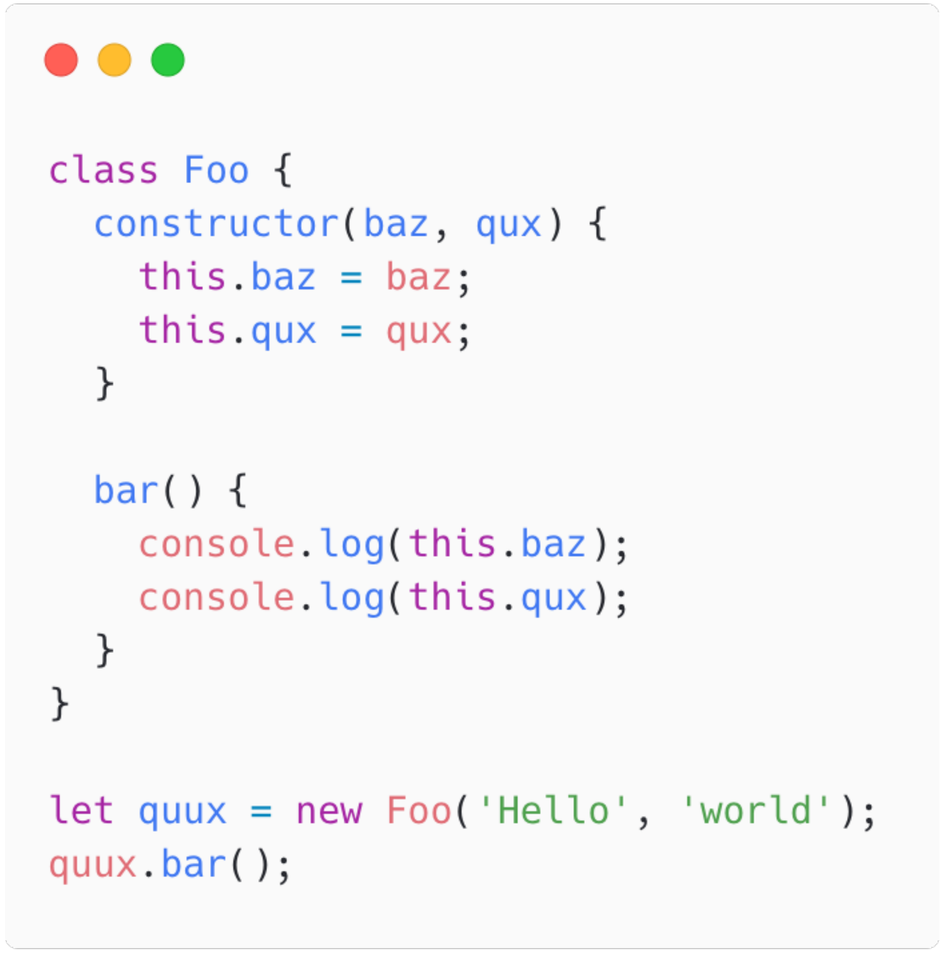
this before super()



```
class Foo extends Bar {
  constructor() {
    this.baz = 'Hello World'; // ReferenceError
    super();
    console.log('I am a constructor');
  }

  bar() {
    console.log(this.baz);
  }
}
```

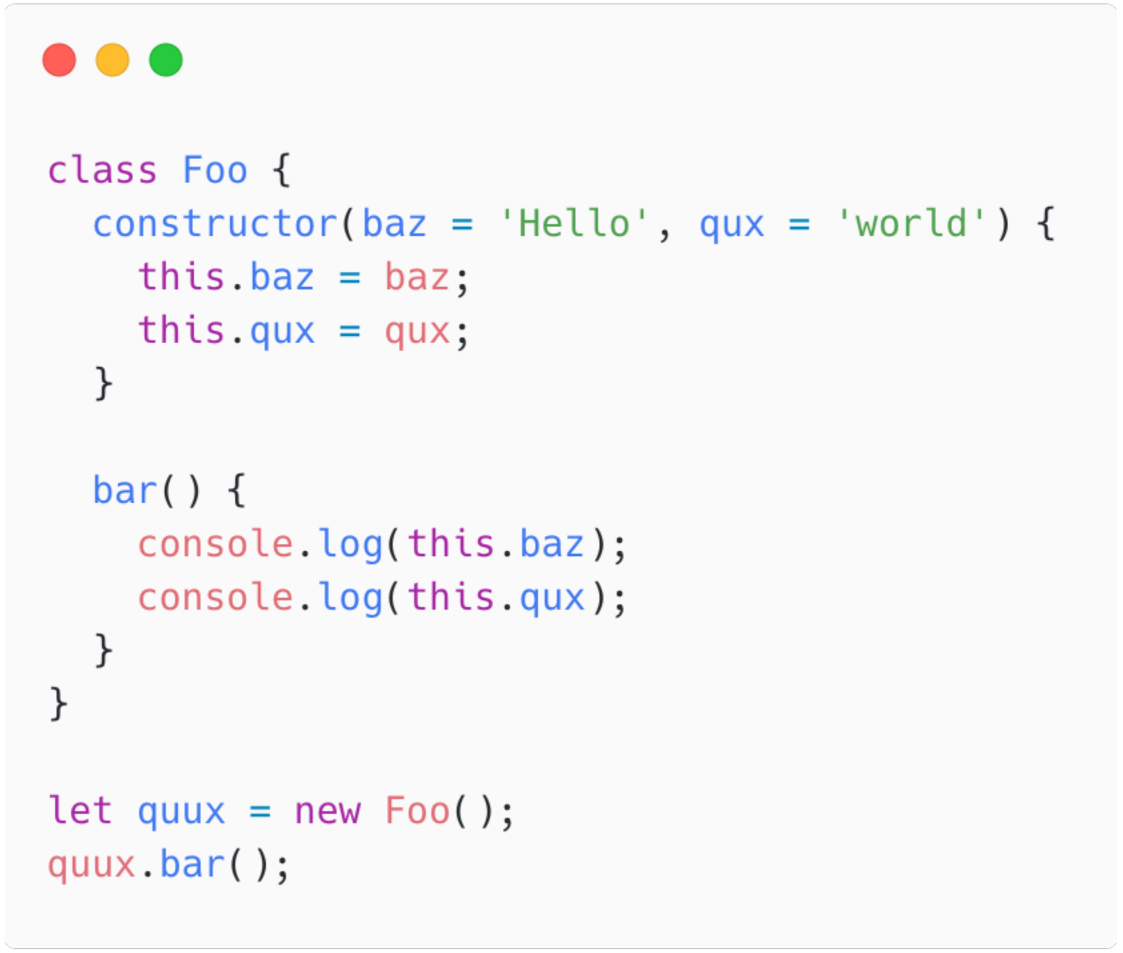
CLASSES

Constructor parameters

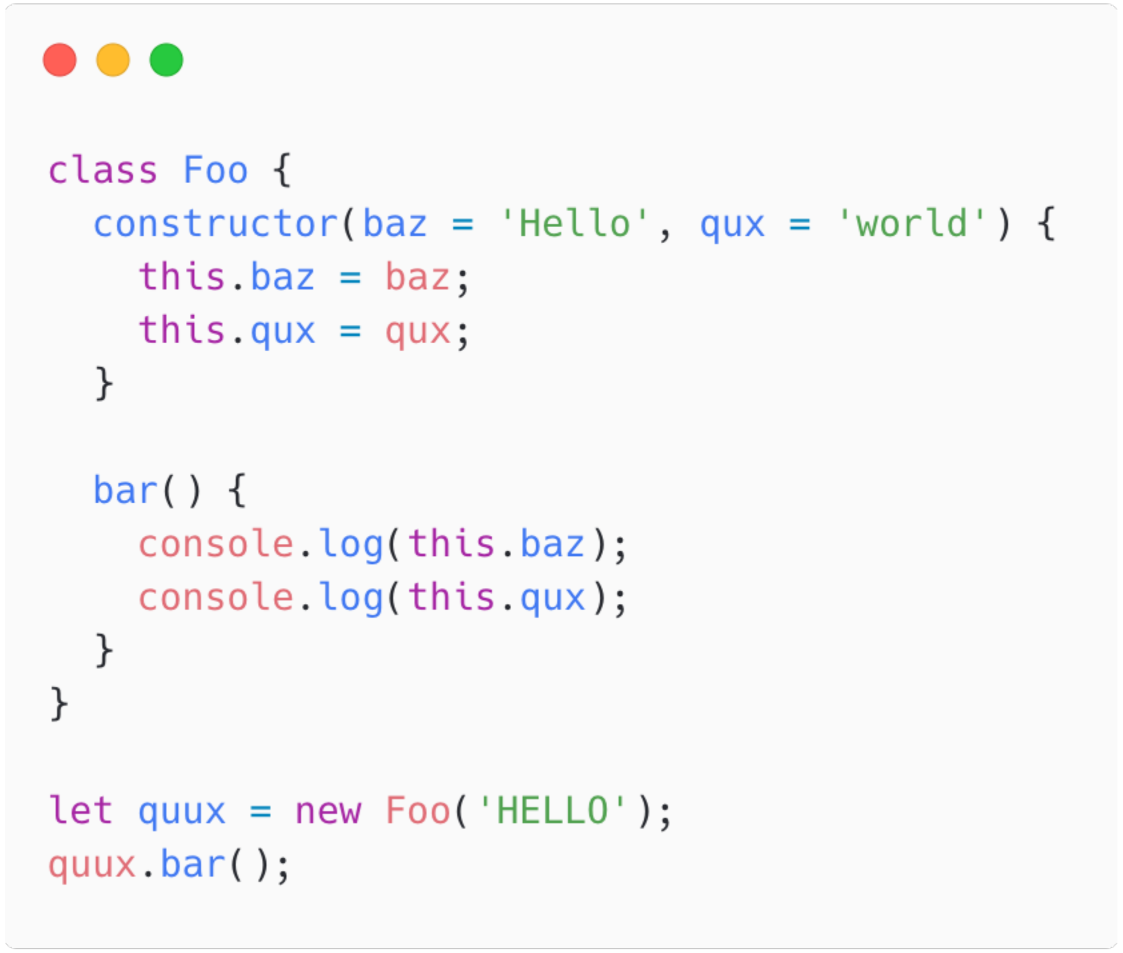


```
class Foo {
  constructor(baz, qux) {
    this.baz = baz;
    this.qux = qux;
  }

  bar() {
    console.log(this.baz);
    console.log(this.qux);
  }
}

let quux = new Foo('Hello', 'world');
quux.bar();
```

