ES6

The bits you'll *actually* use

Agenda

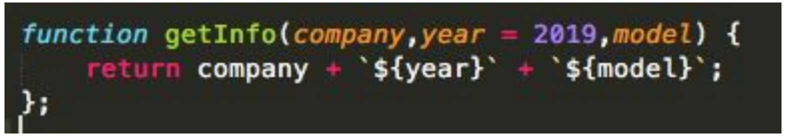
- ❖ Default Parameters
- ❖ Template Literals
- ❖ Multi-line Strings
- ❖ Spread Operator
- ❖ Enhanced Object Literals
- ❖ Arrow Functions
- ❖ Block-Scoped Constructs Let and Const
- ❖ Classes
- ❖ How to use ES6?

Quick History

1. 1995: JavaScript is born as LiveScript
2. 1997: ECMAScript standard is established
3. 1999: ES3 comes out and IE5 is all the rage
4. 2000–2005: XMLHttpRequest, a.k.a. AJAX, gains popularity in app such as Outlook Web Access (2000) and Oddpost (2002), Gmail (2004) and Google Maps (2005).
5. 2009: ES5 comes out with forEach, Object.keys, Object.create and standard JSON
6. 2015: ES6/ECMAScript2015 comes out; it has mostly syntactic sugar.

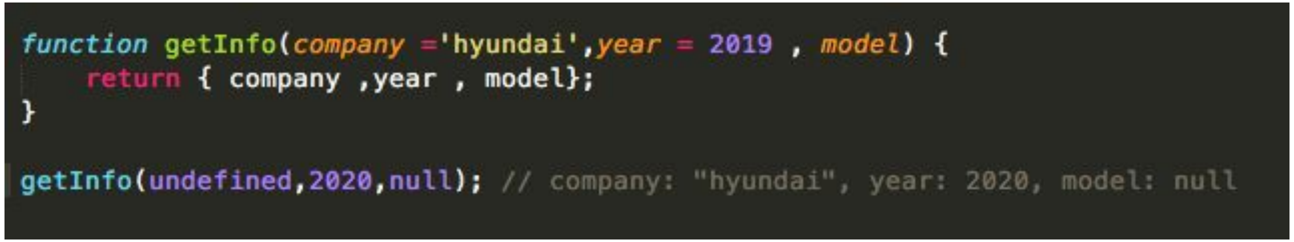
Default Parameters

Default parameters allow us to initialize functions with default values. A default is used when an argument is either omitted or undefined — meaning null is a valid value.



```
function getInfo(company,year = 2019,model) {
    return company + `${year}` + `${model}`;
};
```

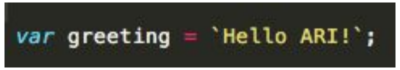
#Dealing with Omitted Values and null



```
function getInfo(company ='hyundai',year = 2019 , model) {
    return { company ,year , model};
}

getInfo(undefined,2020,null); // company: "hyundai", year: 2020, model: null
```

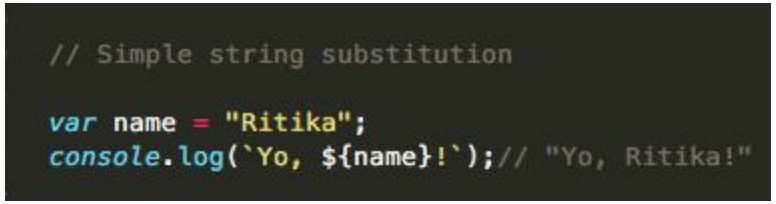
Template Literals

Template Strings use backticks (``) rather than the single or double quotes we're used to with regular strings. A template string could thus be written as follows:



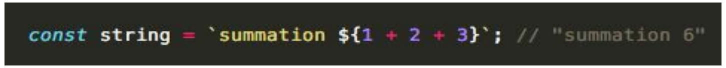
```
var greeting = `Hello ARI!`;
```

One of their first real benefits is string substitution. Template Strings can contain placeholders for string substitution using the `${ }` syntax, as demonstrated below:



```
// Simple string substitution

var name = "Ritika";
console.log(`Yo, ${name}!`);// "Yo, Ritika!"
```

