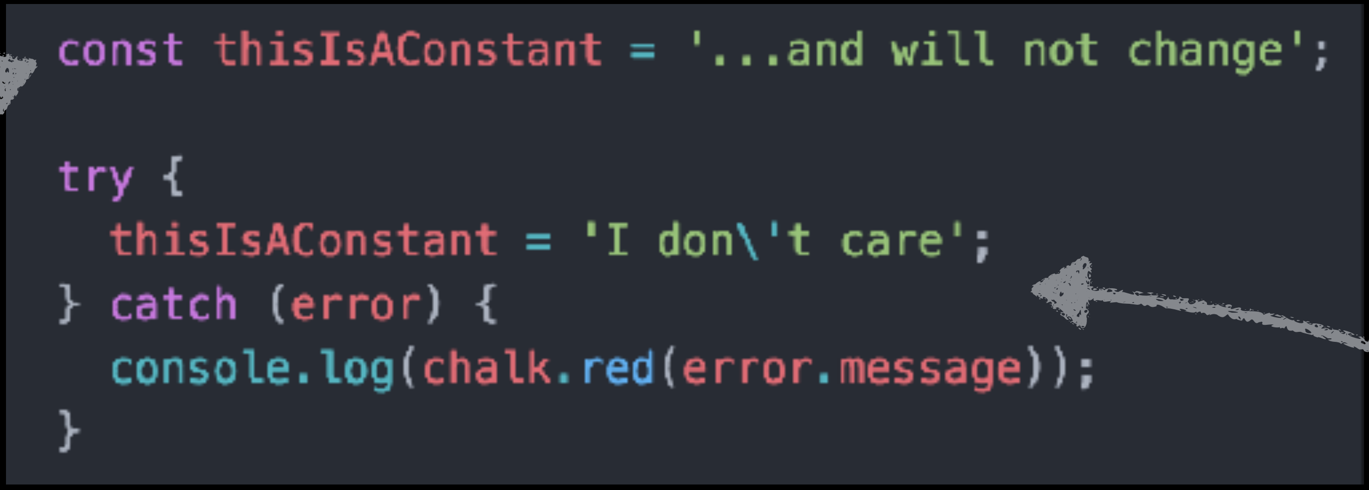
ECMAScript

The newish language for the Web

Index

Const



```
const thisIsAConstant = '...and will not change';

try {
  thisIsAConstant = 'I don\'t care';
} catch (error) {
  console.log(chalk.red(error.message));
}
```

A constant variable stores a value and it is set when you declare the variable.

Trying to change its value throws an exception.

Const

```
try {
  const a = {};
  const b = {};
```

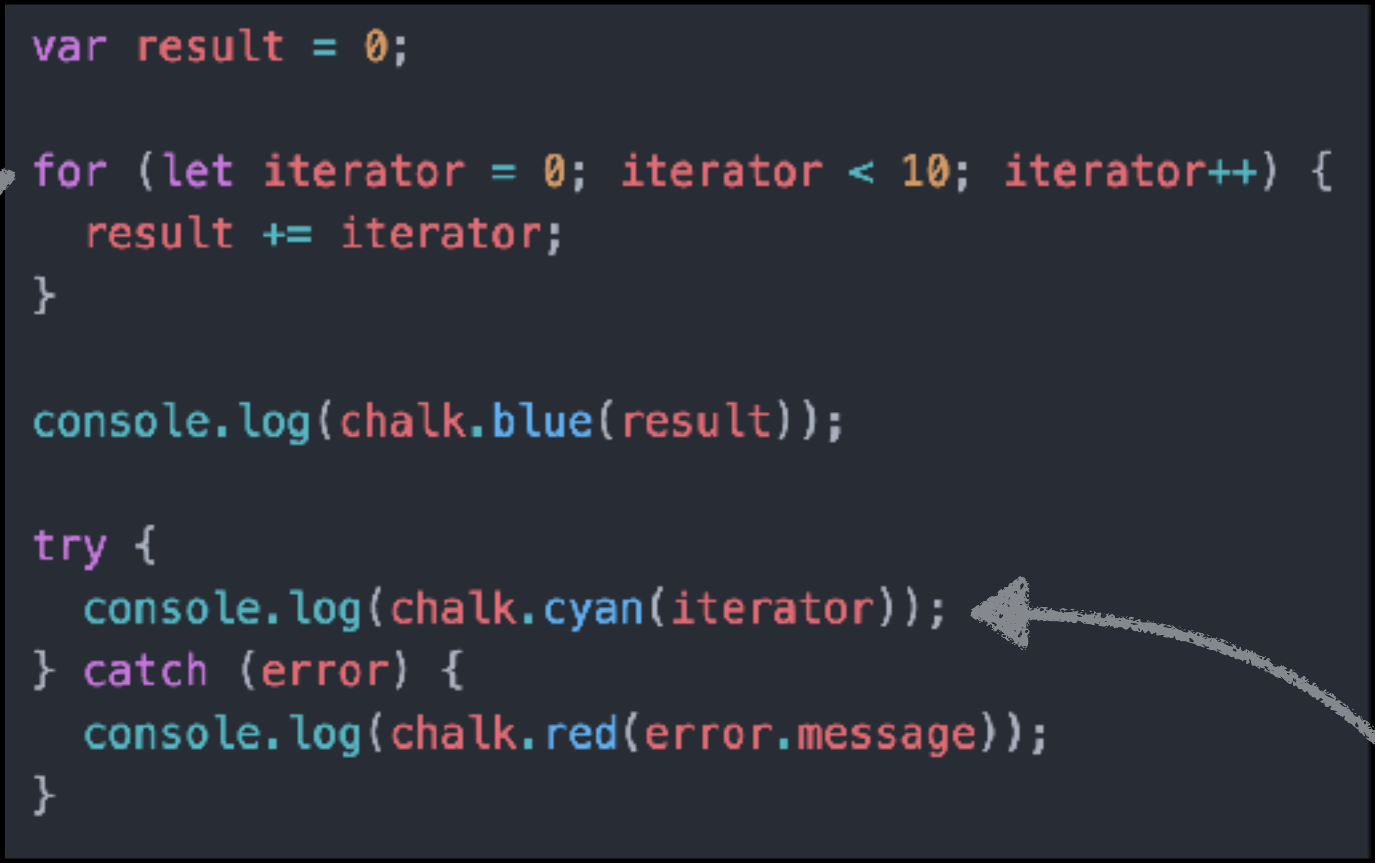
```
a.name = 'John';
```

```
  a = b;
} catch (error) {
  console.log(chalk.red(error.message));
}
```