CLASSES



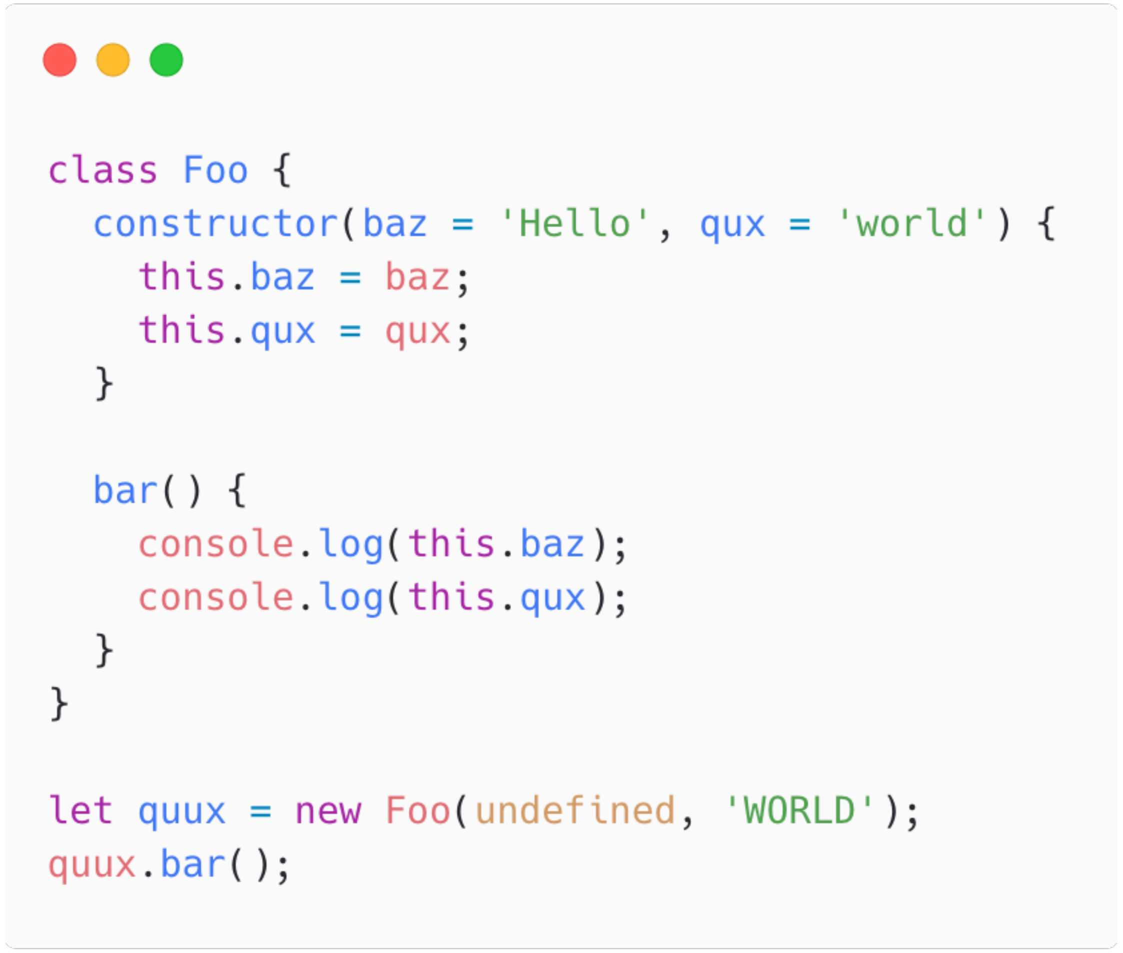
```
class Foo {
  constructor(baz = 'Hello', qux = 'world') {
    this.baz = baz;
    this.qux = qux;
  }

  bar() {
    console.log(this.baz);
    console.log(this.qux);
  }
}

let quux = new Foo();
quux.bar();
```



```
class Foo {
  constructor(baz = 'Hello', qux = 'world') {
    this.baz = baz;
    this.qux = qux;
  }

  bar() {
    console.log(this.baz);
    console.log(this.qux);
  }
}

let quux = new Foo('HELLO');
quux.bar();
```

CLASSES

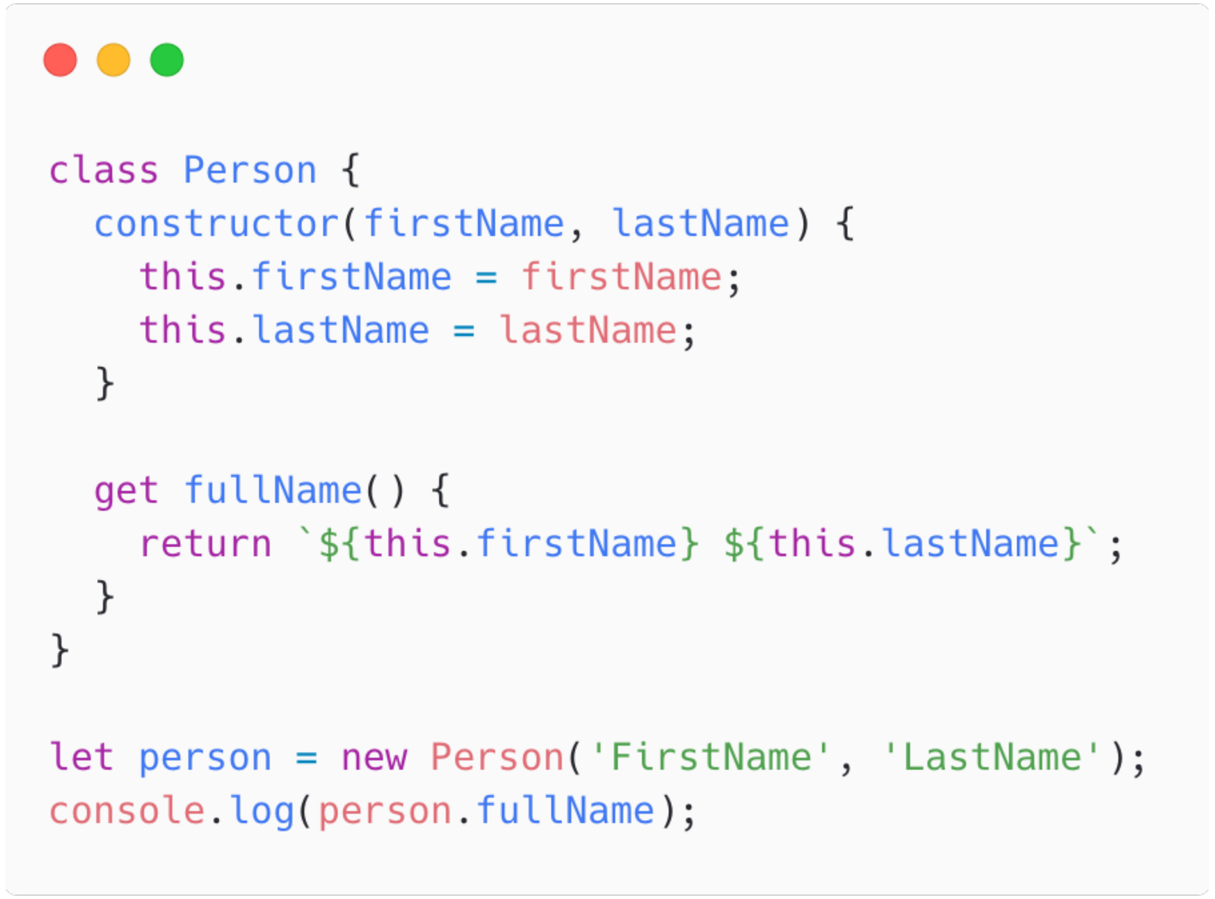
Skipping default parameters



```
class Foo {
  constructor(baz = 'Hello', qux = 'world') {
    this.baz = baz;
    this.qux = qux;
  }

  bar() {
    console.log(this.baz);
    console.log(this.qux);
  }
}

let quux = new Foo(undefined, 'WORLD');
quux.bar();
```

CLASSES

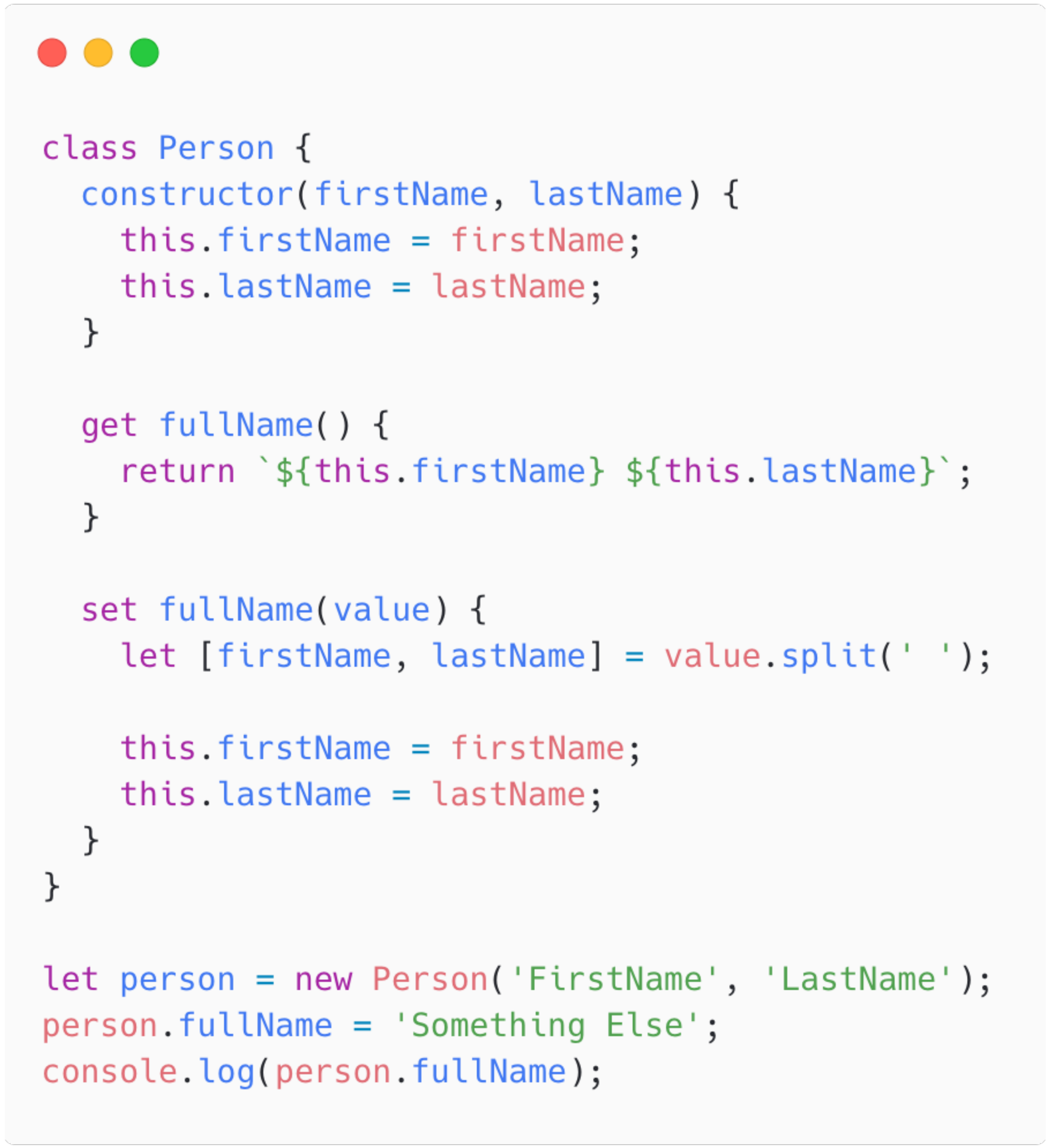
Getters



```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  }
}

let person = new Person('FirstName', 'LastName');
console.log(person.fullName);
```

CLASSES

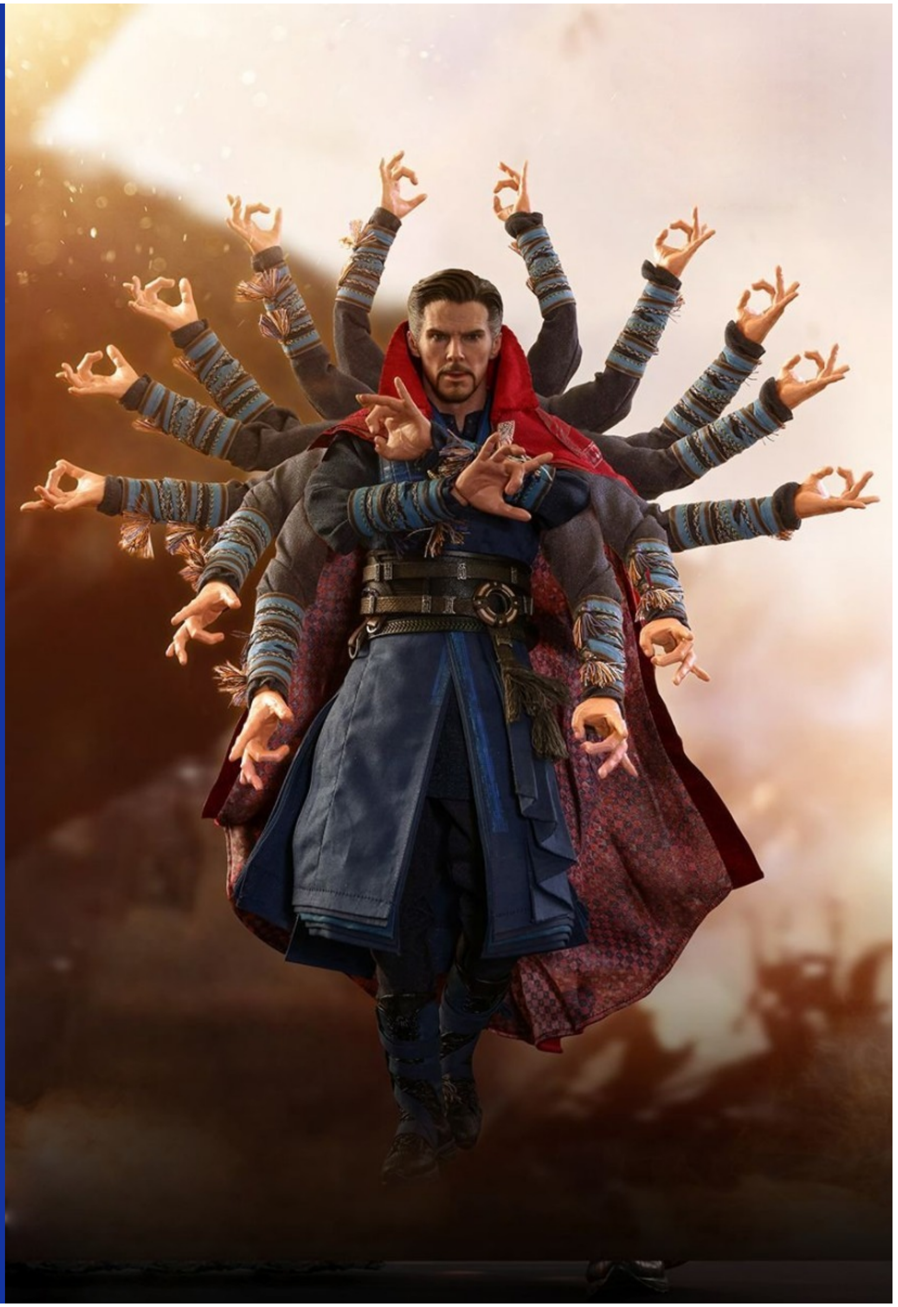
Setters



```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  }

  set fullName(value) {
    let [firstName, lastName] = value.split(' ');

    this.firstName = firstName;
    this.lastName = lastName;
  }
}

let person = new Person('FirstName', 'LastName');
person.fullName = 'Something Else';
console.log(person.fullName);
```

03

COLLECTIONS



COLLECTIONS

ES6 introduces two new data structures: Maps and Sets.