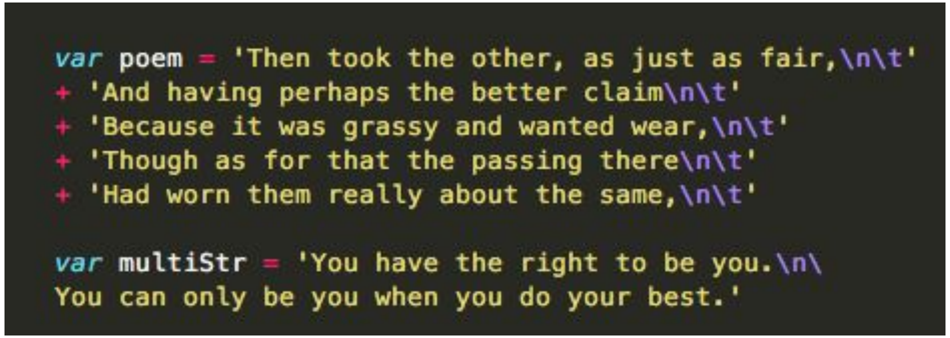
inside the ${} you can add anything, even expressions:



```
const string = `summation ${1 + 2 + 3}`; // "summation 6"
```

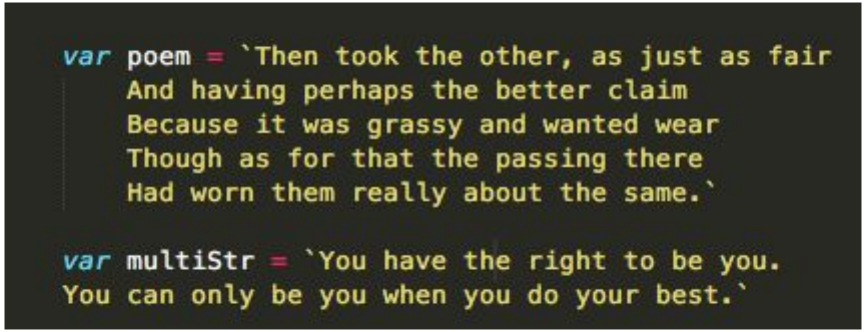
Multi Line Strings ` `

Another yummy syntactic sugar is multi-line string. In ES5, we had to use one of these approaches:



```
var poem = 'Then took the other, as just as fair,\n\t'
+ 'And having perhaps the better claim\n\t'
+ 'Because it was grassy and wanted wear,\n\t'
+ 'Though as for that the passing there\n\t'
+ 'Had worn them really about the same,\n\t'

var multiStr = 'You have the right to be you.\n\
You can only be you when you do your best.'
```

While in ES6, simply utilize the backticks:

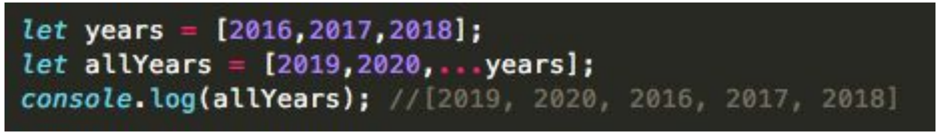


```
var poem = `Then took the other, as just as fair
    And having perhaps the better claim
    Because it was grassy and wanted wear
    Though as for that the passing there
    Had worn them really about the same.`

var multiStr = `You have the right to be you.
You can only be you when you do your best.`
```

Spread Operator {...}

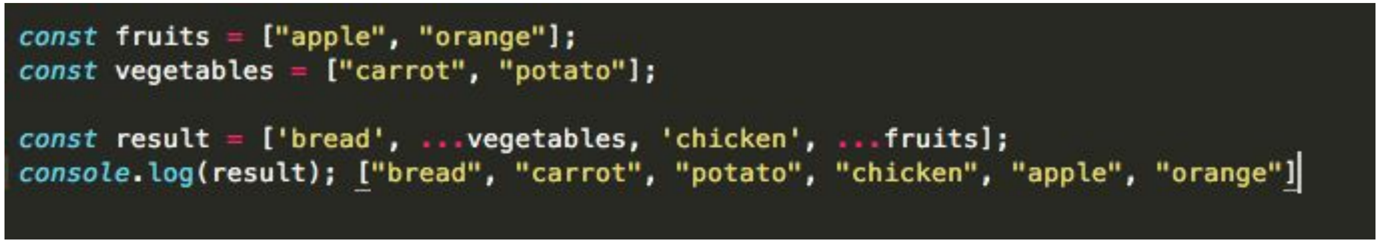
JavaScript ES6 (ECMAScript 6) introduced the spread operator. The syntax is **three dots(...)** followed by the array (or iterable*). It expands the array into individual elements. So, it can be used to expand the array in a places where zero or more elements are expected.

