ECMAScript 2015

LoftSchool
от мыслителя к создателю

Оглавление

Введение

Для начала следует немного разобраться в терминологии, чтобы избежать путаницы с названиями и не смущаться при виде аббревиатур.

Не уходя глубоко в историю стандартов и спецификаций JavaScript, не будет лишним помнить следующие положения:

- ECMA - ассоциация, деятельность которой посвящена стандартизации информационных и коммуникационных технологий.

- ECMAScript - язык программирования общего назначения. Кросс-платформенный и не зависимый от производителя.

- ECMA-262 - спецификация, описывающая стандарт языка ECMAScript.

- TC39 - комитет, развивающий стандарт ECMAScript и принимающий решения по внедрению нового функционала.

- JavaScript - одна из реализаций спецификации ECMAScript.

Стандарт ECMAScript развивается и по мере этого переживает обновления. На данный момент спецификация ECMA-262 насчитывает 6 версий и 7 опубликованных изданий. Не равное число версий и редакций возникло от того, что было выпущено издание ECMAScript 5.1, которое включает исправления опечаток ECMAScript 5 и не содержит нововведений.

ECMAScript 2015 - это новая версия стандарта ECMAScript. Она была утверждена в июне 2015 года. ES2015 включает в себя значительное обновление языка, которое сильно изменило то, к чему мы привыкли. Предыдущее мажорное обновление было в 2009 году, когда был стандартизирован ES5. Для сравнения, прошлое издание спецификации насчитывает порядка 258 страниц, текущее - 566.

Погружение

Стандарт ES2015 (или ES6) был принят в июне 2015. Пока что большинство браузеров его реализуют и до полного соответствия спецификации может пройти немало времени. Текущее состояние реализации различных возможностей можно посмотреть **здесь**.

Для того, чтобы писать кросс-браузерный код на ES6 прямо сейчас мы будем использовать **Babel.js**.

Babel - это транспайлер, переписывающий код на ES6, в код предыдущего стандарта на ES5, который будет понятен браузерам.

Обычно, Babel включают в состав системы сборки (например, **webpack** или **gulp**) где он будет автоматически переписывать весь код на ES5. Настройка такой конвертации тривиальна, единственное - нужно поднять саму систему сборки, а добавить к ней Babel легко, плагины есть к любой из них. Например, **gulp-babel** для gulp или **babel-loader** для webpack.

Для того, чтобы поэкспериментировать с транспайлером, есть специальная онлайн-**песочница**. Слева вводится код ES2015 (ES6), а справа появляется результат его преобразования в ES5.

Операторы let и const

Известно, что область видимости в JavaScript ограничена функцией. Если вам необходимо реализовать блок со своей областью видимости, то наиболее подходящим способом будет использование **выражения немедленно вызывающейся функции (IIFE)**. Например:

```
var a = 10;

(function() {
  var a = 15;
  console.log(a); // 15
})();

console.log(a);   // 10
```

Оператор let

Переменная, объявленная с помощью оператора let будет видна в рамках блока, в котором объявлена. Таким образом, все, что нам необходимо для создания **блочной области видимости** (block scoping) - пара фигурных скобок { ... }. В частности это влияет на объявления внутри if, while или for.

Например, переменная через var:

```
var a = 10;

if (true) {
  var a = 15;
  console.log(a);  // 15
}

console.log(a); // 15
```

В примере выше a - одна переменная на весь код, которая изменяется в if.

Тоже самое с объявлением через let:

```
let a = 10;

if (true) {
  let a = 15;
  console.log(a);  // 15
}

console.log(a); // 10
```

В этом примере мы имеем две независимые переменные, одна в глобальной области видимости, вторая - в блоке if.

Стоит заметить, что при удалении первой строки с объявлением переменной в глобальной области видимости, второй console.log выдаст ошибку:

```
if (true) {
  let a = 15;
  console.log(a);  // 15
}

console.log(a); // ошибка, нет такой переменной
```

Это происходит потому, что переменная let видна непосредственно в том блоке, в котором объявлена.

Также переменная let видна только после объявления, в отличии от var. Объявление через var использует **всплытие** (hosting).

```
console.log(a) // undefined
var a = 10;
```

Тогда как переменные объявленные через let такого свойства не имеют:

```
console.log(a) // ошибка, нет такой переменной
let a = 10;
```

Также переменные let нельзя повторно объявлять, следующий код вызовет ошибку:

```
let a;
let a; // ошибка, переменная a уже объявлена
```

Еще одной интересной особенностью оператора let является то, что переменная объявленная в цикле - своя на каждой итерации:

```
for (let i = 0; i < 10; i++) { /* ... */ }

console.log(i); // ошибка, нет такой переменной
```

При объявлении внутри цикла , переменная i будет видна только в блоке цикла и на каждой итерации она будет своя. Также она не видна снаружи, поэтому будет ошибка в console.log.