- **Map**
- **WeakMap**
- **Set**
- **WeakSet**

MAPS

The Map object is a simple key/value pair. Keys and values in a map may be primitive or objects.

Constructor:

```
new Map([iterable])
```

The parameter iterable represents any iterable object whose elements comprise of a key/value pair. Maps are ordered, i.e. they traverse the elements in the order of their insertion.

MAPS

- Map.prototype.set(key, value)
- Function sets the value for the key in the Map object. function takes two parameters namely, the key and its value. This function returns the Map object.
- Map.prototype.get(key)
- Function is used to retrieve the value corresponding to the specified key.
- Map.prototype.has(key)
- Function returns a boolean value indicating whether the specified key is found in the Map object. This function takes a key as parameter.
- Map.prototype.clear()
- Removes all key/value pairs from the Map object.

MAPS

- Map.prototype.delete(key)
- Removes any value associated to the key and returns the value that Map.prototype.has(key) would have previously returned.
- Map.prototype.entries()
- Returns a new Iterator object that contains an array of [key, value] for each element in the Map object in insertion order.
- Map.prototype.forEach(callbackFn[, thisArg])
- Calls **callbackFn** once for each key-value pair present in the Map object, in insertion order. If a thisArg parameter is provided to forEach, it will be used as the 'this' value for each callback .
- Map.prototype.keys()
- Returns a new Iterator object that contains the **keys** for each element in the Map object in insertion order.
- Map.prototype.values()
- Returns a new Iterator object that contains **an array of** [key, value] for each element in the Map object in insertion order.

WEAKMAPS

A weak map is identical to a map with the following exceptions:

- Its keys must be objects.
- Keys in a weak map can be Garbage collected. **Garbage collection** is a process of clearing the memory occupied by unreferenced objects in a program.
- A weak map cannot be iterated or cleared.

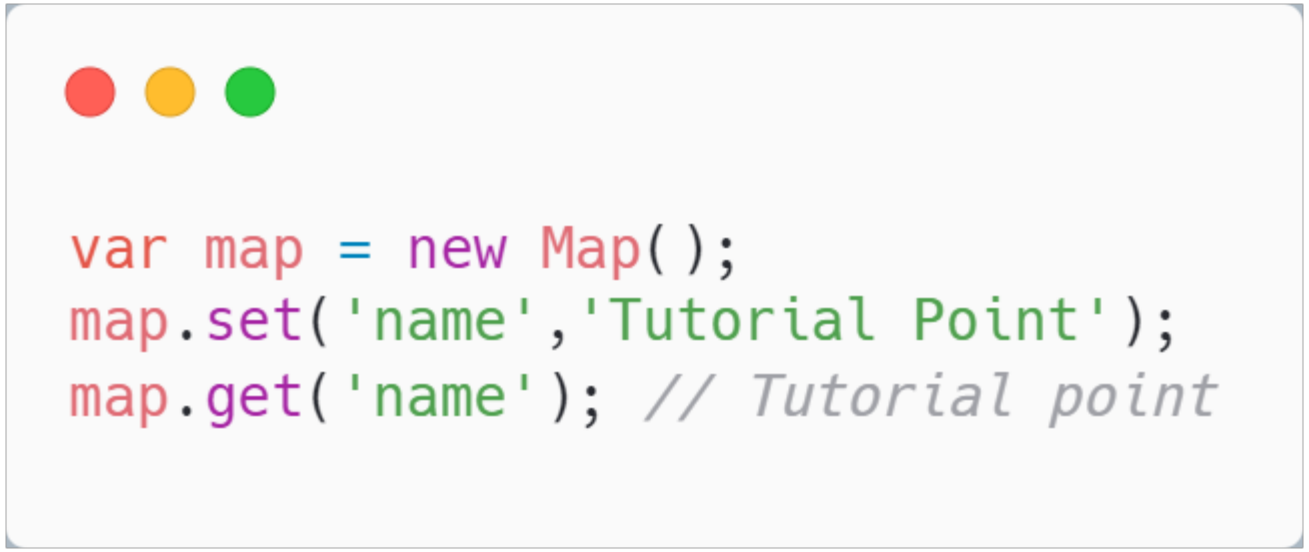
Constructor:

```
new WeakMap([iterable])
```

WEAKMAPS

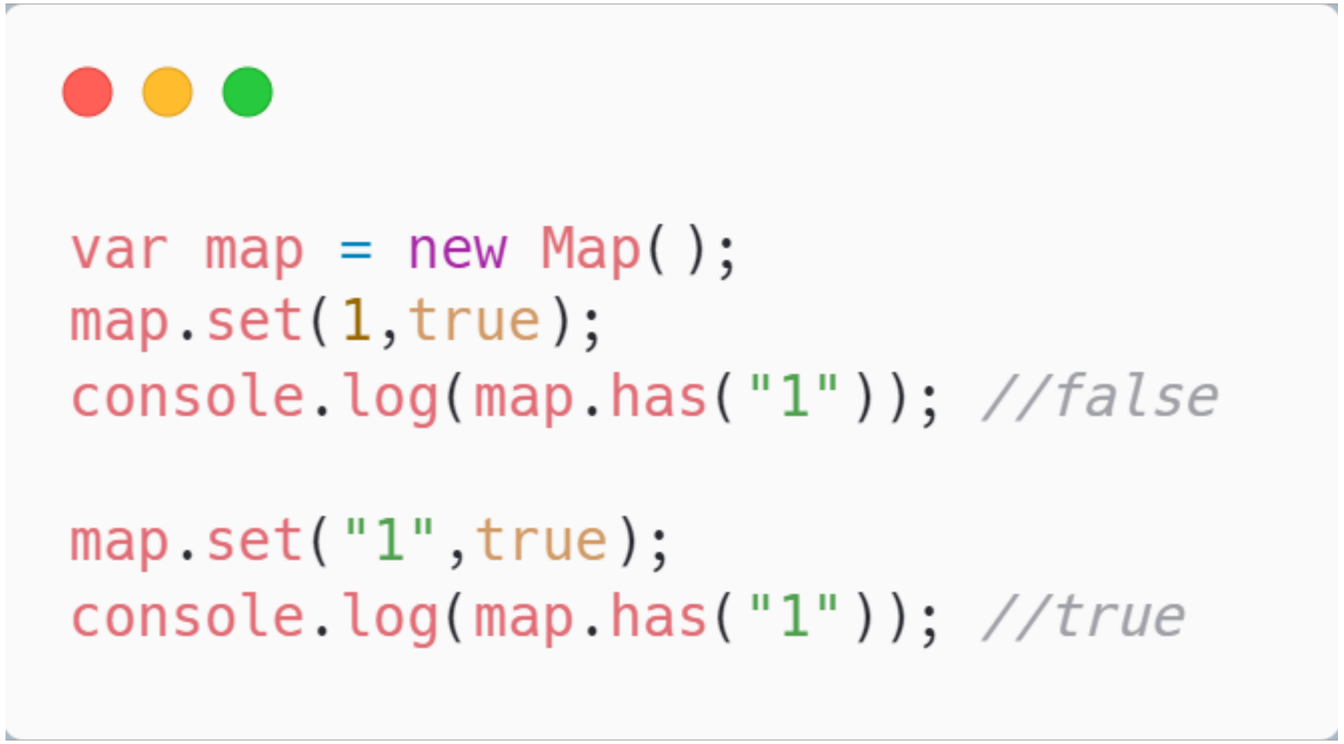
- WeakMap.prototype.delete(key)
- Removes any value associated to the key. WeakMap.prototype.has(key) will return false afterwards.
- WeakMap.prototype.get(key)
- Returns the value associated to the key, or undefined if there is none.
- WeakMap.prototype.has(key)
- Returns a Boolean asserting whether a value has been associated to the key in the WeakMap object or not.
- WeakMap.prototype.set(key, value)
- Sets the value for the key in the WeakMap object. Returns the WeakMap object.

MAPS



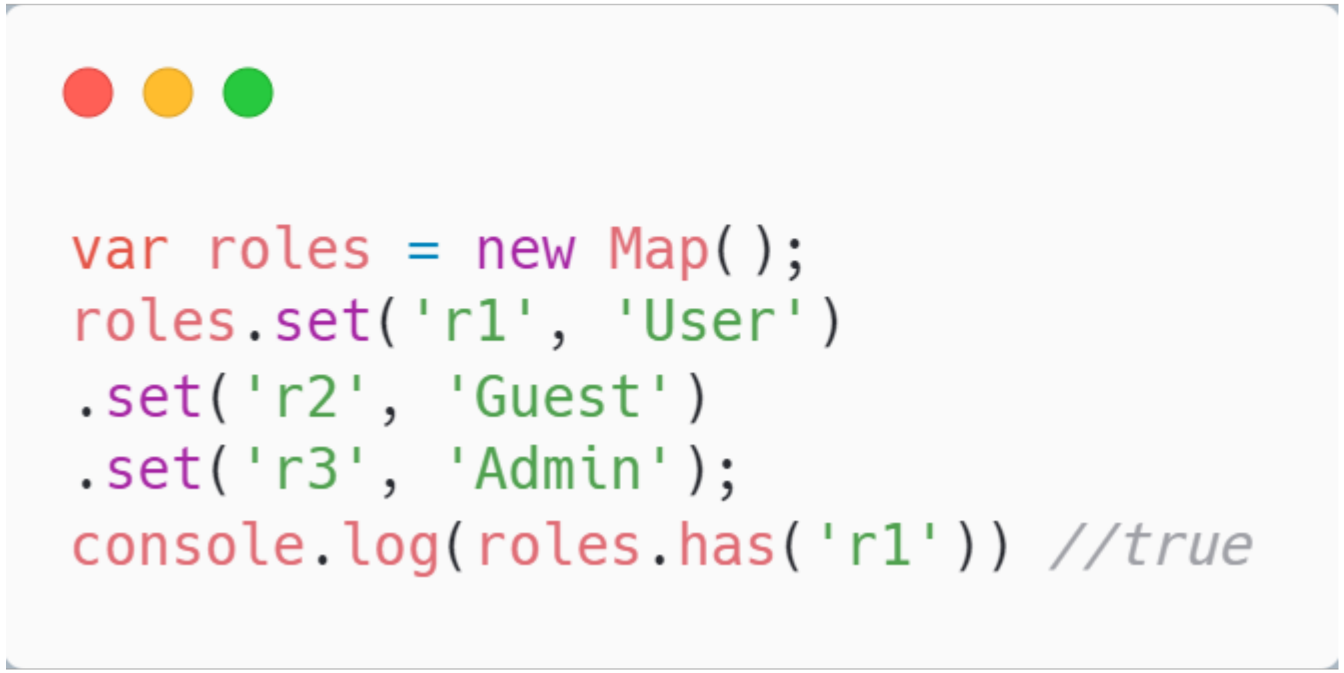
```
var map = new Map();
map.set('name','Tutorial Point');
map.get('name'); // Tutorial point
```