When you declare an object as constant, you make its reference constant.

So you can modify its contents but not its reference.

Let

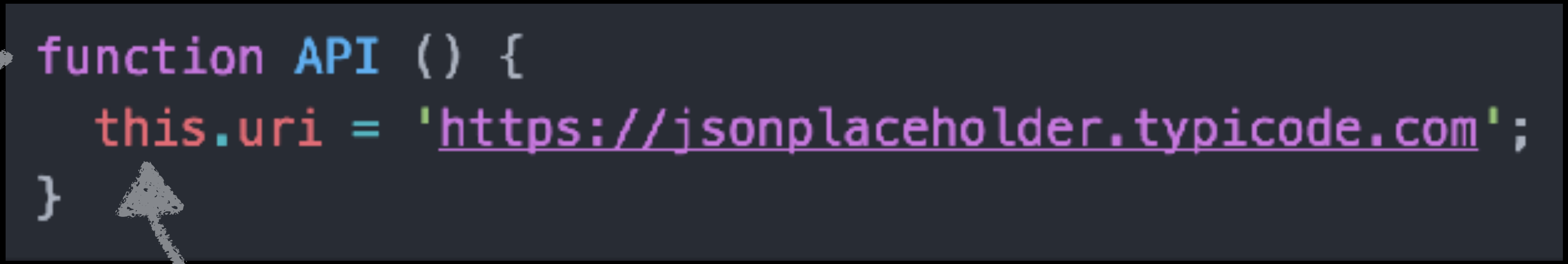


```
var result = 0;

for (let iterator = 0; iterator < 10; iterator++) {
  result += iterator;
}

console.log(chalk.blue(result));

try {
  console.log(chalk.cyan(iterator));
} catch (error) {
  console.log(chalk.red(error.message));
}
```

Let declares variables which only exist insight the block where they are created.

That means we cannot access that variables outside its context.

Function scope

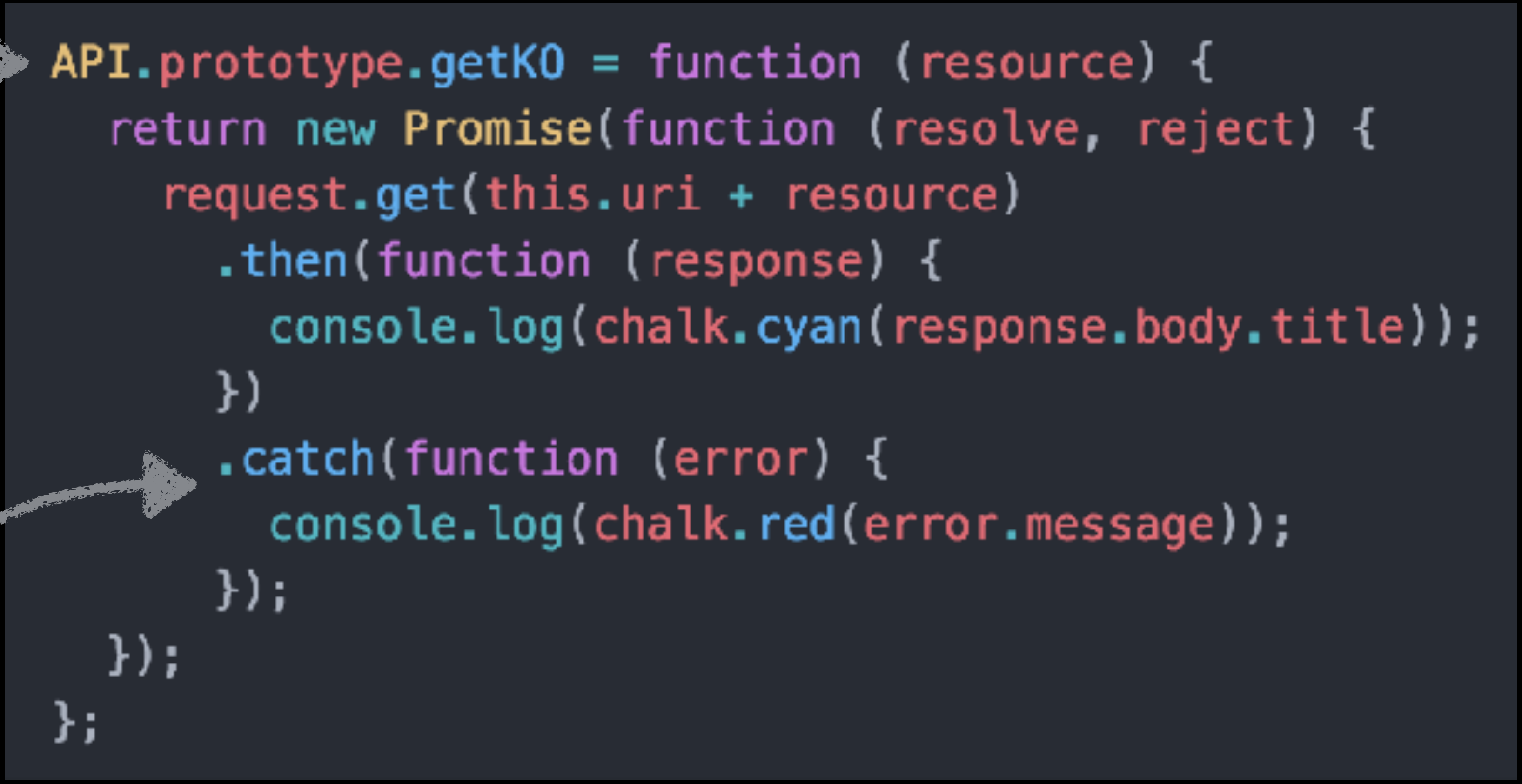


```
function API () {
  this.uri = 'https://jsonplaceholder.typicode.com';
}
```