Тогда как var - одна на все итерации цикла и видна даже после цикла:

```
for (var i = 0; i < 10; i++) { /* ... */ }

console.log(i); // 10
```

Оператор const

Оператор const создает **константу** - переменную, которую нельзя изменить.

```
const a = 10;
a = 15 // ошибка
```

В остальном объявление const полностью аналогично let.

Важно помнить, что если в константу присвоен объект, то его поля как и прежде можно менять, защищена от изменений сама константа.

```
const obj = {
  a: 10
};

obj.a = 15; // допустимо
obj = 10; // ошибка

const arr = [];

arr.push('1'); // допустимо
arr = 10; // ошибка
```

Параметры функций по умолчанию

В JavaScript стандартное значение параметров функций - undefined.

```
function fn(param) {
  console.log(param);
}

fn() // undefined
```

Однако, в некоторых случаях, полезно задать иное значение по умолчанию. Для этого можно использовать оператор ’||’

```
function fn(param) {
  param = param || 'default';
  console.log(param);
}

fn() // 'default'
```

ES6 предлагает нам новый способ задания пареметров по умолчанию:

```
function fn(param = 'default') {
  console.log(param);
}

fn() // 'default'
```

Параметр по умолчанию используется при отсутствии аргумента или переданном undefined. При передаче любого значения, кроме undefined, включая пестую строку, ноль или null, параметр считается переданным, и и значение по умолчанию не используется.

Стрелочные функции (arrow function)

В ES6 появился новый синтаксис задания функций:

```
let sum = (a, b) => a + b;

sum(5, 5); // 10
```

Эта запись примерно эквивалентна следующей:

```
let sum = function(a, b) {
  return a + b;
}

sum(5, 5); // 10
```

Таким образом, слева от => находятся аргументы функции, справа - выражение, результат которого будет возвращен.

Если аргумент один, то оборачивать его в скобки не обязательно:

```
let increment = a => a + 1;

increment(1); // 2
```

Но если необходимо задать функцию без аргументов, то используются пустые скобки:

```
let greting = () => 'Hello!';

greting(); // 'Hello!'
```

Если тело функции достаточно большое, то его можно обернуть в фигурные скобки:

```
let getOdd = arr => {
  let result = [];

  for (let i = 0; i < arr.length; i++) {
   if (arr[i] % 2) {
   result.push(arr[i]);
   }
  }

  return result;
};
getOdd([1, 2, 3, 4, 5]); // [1, 3, 5]
```

Следует знать, что как только тело функции оборачивается в фигурные скобки, то ее результат уже не возвращается автоматически. Если необходимо что-то возвратить, то такая функция должна делать явный return.

Такие функции очень удобны в качестве колбеков:

```
let arr = [1, 2, 3, 4, 5];
let odd = arr.filter(item => item % 2);

console.log(odd); // [1, 3, 5]
```

Нужно помнить, что функции-стрелки нельзя вызывать с оператором new. Так как такие функции не имеют своего this, то и использовать их как конструкторы нельзя.

Если нам нужно чтобы функция возвратила объект, необходимо обернуть его в круглые скобки:

```
var getObj = () => ({name: 'some name'});
getObj(); // {name: 'some name'};
```

Объект оборачивается круглыми скобками для того, чтобы интерпретатор понял, что это именно объект, а не тело функции.

Важным отличием стрелочных функций от обычных является то, что функции-стрелки не имеют своего this. Внутри функции-стрелки используется тот же контекст что и снаружи. Или стрелочный функции наследуют родительский this.

Это очень удобно использовать в обработчиках событий и колбеках:

```
let user = {
  name: 'user name',
  skills: ['js', 'php', 'knitting'],

  showSkills: function() {
    this.skills.forEach(skill => {
      console.log(this.name + ': ' + skill);
    });
  }
};

user.showSkills();  // user name: js
                    // user name: php
                    // user name: knitting
```

В метод forEach была передана функция-стрелка, поэтому this.name тот же самый что и во внешней функции showSkills.