MAPS



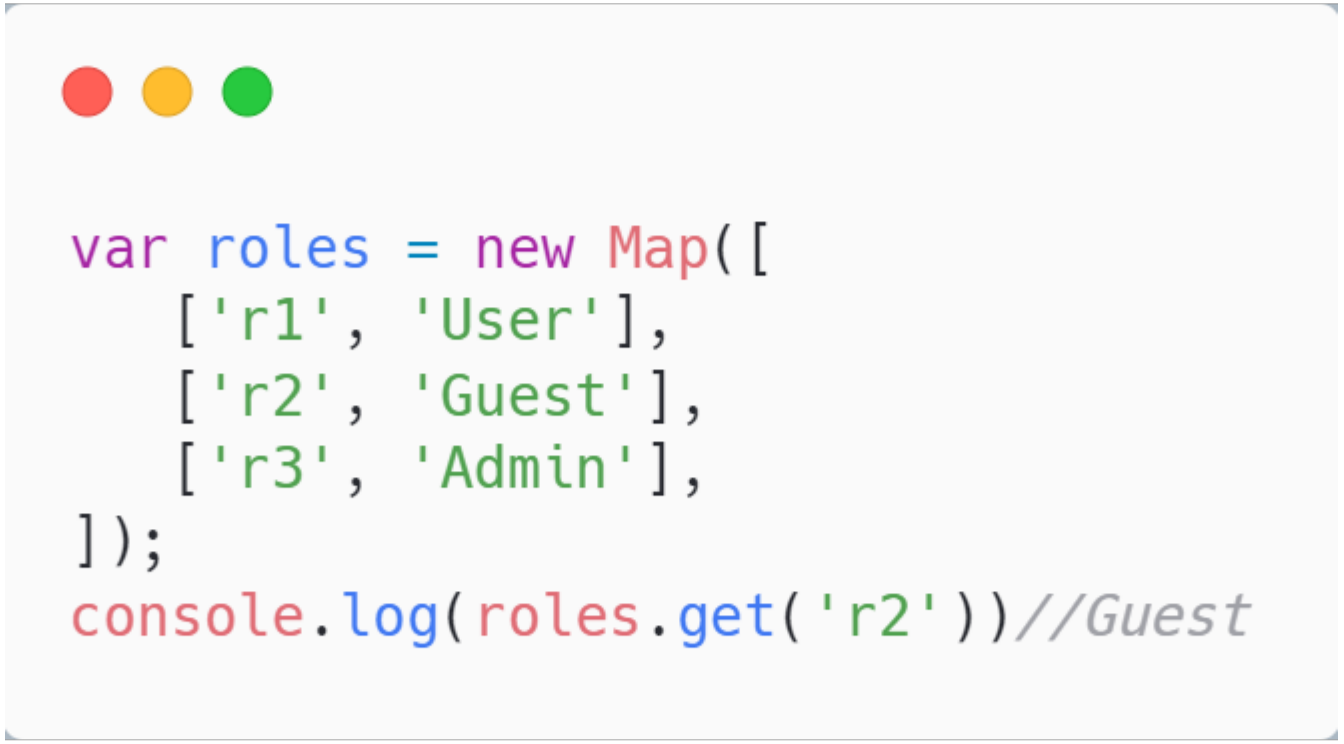
```
var map = new Map();
map.set(1,true);
console.log(map.has("1")); //false

map.set("1",true);
console.log(map.has("1")); //true
```

MAPS

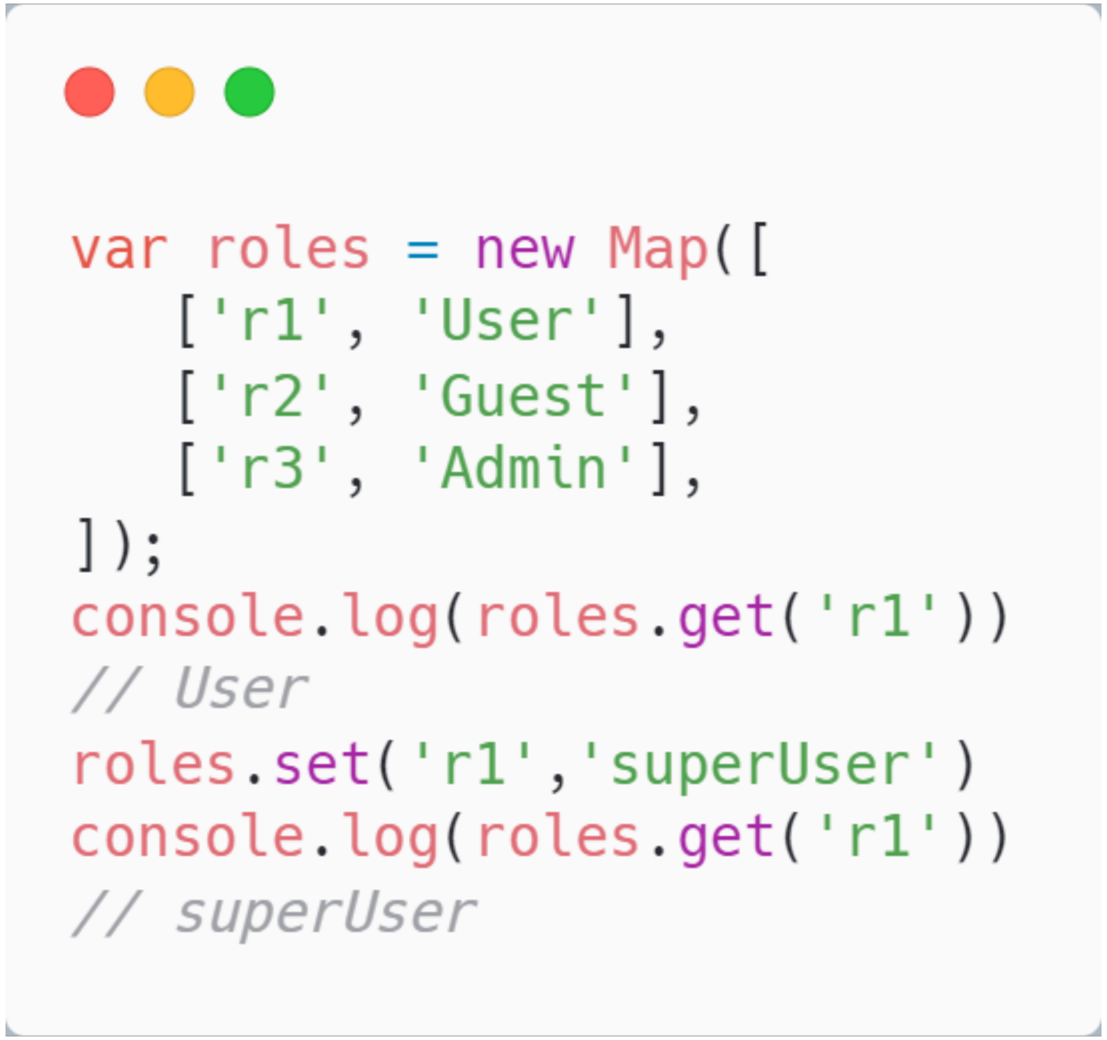


```
var roles = new Map();
roles.set('r1', 'User')
.set('r2', 'Guest')
.set('r3', 'Admin');
console.log(roles.has('r1')) //true
```

MAPS

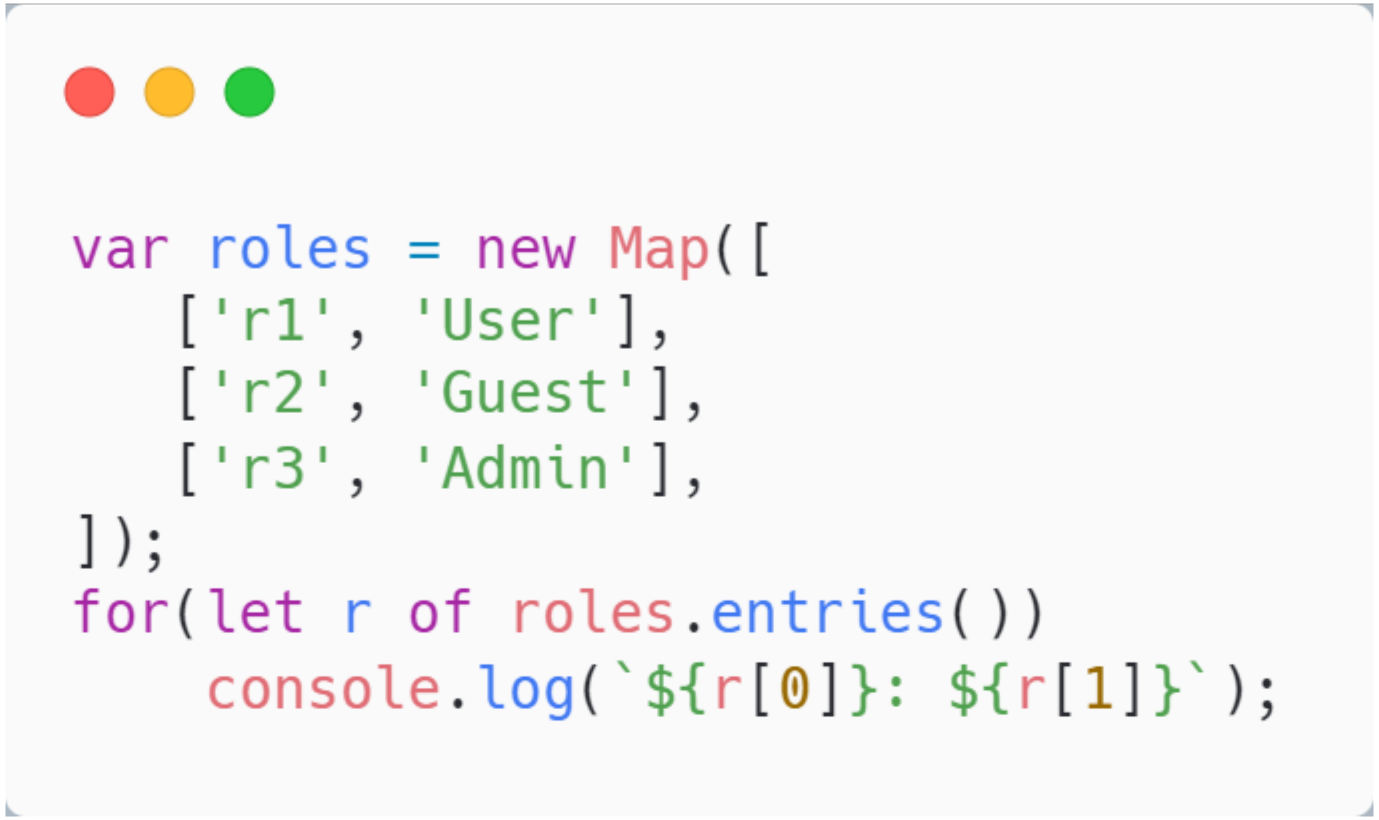

```
var roles = new Map([
    ['r1', 'User'],
    ['r2', 'Guest'],
    ['r3', 'Admin'],
]);
console.log(roles.get('r2'))//Guest
```

MAPS



```
var roles = new Map([
   ['r1', 'User'],
   ['r2', 'Guest'],
   ['r3', 'Admin'],
]);
console.log(roles.get('r1'))
// User
roles.set('r1','superUser')
console.log(roles.get('r1'))
// superUser
```

MAPS



```
var roles = new Map([
   ['r1', 'User'],
   ['r2', 'Guest'],
   ['r3', 'Admin'],
]);
for(let r of roles.entries())
    console.log(`${r[0]}: ${r[1]}`);
```

```
new Set([iterable]);
```

SETS

A set is an ES6 data structure. It is similar to an array with an exception that it cannot contain duplicates. In other words, it lets you store unique values.
Sets support both primitive values and object references.
Just like maps, sets are also ordered, i.e. elements are iterated in their insertion order. A set can be initialized using the following syntax.

```
new Set([iterable])
```

SETS

- Set.prototype.add(value)
- Appends a new element with the given value to the Set object. Returns the Set object.
- Set.prototype.clear()
- Removes all elements from the Set object.
- Set.prototype.delete(value)
- Removes the element associated to the value and returns the value that Set.prototype.has(value) would have previously returned. Set.prototype.has(value) will return false afterwards.
- Set.prototype.entries()
- Returns a new Iterator object that contains an array of [value, value] for each element in the Set object, in insertion order. This is kept similar to the Map object, so that each entry has the same value for its key and value here.