When you create "classes" in the old fashion way, you use a function. Inside it you declare properties that are associated to the function scope.

The "this" word here represent the function context.

Function scope

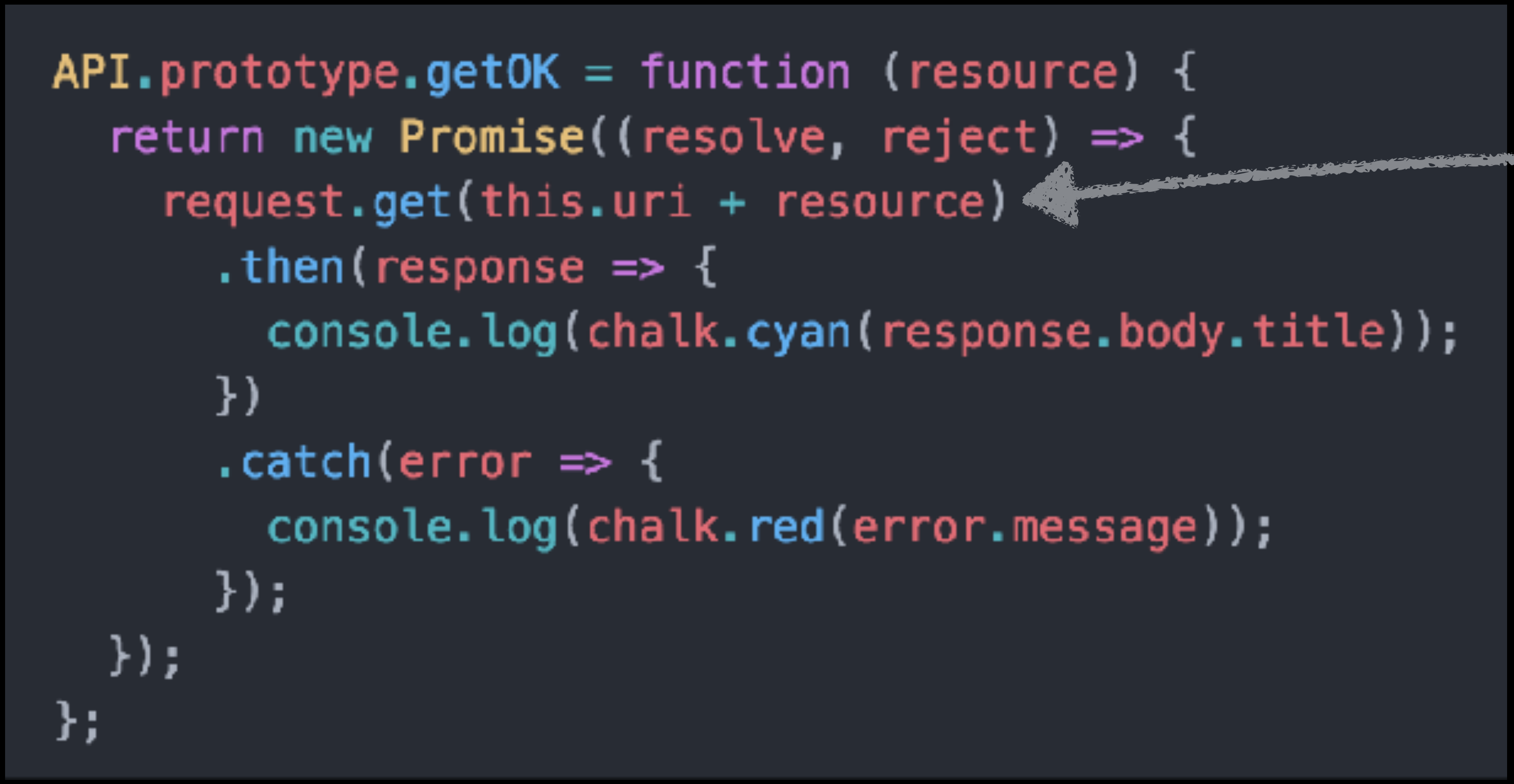


```
API.prototype.getKO = function (resource) {
  return new Promise(function (resolve, reject) {
    request.get(this.uri + resource)
      .then(function (response) {
        console.log(chalk.cyan(response.body.title));
      })
      .catch(function (error) {
        console.log(chalk.red(error.message));
      });
  });
};
```

To create methods, we use the function prototype.