- ❖ **Copying an array**



```
let years = [2016,2017,2018];
let allYears = [2019,2020,...years];
console.log(allYears); //[2019, 2020, 2016, 2017, 2018]
```

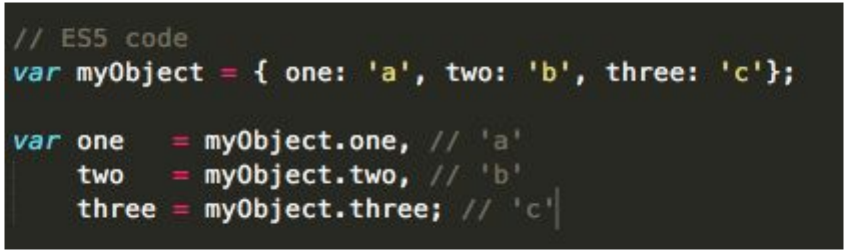
- ❖ **Concatenating Two Array**



```
const fruits = ["apple", "orange"];
const vegetables = ["carrot", "potato"];

const result = ['bread', ...vegetables, 'chicken', ...fruits];
console.log(result); ["bread", "carrot", "potato", "chicken", "apple", "orange"]
```

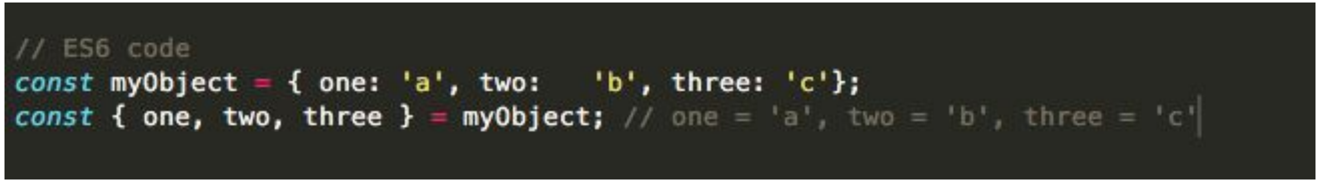
Destructuring Variables From Object Property

It’s often necessary to extract a property value from an object into another variable. This had to be explicitly declared in ES5. For example:



```
// ES5 code
var myObject = { one: 'a', two: 'b', three: 'c'};

var one   = myObject.one, // 'a'
    two   = myObject.two, // 'b'
    three = myObject.three; // 'c'
```

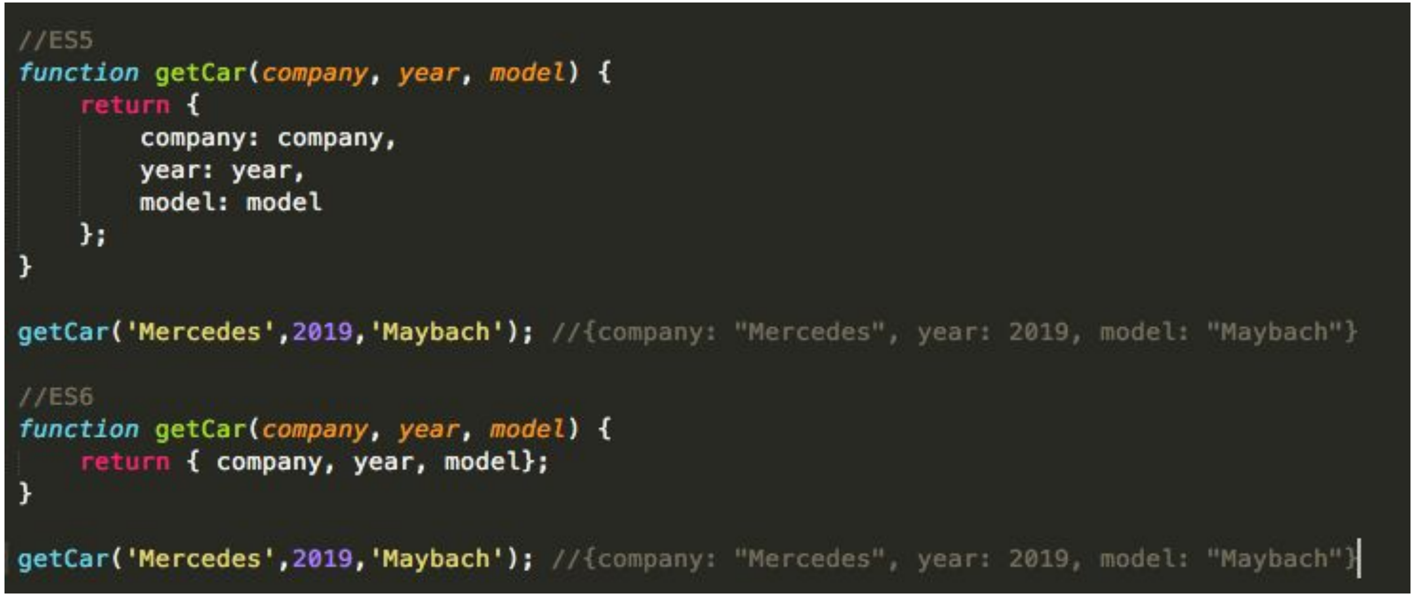
ES6 supports destructuring: you can create a variable with the same name as an equivalent object property. For example:



```
// ES6 code
const myObject = { one: 'a', two:   'b', three: 'c'};
const { one, two, three } = myObject; // one = 'a', two = 'b', three = 'c'
```