Еще одно отличие стрелочных функций заключается в отсутствии у них своего arguments:

```
let empty = () => arguments;

console.log(empty()); // ошибка. arguments не
определен
```

В следующем примере arguments берется от внешней функции:

```
function fn() {
  let showArgs = () => arguments;
  console.log(showArgs());
}

fn(1, 2, 3); // [1, 2, 3]
```

Блочная видимость функций

Объявление функции Function Declaration, сделанное в блоке, доступно только в этом блоке.

```
{
  fn(); // 'Hello!'

  function fn() {
   console.log('Hello!');
  }
}

fn(); // ошибка. функция не существует
```

Деструктуризация

Деструктуризация (destructuring assignment) - это особый синтаксис присваивания, при котором можно присвоить массив или объект сразу нескольким переменным, разбив его на части.

Посмотрим на пример деструктуризации массива:

```
let [foo, bar] = [1, 2];

console.log(foo); // 1
console.log(bar); // 2
```

При таком присваивании первое значение массива пойдет в первую переменную foo, второе - в bar, а последующие, если есть, будут отброшены.

Ненужные элементы массива можно отбросить, поставив запятую:

```
let [, , foo, bar] = [1, 2, 3, 4, 5, 6];

console.log(foo); // 3
console.log(bar); // 4
```

В коде выше первый и второй элементы массива были отброшены, они ни куда не записались. Как, впрочем, и все элементы после третьего.

Значения по умолчанию

Если значений в массиве меньше чем переменных, то просто присвоится undefined:

```
let [foo, bar, baz] = [1, 2];

console.log(foo); // 1
console.log(bar); // 2
console.log(baz); // undefined
```

Используя деструктуризацию, можно задать значение по умолчанию:

```
let [foo, bar=500, baz=3] = [1, 2];

console.log(foo); // 1
console.log(bar); // 2
console.log(baz); // 3
```

В переменную foo значение попало из массива. Переменной bar задано значение по умолчанию и так как в массиве оно присутствует, присвоен был элемент массива. Переменной baz не хватило элементов в массиве и было использовано значение по умолчанию.

В качестве значений по умолчанию можно использовать не только примитивные типы данных, но и выражения, включающие вызовы функций.

```
let getBar = () => 2;
let [foo, bar=getBar()] = [1];

console.log(foo); // 1
console.log(bar); // 2
```

Деструктуризация объекта

Давайте взглянем на синтаксис деструктуризации объектов:

```
let obj = { foo: 1, bar: 2 };
let {foo, bar} = obj;

console.log(foo); // 1
console.log(bar); // 2
```

Т.е. объект справа - уже существующий, который мы хотим разбить на переменные. Слева - список переменных, в которые нужно записать свойства. Как видно, свойства obj.foo и obj.bar автоматически присвоились соответствующим переменным.

Если нужно присвоить свойство объекта в переменную с другим именем, то можно указать соответствие через двоеточие, например:

```
let obj = { foo: 1, bar: 2 };
let {foo: f, bar} = obj;

console.log(f);   // 1
console.log(bar); // 2
```

В примере выше свойство obj.foo записалось в переменную f, а свойство obj.bar в переменную с тем же именем.

Также можно использовать значения по умолчанию:

```
let obj = { foo: 1 };
let [foo, bar=2, baz=3] = obj;

console.log(foo); // 1
console.log(bar); // 2
```

```
console.log(baz); // 3
```

При необходимости имеется возможность сочетать одновременно двоеточие и равенство:

```
let obj = { foo: 1 };
let [foo, bar:b=2] = obj;

console.log(foo); // 1
console.log(b);   // 2
```

Вложенные деструктуризации

Если объект или массив содержат вложенные объекты или массивы, и их тоже нужно разбить это возможно, т.к. деструктуризации можно сочетать и вкладывать друг в друга как угодно.

```
let obj = {
 foo: 1,
 bar: {
   bar1: 2,
   bar2: 3
 },
 baz: [4, 5]
};

let { foo, bar: {bar1, bar2}, baz: [baz1, baz2] } =
obj;

console.log(foo);  // 1
console.log(bar1); // 2
console.log(bar2); // 3
console.log(baz1); // 4
console.log(baz2); // 5
```

Как видно, весь объект корректно разбит на переменные.

Оператор расширения (spread)

Оператор расширения позволяет расширить выражения в тех местах, где предусмотрено использование нескольких аргументов (при вызовах функций) или ожидается несколько элементов (для массивов).

Давайте посмотрим несколько распространенных случаев использования оператора расширения.