The problem is that each function has its own scope, so we will need the “var self = this;” to access the outer scope inside our inner functions.

Function scope



```
API.prototype.getOK = function (resource) {
  return new Promise((resolve, reject) => {
    request.get(this.uri + resource)
      .then(response => {
        console.log(chalk.cyan(response.body.title));
      })
      .catch(error => {
        console.log(chalk.red(error.message));
      });
  });
};
```

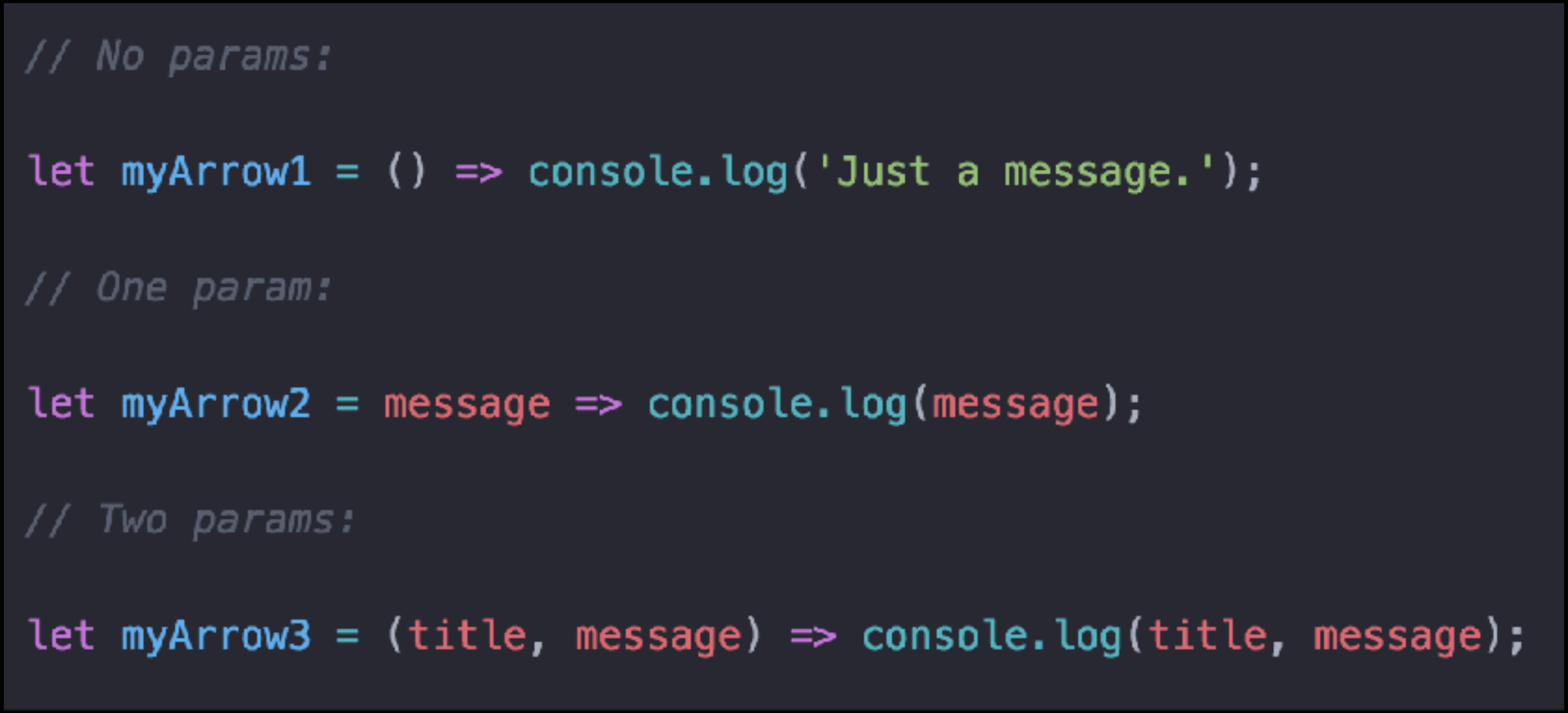
This is an arrow function, and it is transparent to its parent scope.

That means that this code will work. And the reason is because the “this” inside an arrow function is its parent “this”, and “this.url” really exists.