Enhanced Object Literals

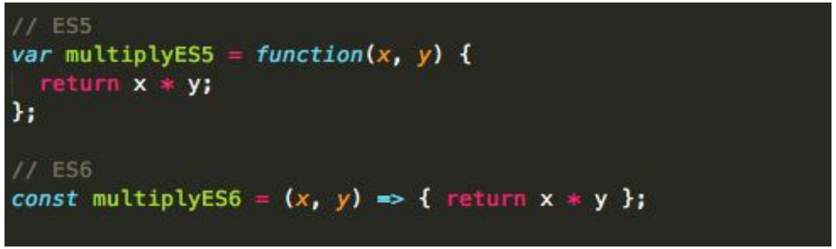
Object literals make it easy to quickly create objects with properties inside the curly braces.ES6 makes the declaring of object literals concise and thus easier.ES6 removes all of that repetition.



```
//ES5
function getCar(company, year, model) {
    return {
        company: company,
        year: year,
        model: model
    };
}

getCar('Mercedes',2019,'Maybach'); //{company: "Mercedes", year: 2019, model: "Maybach"}

//ES6
function getCar(company, year, model) {
    return { company, year, model};
}

getCar('Mercedes',2019,'Maybach'); //{company: "Mercedes", year: 2019, model: "Maybach"}
```

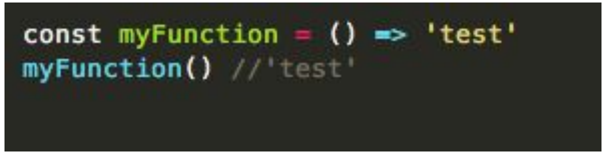
Arrow Functions () => {}

Arrow functions – also called “fat arrow” functions, are a more concise syntax for writing function expressions. They utilize a new token, `=>`, that looks like a fat arrow. Arrow functions are anonymous and change the way `this` binds in functions.



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

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

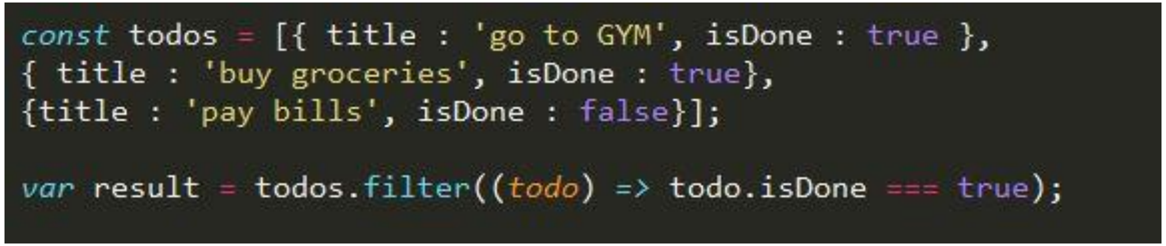
Arrow functions allow you to have an implicit return: values are returned without having to use the return keyword.It works when there is a one-line statement in the function body:



```
const myFunction = () => 'test'
myFunction() //'test'
```