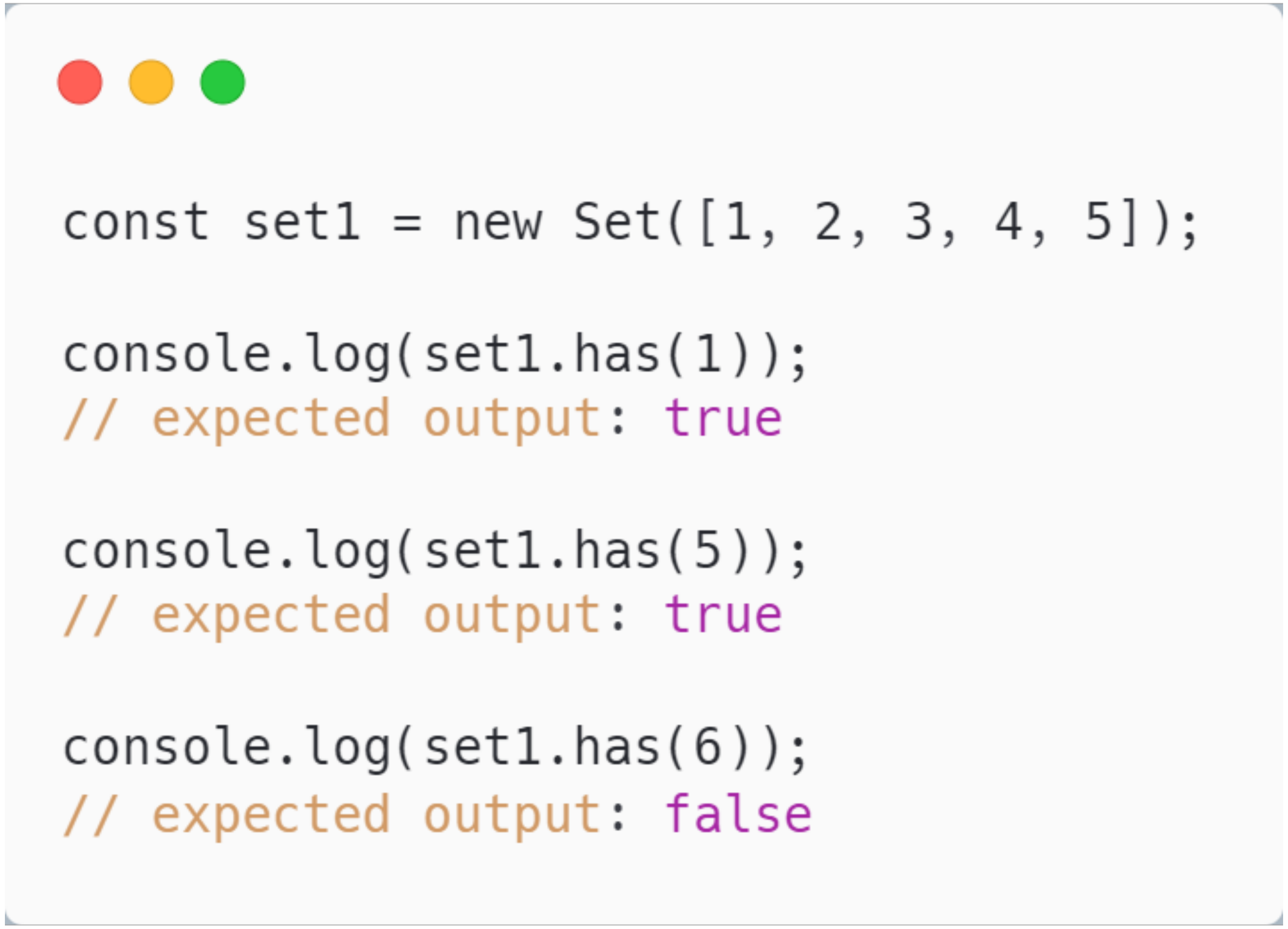
SETS

- Set.prototype.forEach(callbackFn[, thisArg])
- Calls callbackFn once for each value present in the Set object, in insertion order. If a thisArg parameter is provided to forEach, it will be used as the this value for each callback.
- Set.prototype.has(value)
- Returns a boolean asserting whether an element is present with the given value in the Set object or not.
- Set.prototype.keys()
- Is the same function as the values() function and returns a new Iterator object that contains the values for each element in the Set object in insertion order.
- Set.prototype.values()
- Returns a new Iterator object that contains the values for each element in the Set object in insertion order.
- Set.prototype[@@iterator]()
- Returns a new Iterator object that contains the values for each element in the Set object in insertion order.

WEAKSETS

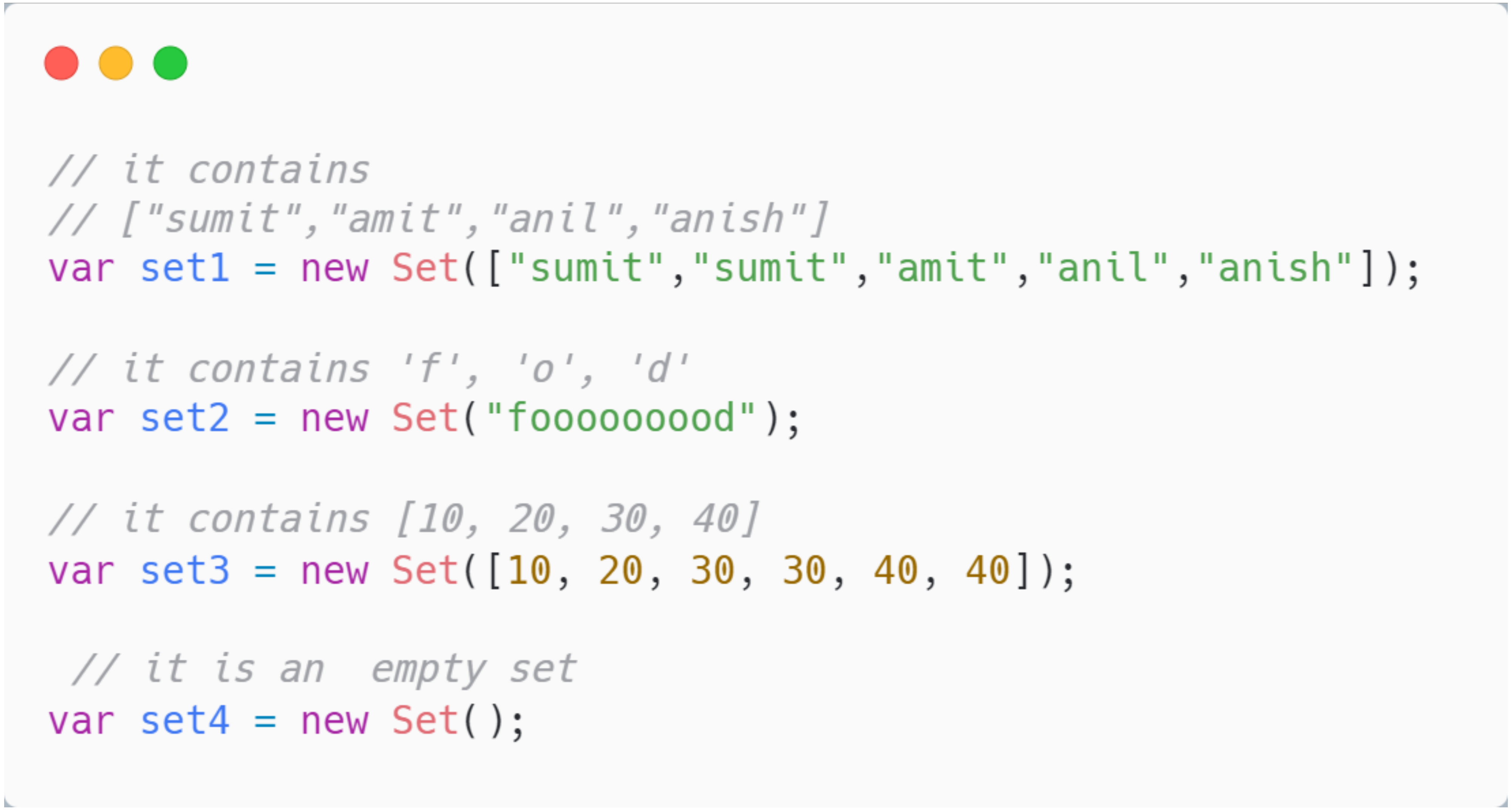
- WeakSet.prototype.add(value)
- Appends a new object with the given value to the WeakSet object.
- WeakSet.prototype.delete(value)
- Removes the element associated to the value. WeakSet.prototype.has(value) will return false afterwards.
- WeakSet.prototype.has(value)
- Returns a boolean asserting whether an element is present with the given value in the WeakSet object or not.

SETS



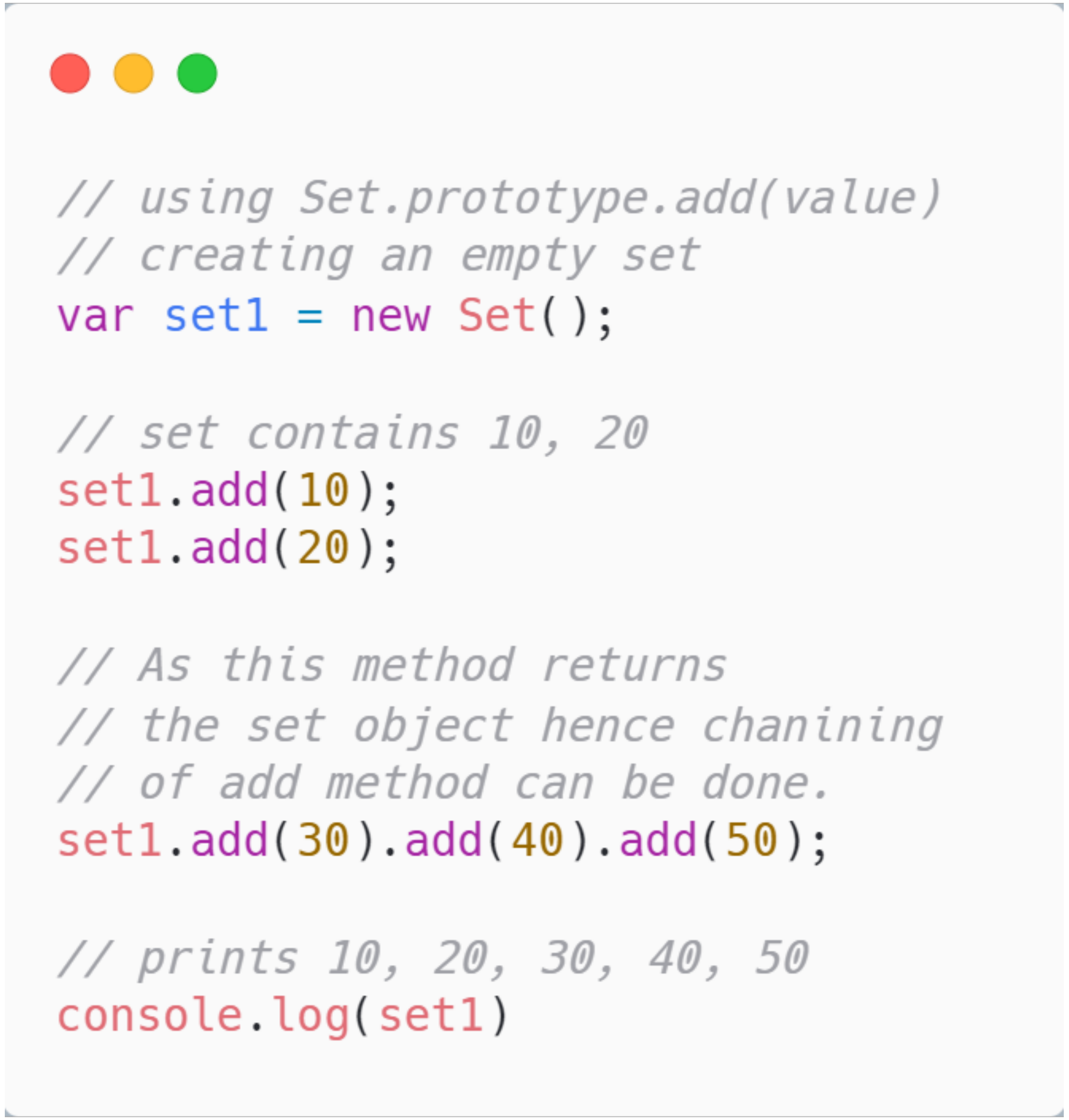
```
const set1 = new Set([1, 2, 3, 4, 5]);

console.log(set1.has(1));
// expected output: true

console.log(set1.has(5));
// expected output: true

console.log(set1.has(6));
// expected output: false
```