Arrow function =>

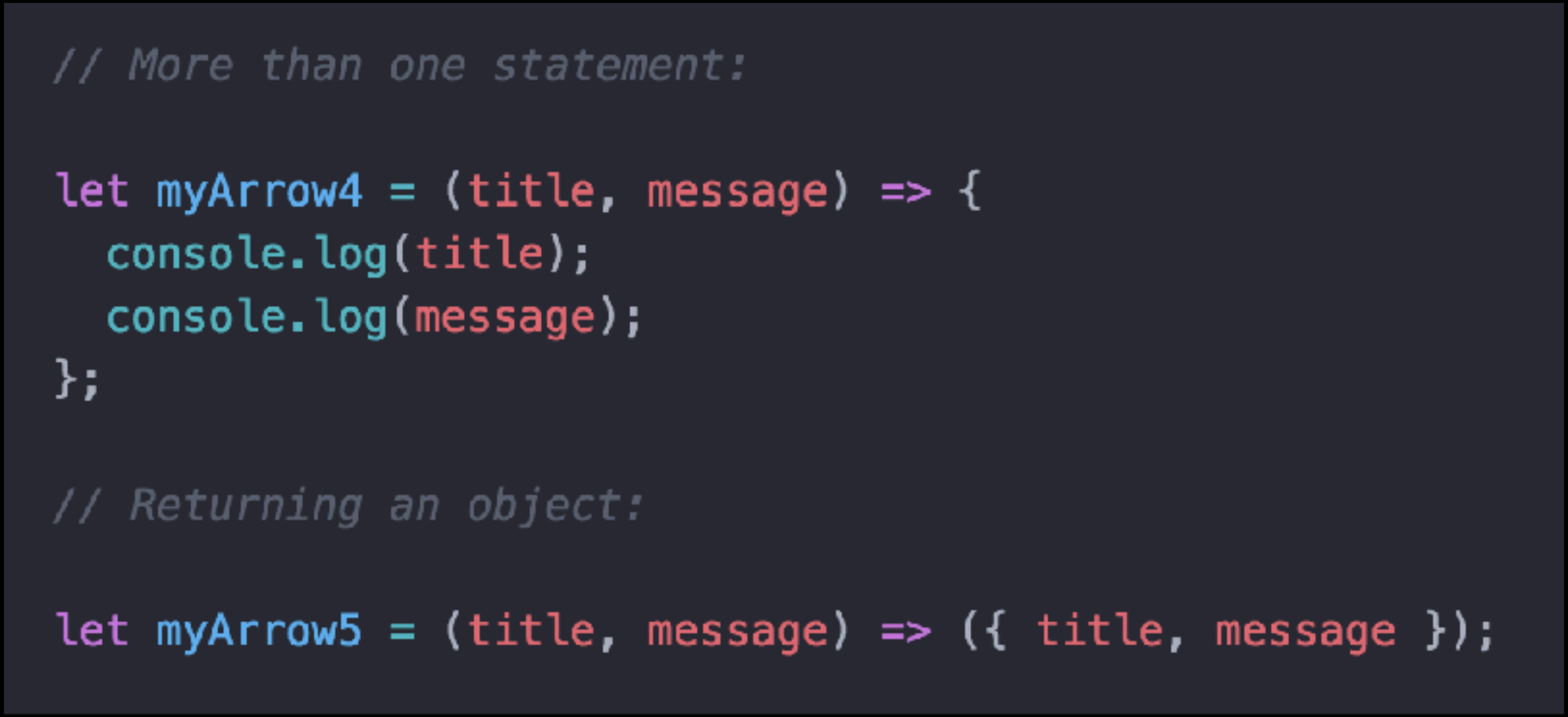
- The arrow function has a defined structure:
  () => {}.
- After the arrow, by default you have a return. So functions in one line, always execute a command or return a value.
- You can have none, one or multiple parameters, and the way you use the parenthesis is different for each case.
- To return and object, you need parenthesis: ({}).

Arrow function =>



```
// No params:

let myArrow1 = () => console.log('Just a message.');

// One param:

let myArrow2 = message => console.log(message);

// Two params:

let myArrow3 = (title, message) => console.log(title, message);
```

Arrow function =>



```javascript
// More than one statement:

let myArrow4 = (title, message) => {
  console.log(title);
  console.log(message);
};

// Returning an object:

let myArrow5 = (title, message) => ({ title, message });
```