Usage

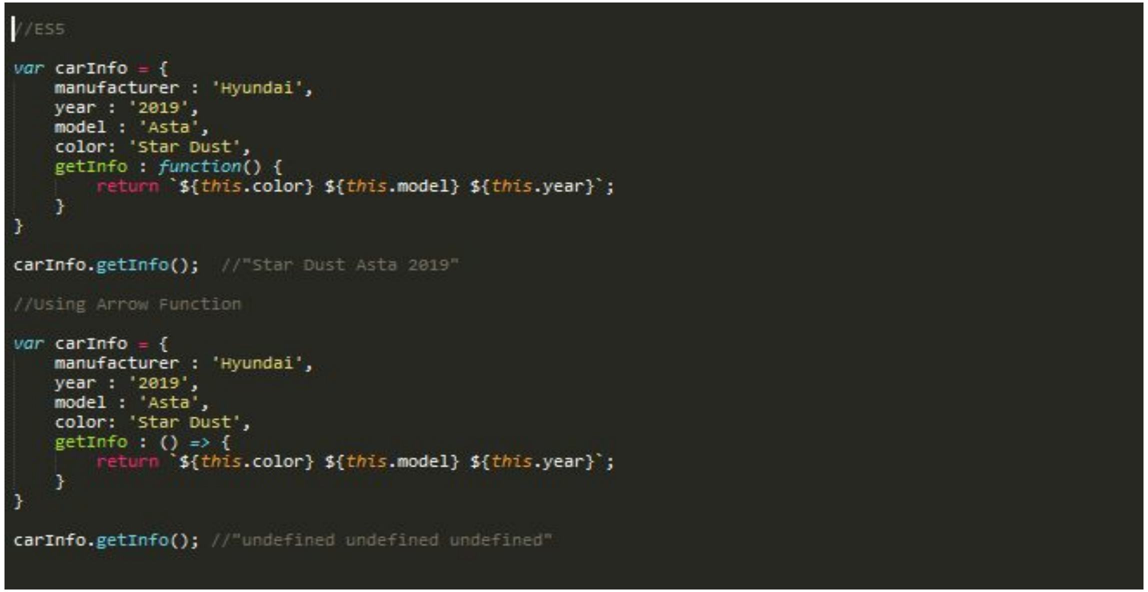
One of the primary use cases for traditional lambda functions, and now for arrow functions in JavaScript, is for functions that get applied over and over again to items in a list.



```
const todos = [{ title : 'go to GYM', isDone : true },
{ title : 'buy groceries', isDone : true},
{title : 'pay bills', isDone : false}];

var result = todos.filter((todo) => todo.isDone === true);
```

When ‘NOT’ to use Arrow Functions

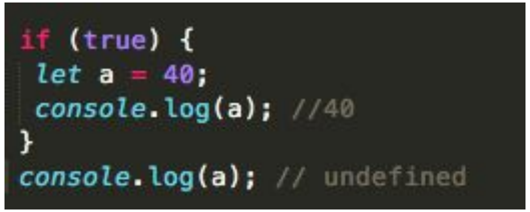
Unlike every other form of function, arrow functions do not have their own execution context which means that both this and arguments are *inherited* from their parent function.One of the example is not to use in the method which is stored as property of an object.



```
//ES5

var carInfo = {
    manufacturer : 'Hyundai',
    year : '2019',
    model : 'Asta',
    color: 'Star Dust',
    getInfo : function() {
        return `${this.color} ${this.model} ${this.year}`;
    }
}

carInfo.getInfo();  //"Star Dust Asta 2019"

//Using Arrow Function

var carInfo = {
    manufacturer : 'Hyundai',
    year : '2019',
    model : 'Asta',
    color: 'Star Dust',
    getInfo : () => {
        return `${this.color} ${this.model} ${this.year}`;
    }
}

carInfo.getInfo(); //"undefined undefined undefined"
```