SETS



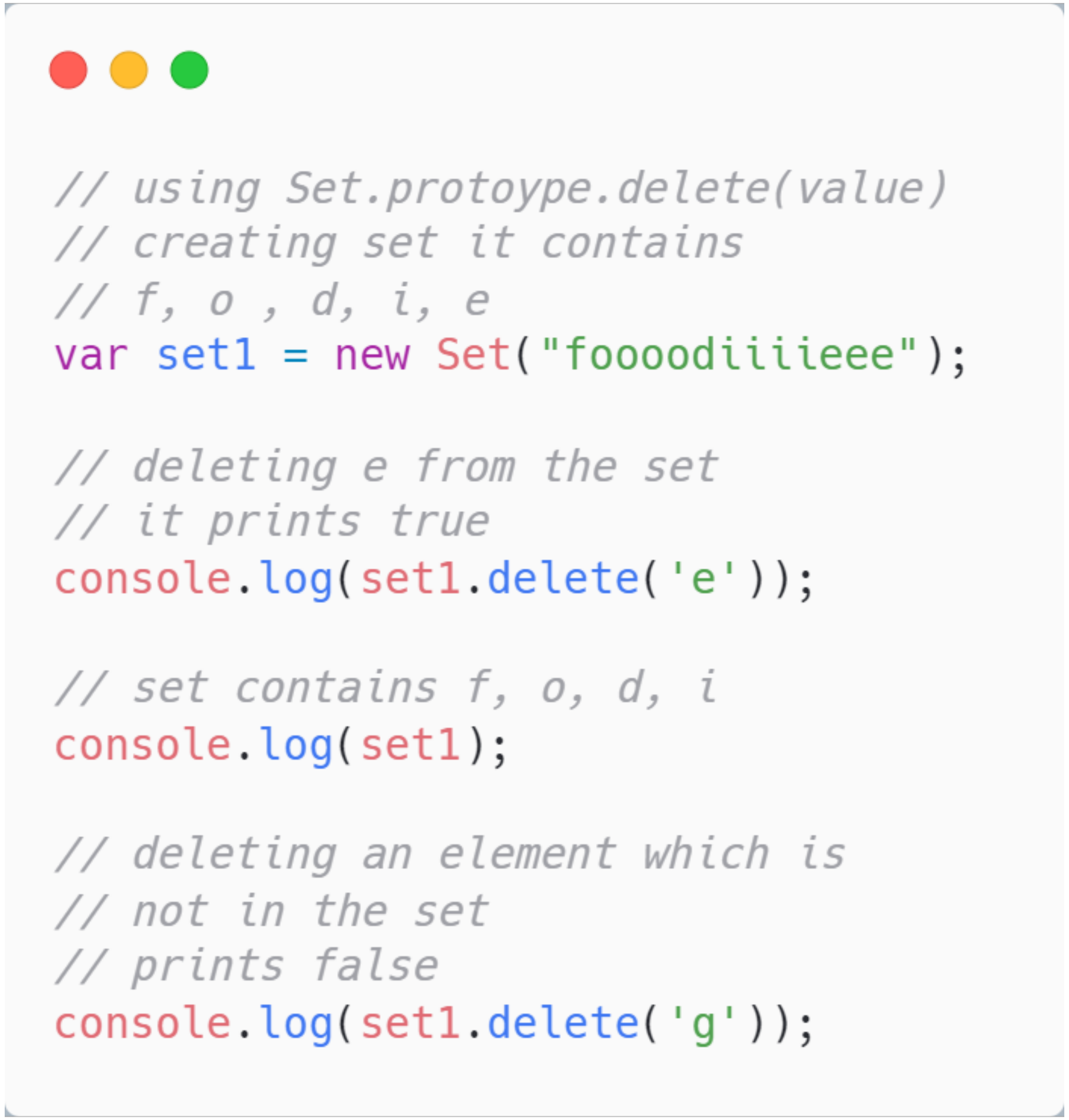
```
// it contains
// ["sumit","amit","anil","anish"]
var set1 = new Set(["sumit","sumit","amit","anil","anish"]);

// it contains 'f', 'o', 'd'
var set2 = new Set("fooooooood");

// it contains [10, 20, 30, 40]
var set3 = new Set([10, 20, 30, 30, 40, 40]);

 // it is an  empty set
var set4 = new Set();
```

SETS

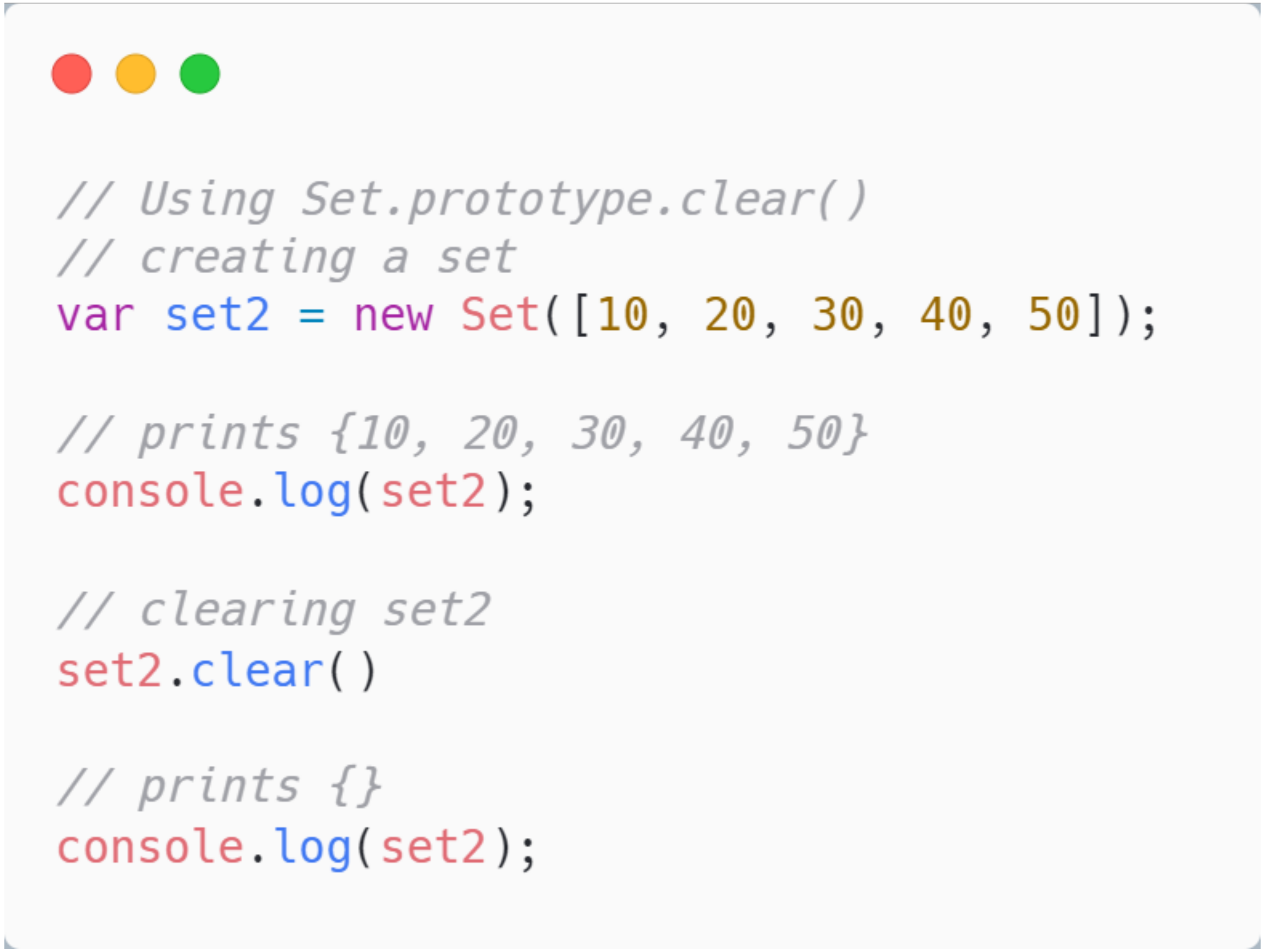


```
// using Set.prototype.add(value)
// creating an empty set
var set1 = new Set();

// set contains 10, 20
set1.add(10);
set1.add(20);

// As this method returns
// the set object hence chaning
// of add method can be done.
set1.add(30).add(40).add(50);

// prints 10, 20, 30, 40, 50
console.log(set1)
```

SETS

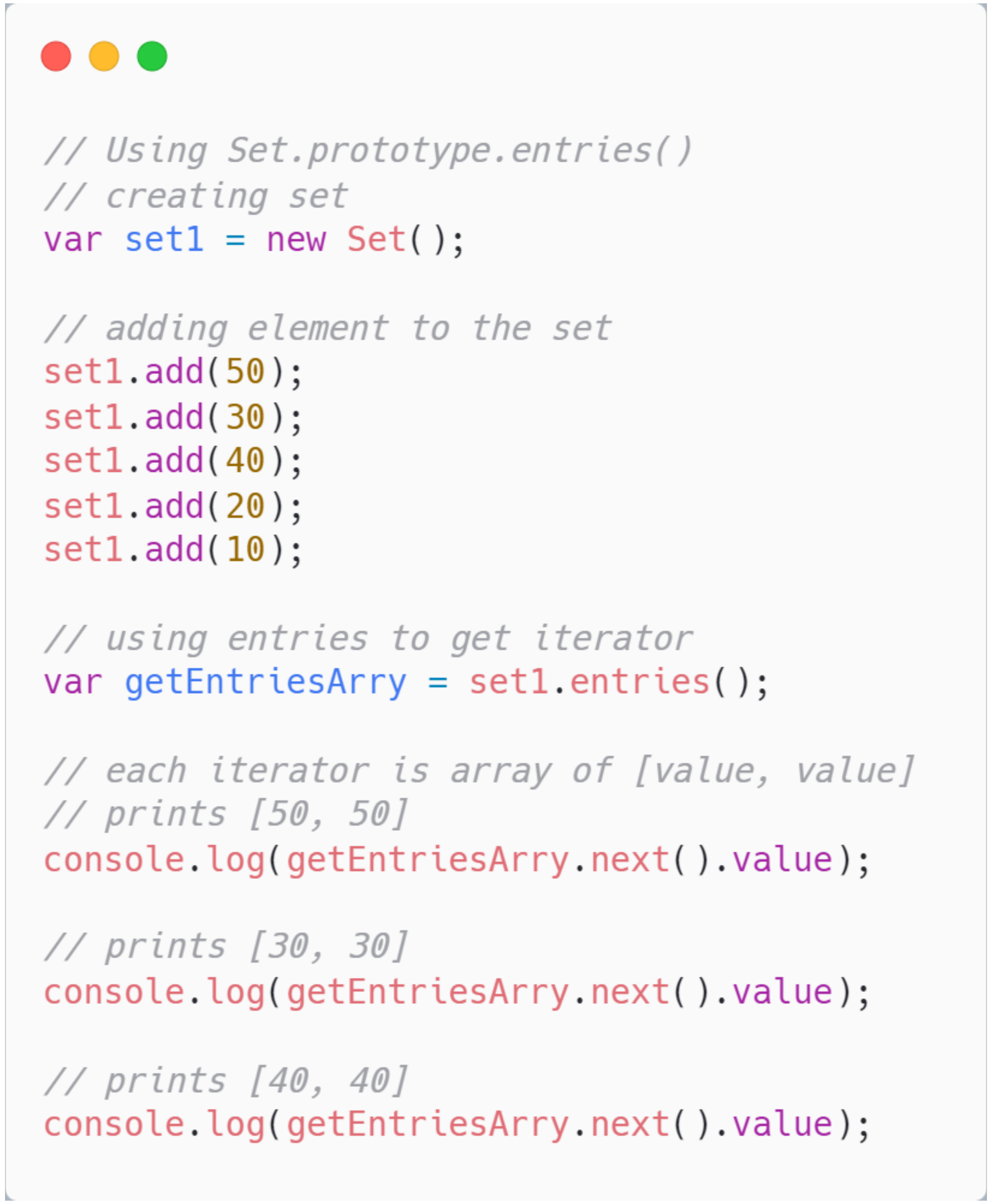

```
// using Set.protoype.delete(value)
// creating set it contains
// f, o , d, i, e
var set1 = new Set("foooodiiiieee");

// deleting e from the set
// it prints true
console.log(set1.delete('e'));

// set contains f, o, d, i
console.log(set1);

// deleting an element which is
// not in the set
// prints false
console.log(set1.delete('g'));
```