Classes

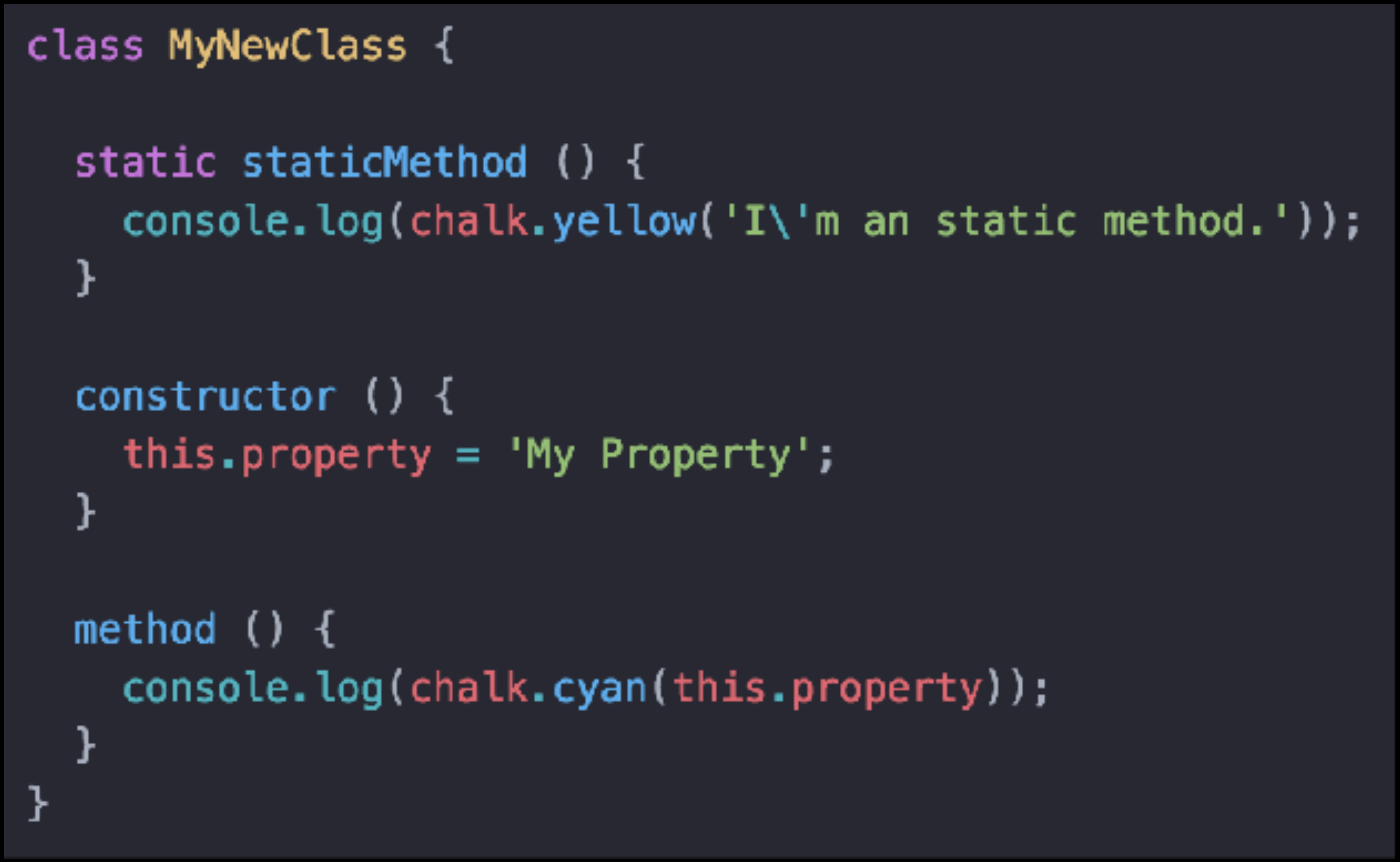
- As you have seen, old fashion way to create classes was using a function and its prototype.

- The modern approach is using “class”.

- A modern class has three main block inside: constructor, static methods, and normal methods.

- Statics methods are called using the class, and normal methods using its instance.

Classes



```
class MyNewClass {

  static staticMethod () {
    console.log(chalk.yellow('I\'m an static method.'));
  }

  constructor () {
    this.property = 'My Property';
  }

  method () {
    console.log(chalk.cyan(this.property));
  }
}
```

Static method

Constructor

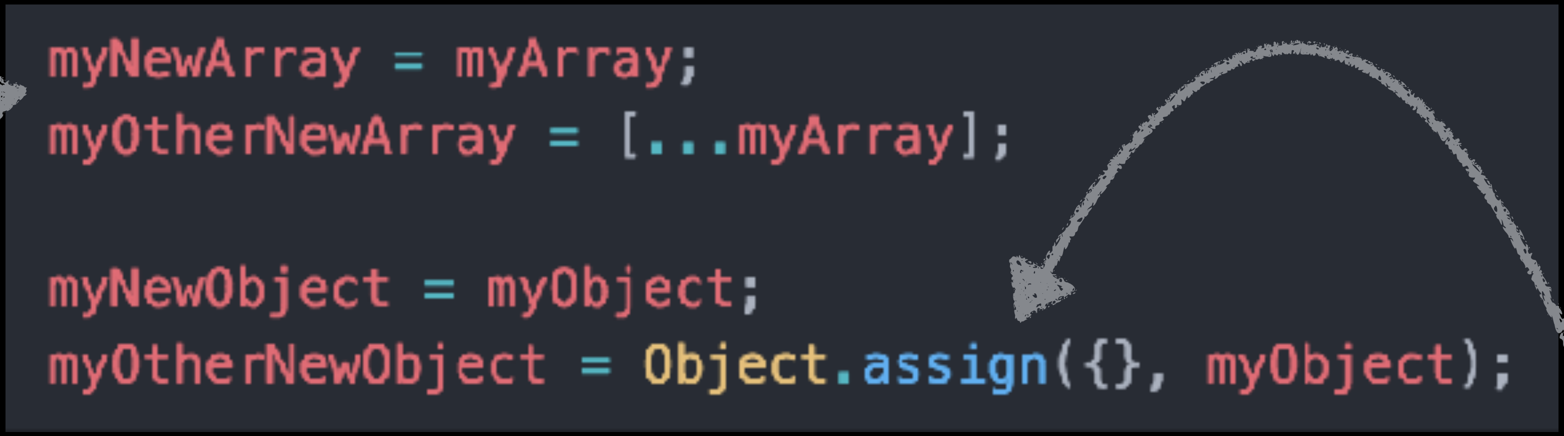
Normal method

To instantiate the class we use the “new” word, and the we can use all the methods of the class using this new object. To use the static methods, we will use the class.

Spread operator

- When you assign an array or an objet using = operator, you are creating a reference over the same objet, so both are equals.
- If you want different object (copy only contents), you can use the spread operator (…) or “assign” method from “Object”.