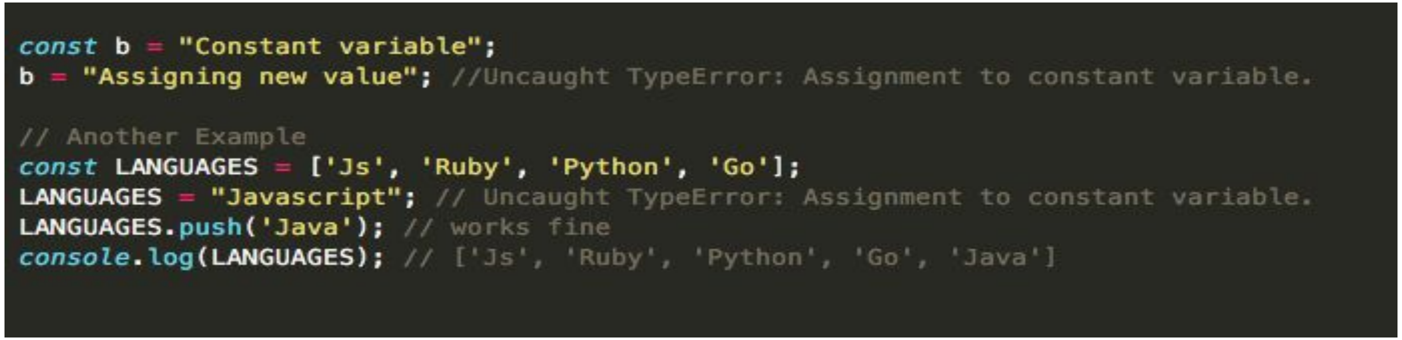
Block Scope Constructs Let and Const

let is a new var which allows to scope the variable to the blocks. We define blocks by the curly braces. In ES5, the blocks did NOTHING to the vars.In ES6, we use let to restrict the scope to the blocks. Vars are function scoped.



```
if (true) {
 let a = 40;
 console.log(a); //40
}
console.log(a); // undefined
```

When it comes to const, things are easier; it’s just an immutable, and it’s also block-scoped like let.
Const is used to assign a constant value to the variable. And the value cannot be changed. Its fixed.

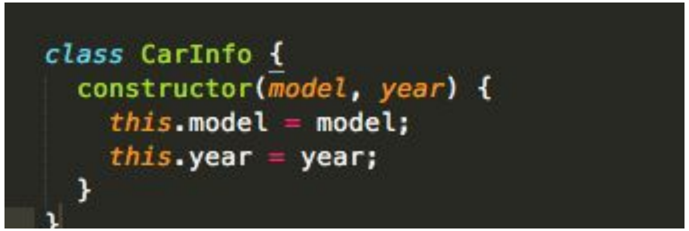


```
const b = "Constant variable";
b = "Assigning new value"; //Uncaught TypeError: Assignment to constant variable.

// Another Example
const LANGUAGES = ['Js', 'Ruby', 'Python', 'Go'];
LANGUAGES = "Javascript"; // Uncaught TypeError: Assignment to constant variable.
LANGUAGES.push('Java'); // works fine
console.log(LANGUAGES); // ['Js', 'Ruby', 'Python', 'Go', 'Java']
```

Classes

Classes are in fact "special functions", and just as you can define function expressions and function declarations, the class syntax has two components: class expressions and class declarations.

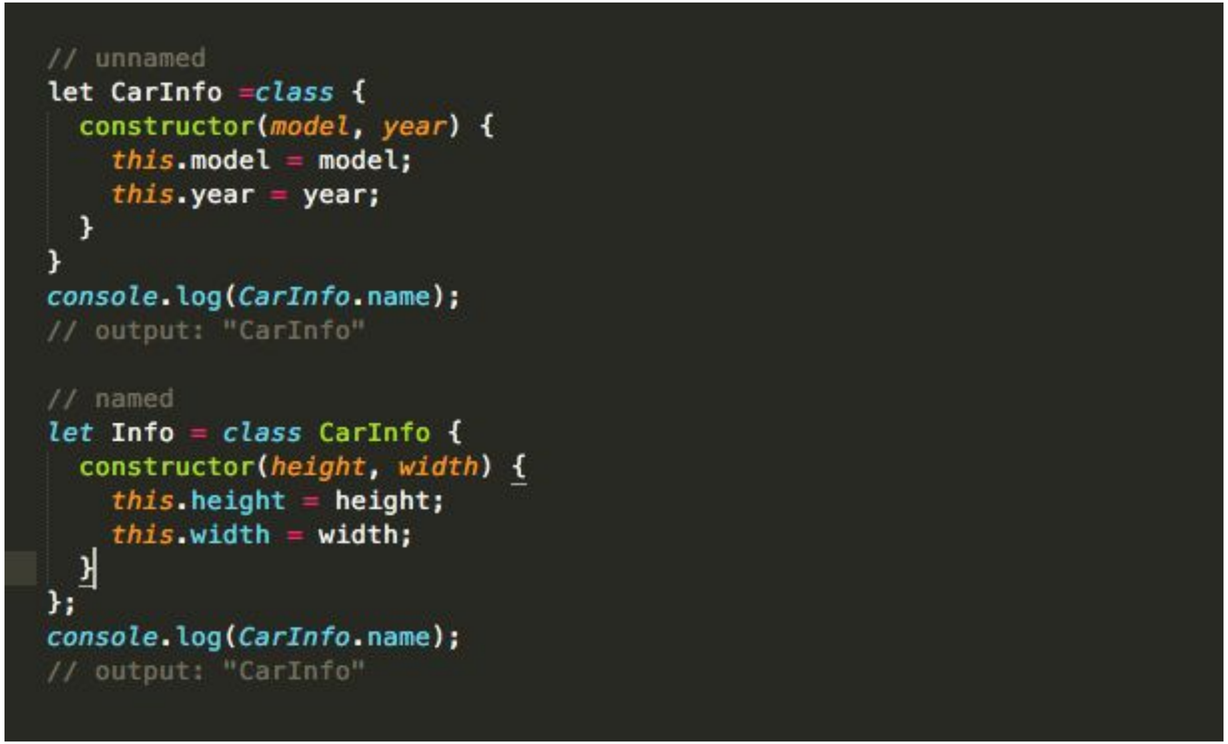
One way to define a class is using a class declaration.



```
class CarInfo {
  constructor(model, year) {
    this.model = model;
    this.year = year;
  }
}
```

