## Переменные

**LET**

Let – ограничивает область видимости переменной блоком, в котором выполнено объявление.

*if(true) {*

*let x = 1;*

*}*

*console.log(x); // undefined*

**CONST**

С помощью  const  объявляется ссылка на значение, доступную только для чтения. Значение переменной необходимо задать непосредственно при объявлении. Если попытаться изменить значение переменной или сразу не задать значение, то выведется сообщение об ошибке:

*const MY\_CONSTANT = 1;*

*MY\_CONSTANT = 2 // Error*

*const SOME\_CONST; // Error*

Однако изменять значения свойств объекта или членов массива всё ещё можно:

*const MY\_OBJECT = {some: 1};*

*MY\_OBJECT.some = 'body'; // Cool*

**Классы**

Классы построены вокруг ключевых слов class и constructor:

*class Vehicle {*

*constructor(name) {*

*this.name = name;*

*this.kind = 'vehicle';*

*}*

*getName() {*

*return this.name;*

*} }*

*// Создадим экземпляр*

*let myVehicle = new Vehicle('rocky');*

Определение класса не является обычным объектом, следовательно, нет запятых между членами класса.

Чтобы создать экземпляр класса, необходимо использовать ключевое слово new. Для наследования базового класса - extend:

*class Car extends Vehicle {*

*constructor(name) {*

*super(name);*

*this.kind = 'car'*

*}}*

*let myCar = new Car('bumpy');*

*myCar.getName(); // 'bumpy'*

*myCar instanceof Car; // true*

*myCar instanceof Vehicle; //true*

**Определения методов**

В **EcmaScript5** методы - свойства, значениями которых является функция.

*var obj = {  
  method: function() {  
  }};*

В **EcmaScript6** методы также являются функциями, однако объявлять их можно более компактно

*let obj {  
  method() {  
  }};*

Геттеры и Сеттеры продолжают работать, как они работали в **EcmaScript 5**

*let obj = {  
  get foo() {  
   console.log('GET foo');  
   return 123;  
  },  
  set bar(value) {  
   console.log('SET bar to '+value);  
   // возвращаемое значение игнорируется  
  }};*

Давайте использовать объект

*> obj.foo  
GET foo  
123  
> obj.bar = true  
SET bar to true  
true*

Существует также способ кратко определить свойства, значения которых являются функциями генератора

*let obj = {  
  \* myGeneratorMethod() {  
  }};*

Этот код равен этому:

*let obj = {  
  myGeneratorMethod: function\* () {  
  }};*

**Сокращения значений свойств**

Сокращения позволяют сократить объявление свойства в литерале объекта. Если название переменной, которая определяет значение свойства, также ключевое слово, то можно опустить ключ:

*let x = 4;  
let y = 1;  
let obj = { x, y }; // = let obj = { x: x, y: y };*

Сокращения значений свойств прекрасно работают с деструктором

*let obj = { x: 4, y: 1 };  
let {x,y} = obj;  
console.log(x); // 4  
console.log(y); // 1*

**Вычисленные ключи свойств**

Существует 2 способа определить ключ для свойства.

* через определенное имя: **obj.foo = true**
* через выражение: **obj['b'+'ar'] = 123**

В литерале объекта вы можете использовать только первый способ в **ES5**. В **ES6** добавили и второй способ

*let propKey = 'foo';  
  let obj = {  
  [propKey]: true,  
  ['b'+'ar']: 123  
};*

Этот новый синтаксис может быть совмещен с объявлением метода

*let obj = {  
  ['h'+'ello']() {  
   return 'hi';  
  }};*console.log(obj.hello()); // hi

**Новые методы для объектов**

**Object.assign(target, source\_1, source\_2, ...)**

Этот метод соединяет **sources** в **target**. Это изменяет **target**, во-первых, копирует все перечислимые собственные свойства **source\_1** в **target**, затем то же самое с **source\_2**. В конце возвращается **target**.

*let obj = { foo: 123 };  
Object.assign(obj, { bar: true });  
console.log(JSON.stringify(obj));  
// {"foo":123,"bar":true}*

**Object.getOwnPropertySymbols(obj)**

В **ES6** ключ свойства может быть либо строкой, либо символом. Теперь есть 5 методов, которые извлекают ключи объекта

* Object.keys(obj) - извлекает все строковые перечисляемые ключи свойств
* Object.getOwnPropertyNames(obj) - извлекает все строковые ключи свойств
* Object.getOwnPropertySymbols(obj) - возвращает все символьные ключи свойств
* Reflect.ownKeys(obj) - извлекает все ключи всех собственных свойств
* Reflect.enumerate(obj) - извлекает все строчные значения ключей всех перечисляемых свойств

**Object.is(value1, value2)**

Строгий оператор эквивалентности рассматривает два значения иначе, чем можно было ожидать

Первое: **NaN** не равен самому себе. Это очень печально, потому что это часто мешает нам обнаружить **NaN**

*[0,NaN,2].indexOf(NaN)  
-1*

Второе: в **javascript** 2 нуля, но строгое сравнение относится к них одинаково

*-0 === +0  
true*

**Object.is()** предоставляет возможность сравнения значений чуть более точно, чем обычное строго сравнение.

*Object.is(NaN, NaN)  
true  
Object.is(-0, +0)  
false*

Все другое сравнивается так же, как и строгое сравнение.

Если мы совместим **Object.is()** с новым **ES6** методом массивов **findIndex()**, то получим **NaN**.

*[0,NaN,2].findIndex(x => Object.is(x, NaN))  
1*

**Object.setPrototypeOf(obj, proto)**

Этот метод устанавливает прототип **proto** для объекта **obj**. В **ES5** мы должны были делать это через **Object.create()**, теперь же мы можем делать это через удобный метод.

[**Таблица с потдержкой(ссылка)**](https://kangax.github.io/compat-table/es6/#chrome50)