Spread operator

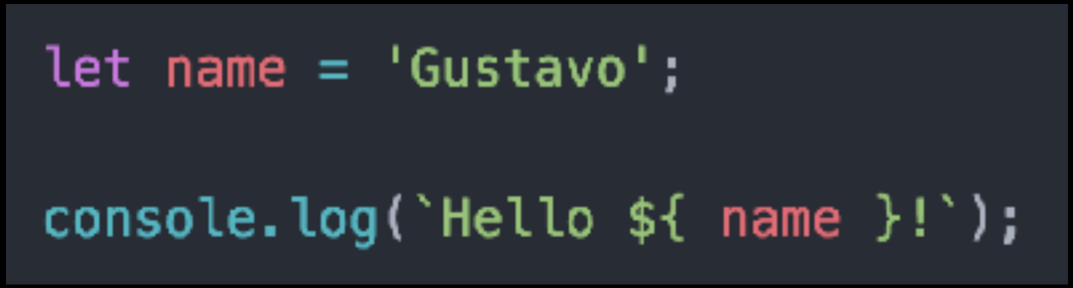


```
myNewArray = myArray;
myOtherNewArray = [...myArray];

myNewObject = myObject;
myOtherNewObject = Object.assign({}, myObject);
```

The spread operator separates every element in an array or group of elements.

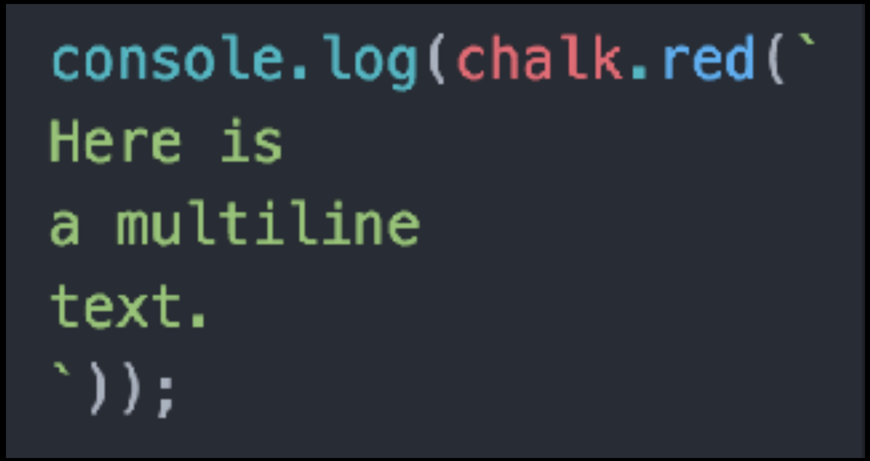
The assign method of Object, returns an object reduced from the properties of other objects.

String template



```
let name = 'Gustavo';

console.log(`Hello ${ name }!`);
```

Template string using a variable to compose the final string.



```
console.log(chalk.red(`
Here is
a multiline
text.
`));
```

Multiline allow us to compose multiline strings without using “\n”.

Asynchrony

- One of the most interesting things that is delivered with the new ES, is asynchrony.

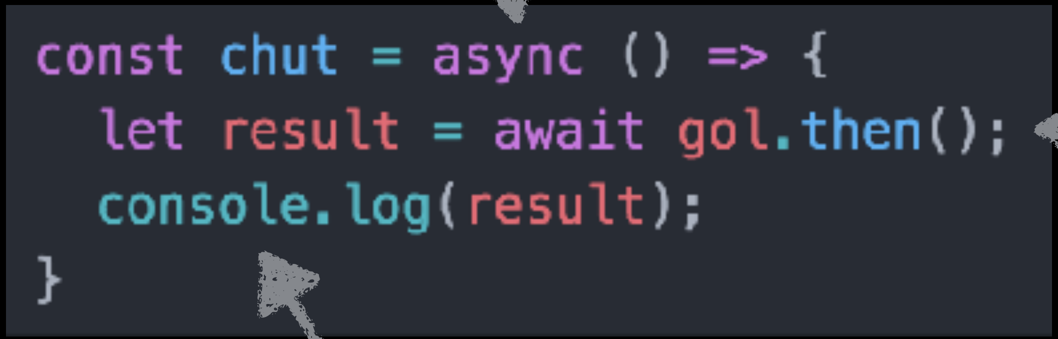
- We no longer will need to use callbacks or promises if it is the own function that waits the responses inside it to continue with its own execution.

- The function is declared as “async”, and we will wait in the “await” spot.

Asynchrony

Here we declare that the function is asynchronous.

Here we wait for the promise to be resolved and return the result.



```
const chut = async () => {
  let result = await gol.then();
  console.log(result);
}
```