SETS



```
// Using Set.prototype.clear()
// creating a set
var set2 = new Set([10, 20, 30, 40, 50]);

// prints {10, 20, 30, 40, 50}
console.log(set2);

// clearing set2
set2.clear()

// prints {}
console.log(set2);
```

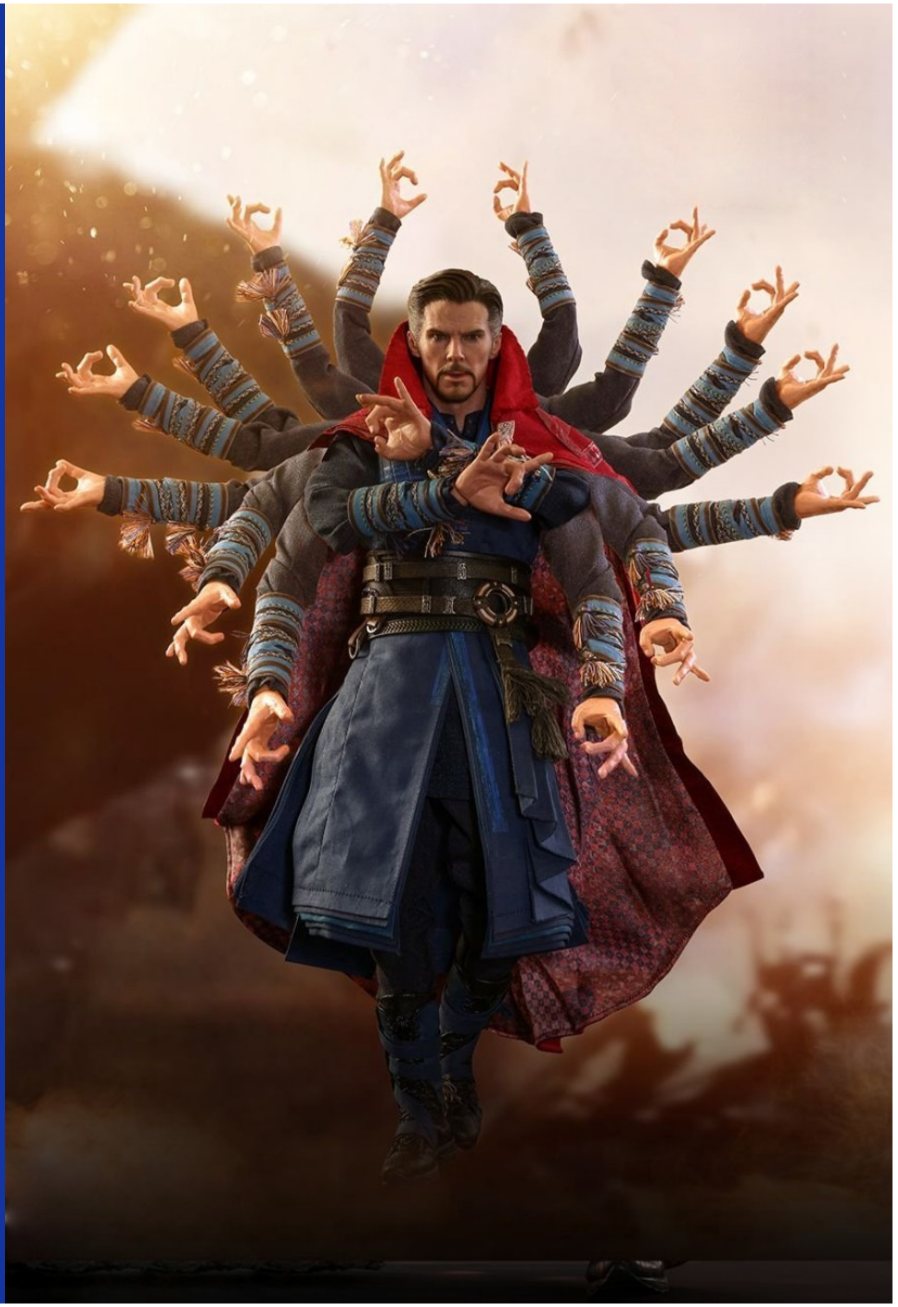
SETS



```
// Using Set.prototype.entries()
// creating set
var set1 = new Set();

// adding element to the set
set1.add(50);
set1.add(30);
set1.add(40);
set1.add(20);
set1.add(10);

// using entries to get iterator
var getEntriesArry = set1.entries();

// each iterator is array of [value, value]
// prints [50, 50]
console.log(getEntriesArry.next().value);

// prints [30, 30]
console.log(getEntriesArry.next().value);

// prints [40, 40]
console.log(getEntriesArry.next().value);
```

04

ARROW FUNCTIONS

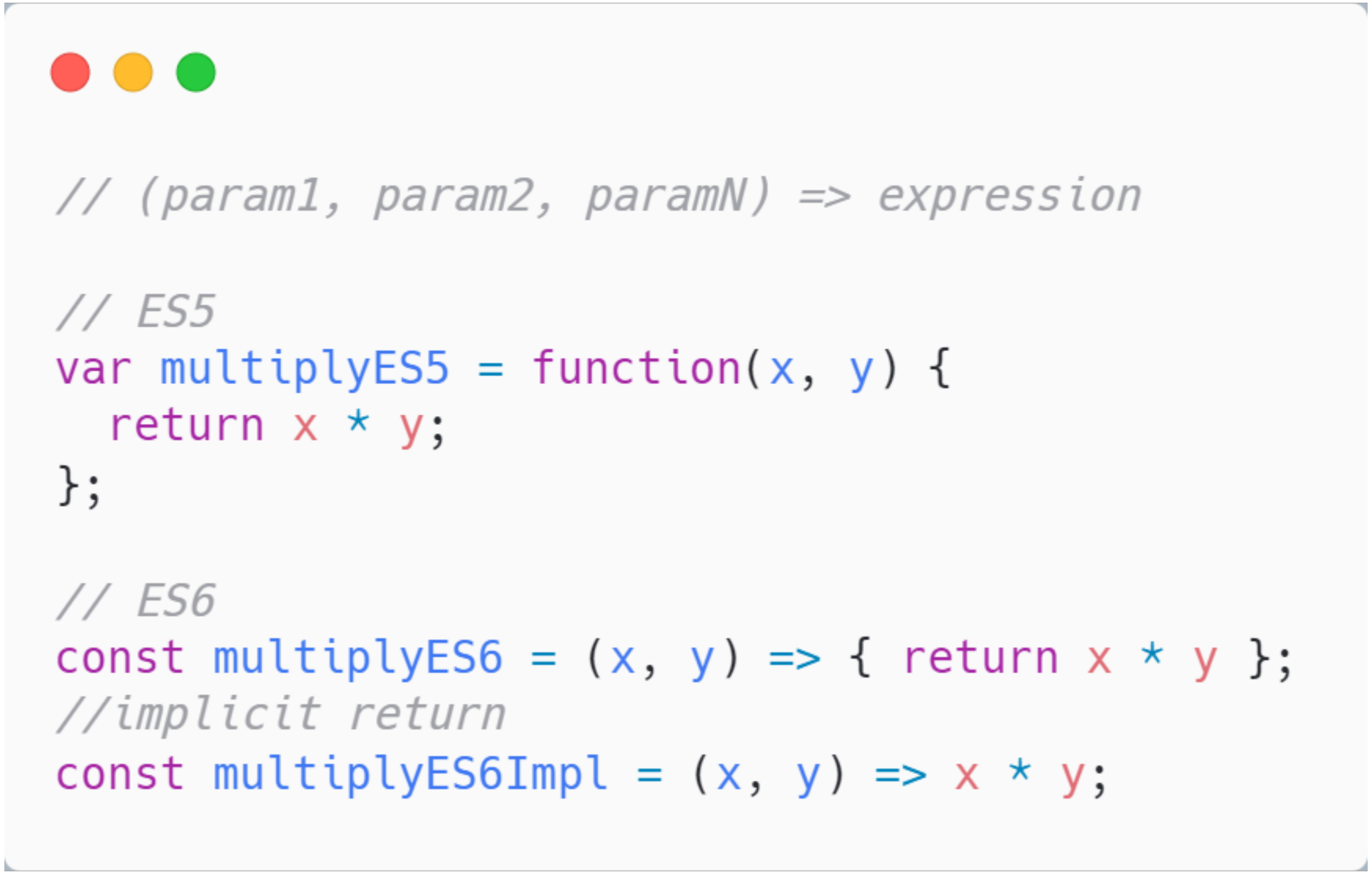


ARROW FUNCTIONS

Arrow functions were introduced with ES6 as a new syntax for writing J avaScript functions. They save developers time and simplify function sco pe

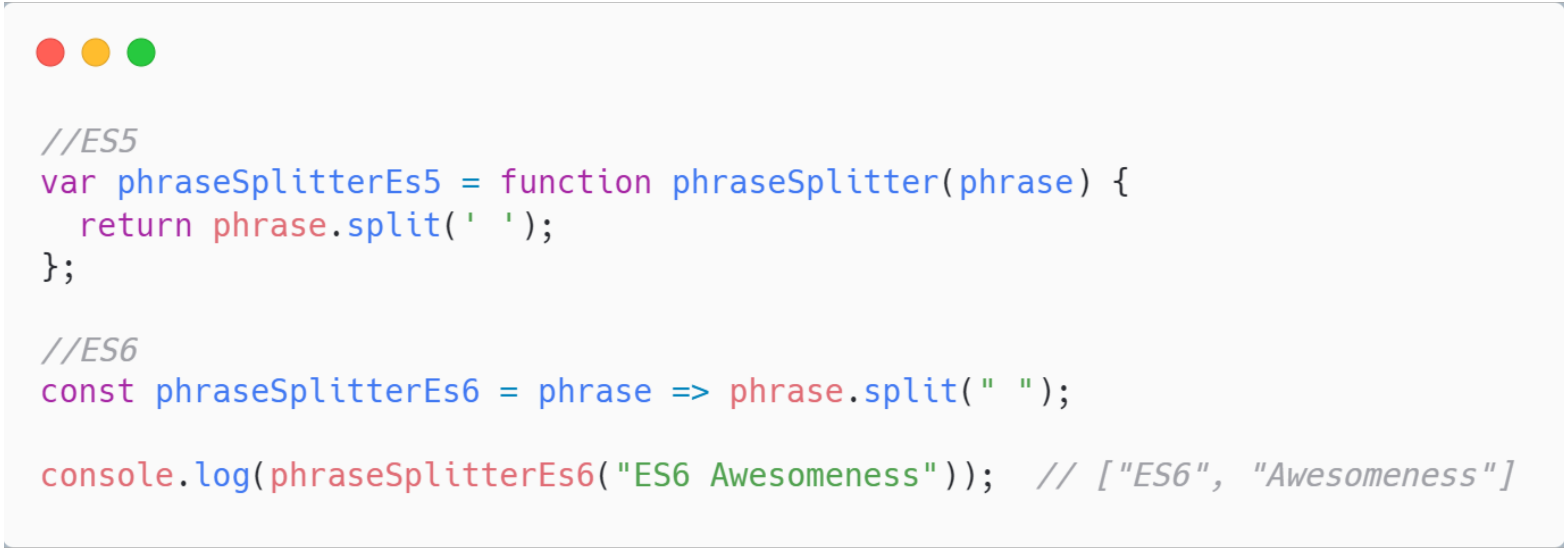
Arrow functions are a more concise syntax for writing function expressio ns. They utilize a new token, =>, that looks like a fat arrow. Arrow functi ons are anonymous and change the way this binds in functions.