The constructor method is a special method for creating and initializing an object created with a class. There can only be one special method with the name "constructor" in a class. A SyntaxError will be thrown if the class contains more than one occurrence of a constructor method.

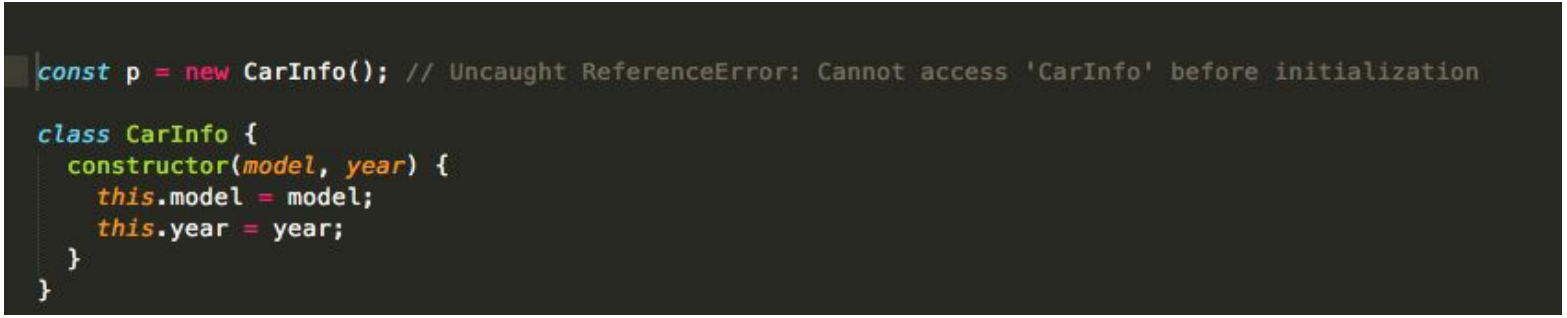
A class expression is another way to define a class. Class expressions can be named or unnamed. The name given to a named class expression is local to the class's body.(it can be retrieved through the class's (not an instance's) name property, though).



```
// unnamed
let CarInfo =class {
  constructor(model, year) {
    this.model = model;
    this.year = year;
  }
}
console.log(CarInfo.name);
// output: "CarInfo"

// named
let Info = class CarInfo {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
console.log(CarInfo.name);
// output: "CarInfo"
```

Hoisting

An important difference between **function declarations** and **class declarations** is that function declarations are hoisted and class declarations are not. You first need to declare your class and then access it, otherwise code like the following will throw a ReferenceError:



```
const p = new CarInfo(); // Uncaught ReferenceError: Cannot access 'CarInfo' before initialization

class CarInfo {
  constructor(model, year) {
    this.model = model;
    this.year = year;
  }
}
```

How to use ES6 today?

ES6 is finalized, but not fully supported by all browsers (e.g., ES6 Firefox support). To use ES6 today, get a compiler like Babel. You can run it as a standalone tool or use with your build system. There are Babel plugins for Grunt, Gulp and Webpack.

That's a
WRAP.