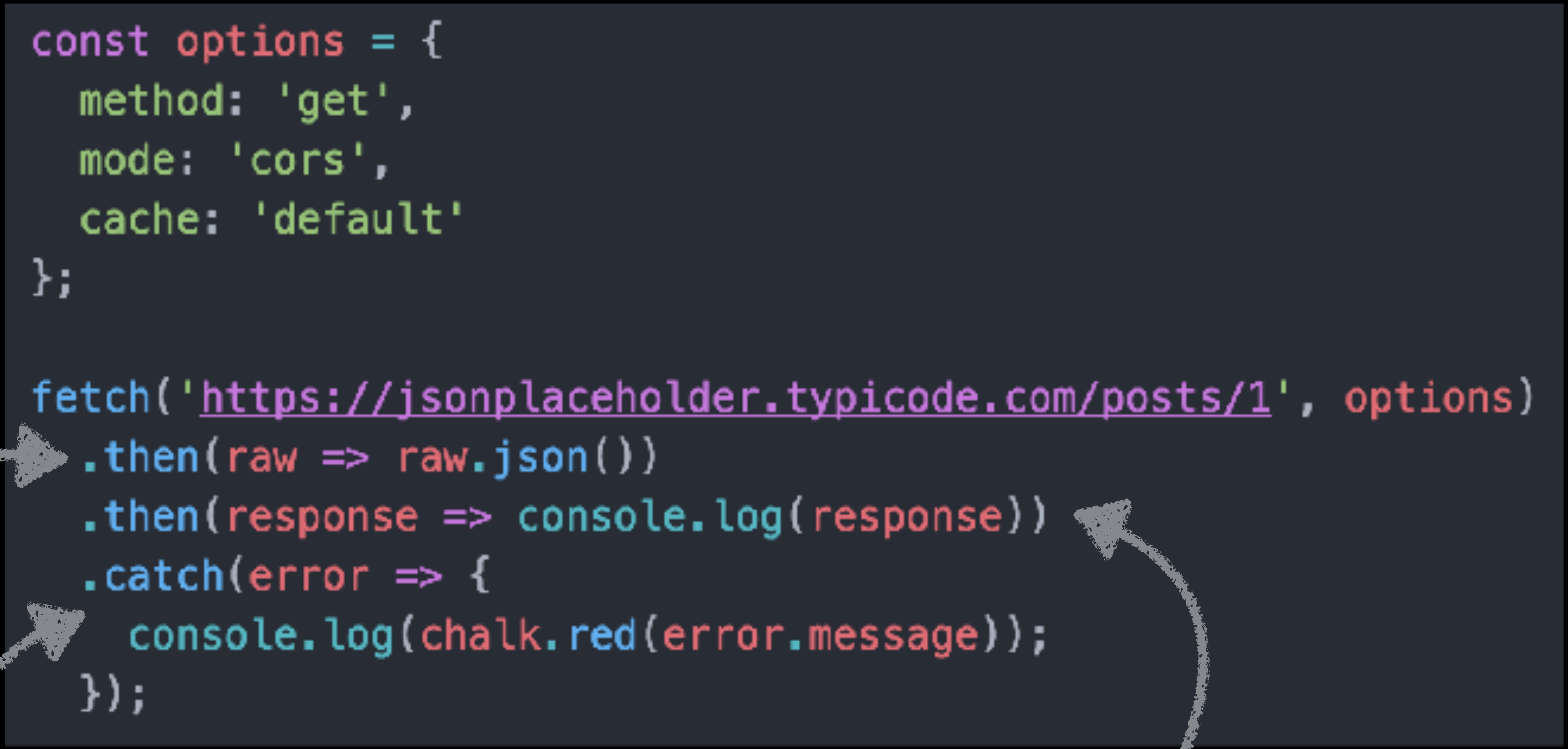
And when we have the result, we print it.

Fetch

- Fetch is the new MDN standard library for AJAX calls, based on promises.
- It needs a configuration object to configure the request to the server.
- The resolver has multiple functions to manage the response format and the it is passed to the next resolver.

Fetch

Options



```
const options = {
  method: 'get',
  mode: 'cors',
  cache: 'default'
};

fetch('https://jsonplaceholder.typicode.com/posts/1', options)
  .then(raw => raw.json())
  .then(response => console.log(response))
  .catch(error => {
    console.log(chalk.red(error.message));
  });
```

Processing the response.

Processing the error.

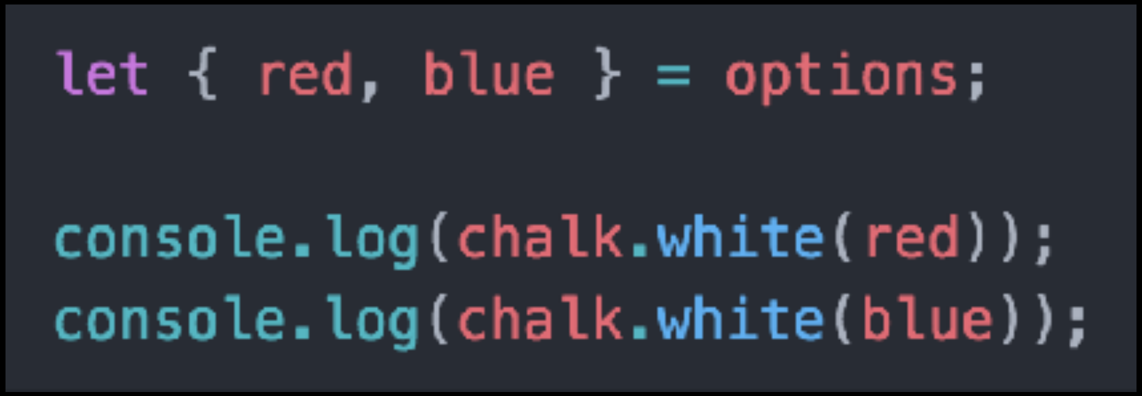
Using the processed data from the response.

Miscellaneous

- Among other things, we have the import and export methods using “module.exports” and “require” because we are using Node 7.7.x.
- Classes can extend other classes using “super” in the constructor.
- Object use variables that matches their properties names without needing to write them.
- “Object.assign()” is a reducer function for objects.

Miscellaneous

- Functions can now get default parameters, so we no longer need to check if the parameter is passed or check its value.
- We can extract variables from objects using {} to declare variables that the object contains.



```
let { red, blue } = options;

console.log(chalk.white(red));
console.log(chalk.white(blue));
```

Useful links

- Atom: https://atom.io/
- Node ES Coverage: http://node.green/
- Node.js: https://nodejs.org/en/
- Pretty console output: https://github.com/chalk/chalk

Thanks and “may the Code be with you”