ARROW FUNCTIONS



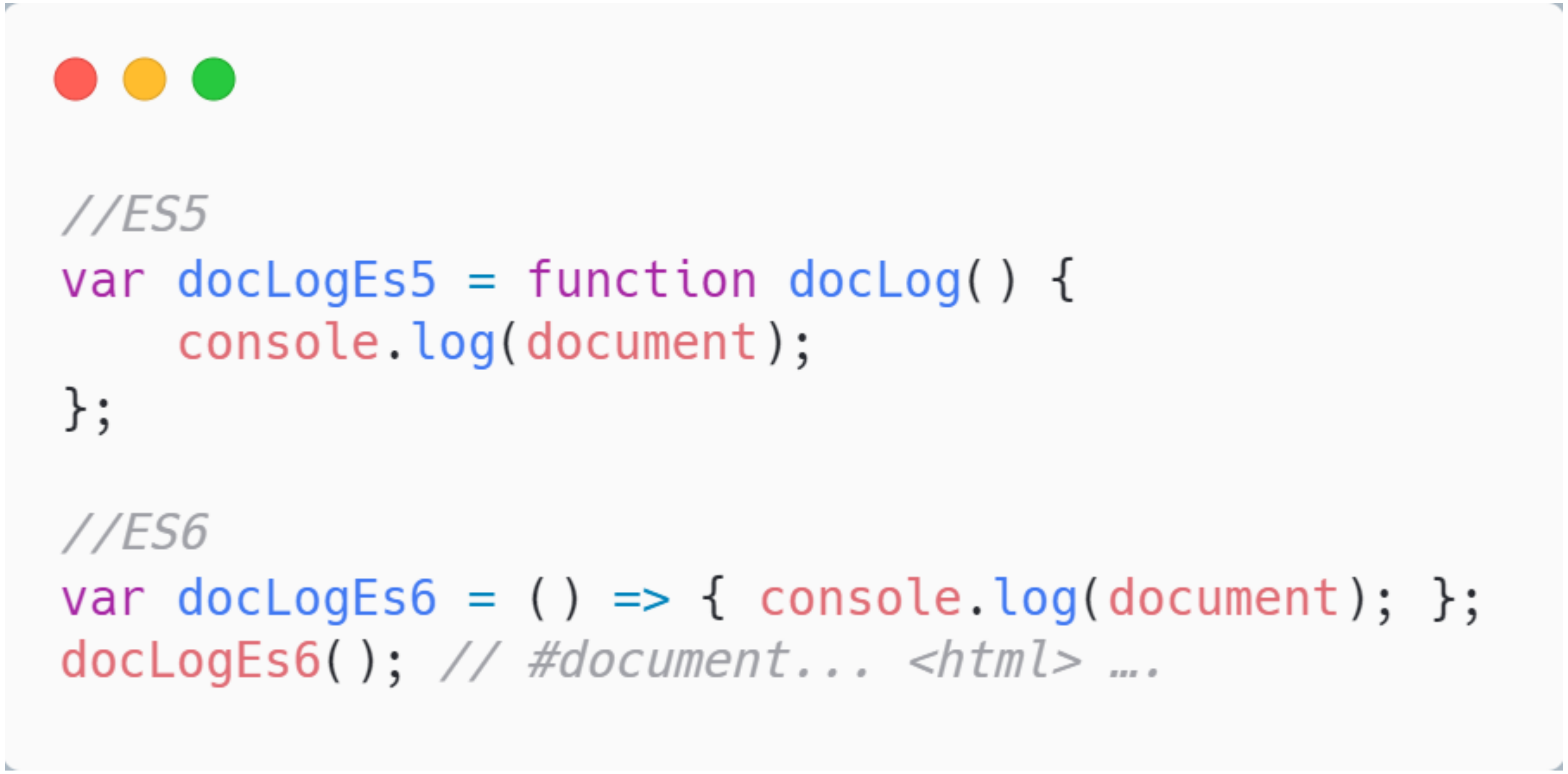
```
// (param1, param2, paramN) => expression

// ES5
var multiplyES5 = function(x, y) {
  return x * y;
};

// ES6
const multiplyES6 = (x, y) => { return x * y };
//implicit return
const multiplyES6Impl = (x, y) => x * y;
```

ARROW FUNCTIONS



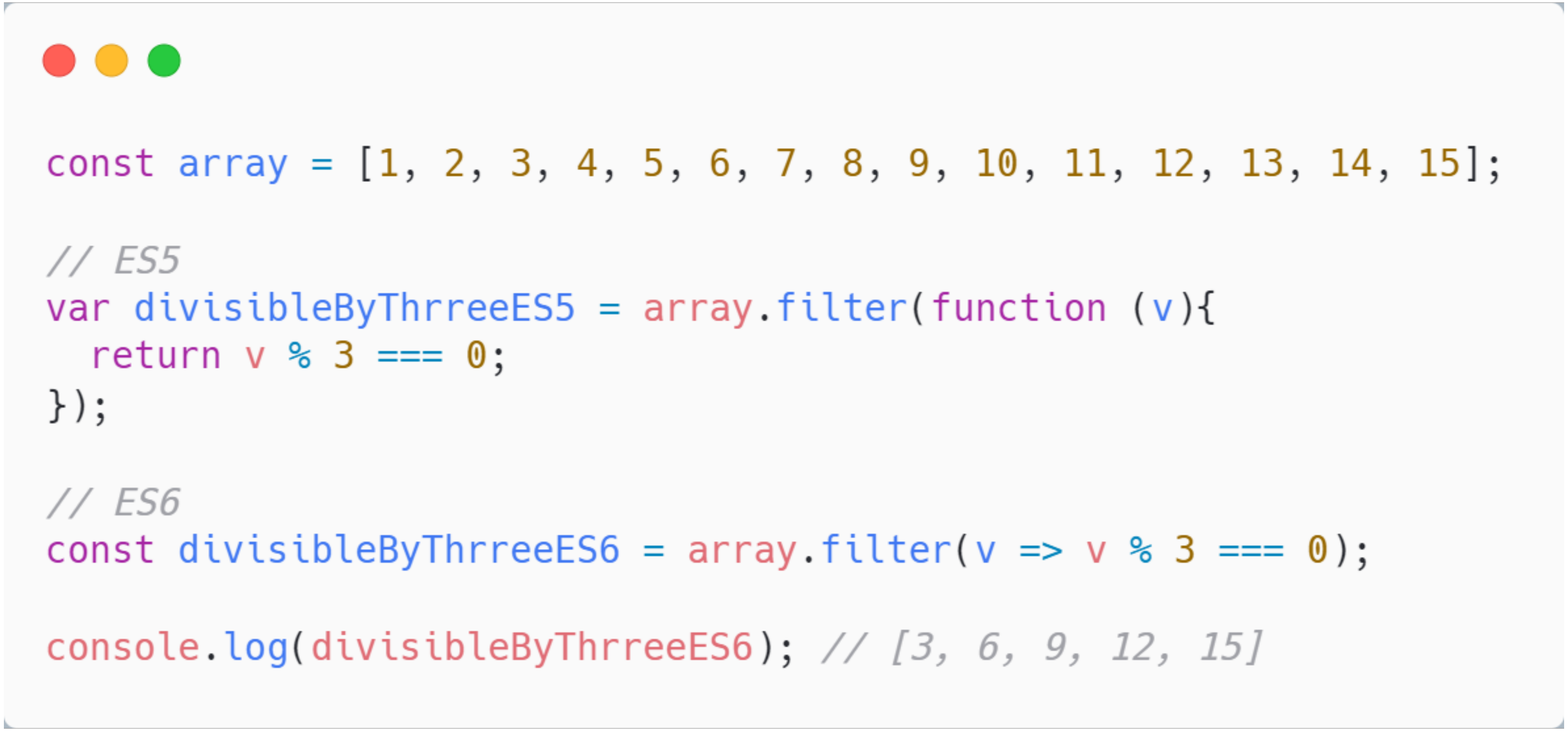
```
//ES5
var phraseSplitterEs5 = function phraseSplitter(phrase) {
  return phrase.split(' ');
};

//ES6
const phraseSplitterEs6 = phrase => phrase.split(" ");

console.log(phraseSplitterEs6("ES6 Awesomeness"));  // ["ES6", "Awesomeness"]
```

ARROW FUNCTIONS



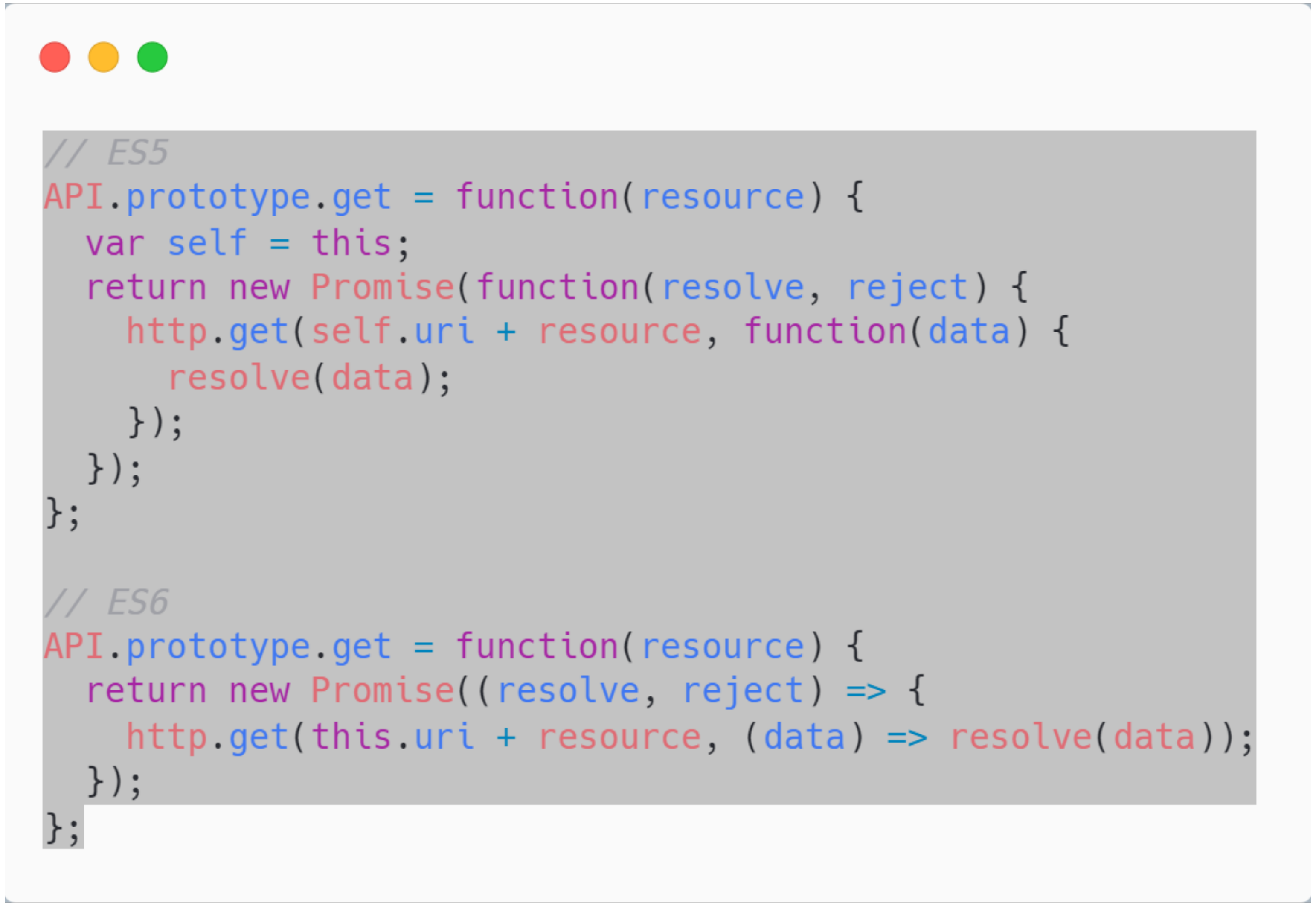
```
//ES5
var docLogEs5 = function docLog() {
    console.log(document);
};

//ES6
var docLogEs6 = () => { console.log(document); };
docLogEs6(); // #document... <html> ….
```

ARROW FUNCTIONS



```
const array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15];

// ES5
var divisibleByThrreeES5 = array.filter(function (v){
  return v % 3 === 0;
});

// ES6
const divisibleByThrreeES6 = array.filter(v => v % 3 === 0);

console.log(divisibleByThrreeES6); // [3, 6, 9, 12, 15]
```

ARROW FUNCTIONS


```
// ES5
API.prototype.get = function(resource) {
  var self = this;
  return new Promise(function(resolve, reject) {
    http.get(self.uri + resource, function(data) {
      resolve(data);
    });
  });
};

// ES6
API.prototype.get = function(resource) {
  return new Promise((resolve, reject) => {
    http.get(this.uri + resource, (data) => resolve(data));
  });
};
```

QUESTIONS