В следующем примере мы сконкатенируем два массива при помощи оператора spread:

```
let foo = [3, 4];
let bar = [1, 2, ...foo];

console.log(bar); // [1, 2, 3, 4];
```

Как мы видим, элементы массива foo переместились в конец массива bar. Но можно вставить элементы массива bar в любое место:

```
let foo = [3, 4];
let bar = [1, ...foo, 2];

console.log(bar); // [1, 3, 4, 2];
```

Очень широко используется прием передачи параметров в функцию при помощи метода apply:

```
function fn(a, b, c) { console.log(a, b, c); }

let params = [1, 2, 3];

fn.apply(null, params); // 1 2 3
```

При помощи ES6 можно использовать следующий эквивалент:

```
function fn(a, b, c) { console.log(a, b, c); }

let params = [1, 2, 3];

fn(...params); // 1 2 3
```

Оператор можно использовать для любого аргумента, в том числе несколько раз:

```
function fn(a, b, c, d, e) {
console.log(a, b, c, d, e); }

let params = [2, 3];

fn(1, ...params, 4, …[5]); // 1 2 3 4 5
```

В ES5 невозможно выполнить комбинацию new и apply т.к. apply выполняет вызов метода, а не конструктора. В ES6 оператор расширения поддерживает это:

```
function Fn(a, b) { console.log(a, b); }

Fn(...[1, 2]); // 1 2
```

Оператор spread также дает нам возможность удобнее работать с методом push. Например, мы хотим добавить элементы одного массива в конец другого. В ES5 мы бы сделали что-то подобное:

```
let arr1 = [1, 2];
let arr2 = [3, 4];

Array.prototype.push.apply(arr1, arr2);

console.log(arr1); // 1, 2, 3, 4
```

Теперь доступна такая возможность:

```
let arr1 = [1, 2];
let arr2 = [3, 4];

arr1.push(...arr2);

console.log(arr1); // 1, 2, 3, 4
```

Шаблонные строки

Шаблонные строки дают нам возможность использования многострочных литералов, т.е. в них разрешен перенос строк. Для использования таких строк текст необходимо обернуть в обратные кавычки:

```
console.log('some
             long
             string');
```

Пробелы и переводы строк также будут выведены.

В обычных строках необходимо использовать следующий синтаксис:

```
console.log('some\n\
             long\n\
             string');
```

Используя шаблонные строки, нам доступна интерполяция или вставка выражения. Такие строки могут содержать выражения, которые обозначаются знаком доллара и фигурными скобками ${ ... }.

Интерполяция имеет следующий синтаксис:

```
let a = 10;
console.log('messages: ${a}'); // 'messages: 10'

let b = 15;
console.log('messages: ${a + b}'); // 'messages:
25'

let getNumber = () => 30;
console.log('messages: ${getNumber()}'); //
'messages: 30'
```

При помощи ${...} можно вставлять как значения переменных, так и более сложные выражения, которые могут включать в себя операторы, вызовы функций и т.п.

Функции шаблонизации

Для шаблонизации строк можно использовать свои функции. Ее первый аргумент будет содержать массив строк. Второй и последующие содержат значения вычисленных выражений:

```
let a = 5;
let b = 10;

function fn(strings, ...values) {
  console.log(strings[0]); // 'sum: '
  console.log(strings[1]); // 'division: '
  console.log(values[0]);  // 15
  console.log(values[1]);  // 2
}

fn'sum: ${ a + b } division: ${ b / a }';
```

В примере выше видно что строка разбивается как: “кусок строки” - “параметр” - “кусок строки” - “параметр”.

Функция шаблонизации должна как-то пребразовать строку и вернуть результат. В простейшем случае можно просто объединить полученные фрагменты в строку:

```
function fn(strings, ...values) {
  let str = '';

  for(let i = 0; i < values.length; i++) {
   str += strings[i];
   str += values[i];
  }

  str += strings[strings.length - 1];
  return str;
}

let a = 5;
let b = 10;
let result = fn'sum: ${ a + b } division: ${ b /
a }';

console.log(result); // sum: 15 division: 2
```

Таким образом, можно добиться интересных результатов. Стоит сказать, что функция шаблонизации не обязательно должна возвращать именно строку.

Объекты

В ES6 имеются ряд нововведения, касающиеся улучшения записи свойств и методов объектов.

Определение свойств

Бывают случаи, когда у нас имеется некоторая переменная, которую мы хотим записать в объект под тем же самым именем:

```
let name = 'Some name';
let isAdmin = true;

var user = {
  name: name,
  isAdmin: isAdmin
};
```

Теперь это можно сделать несколько проще:

```
let name = 'Some name';
let isAdmin = true;

var user = {
  name,
  isAdmin
};
```

В этом случае, при объявлении объекта, достаточно указать только имя свойства, а значение будет взять их переменной с аналогичным именем.

Вычисляемые свойства

Короткий синтаксис также поддерживает использование выражений, например:

```
let property = 'name';

var user = {
  [property]: 'Some name'
};

console.log(user.name); // Some name
```

Или более сложное выражение:

```
let fn = () => 'foo ';

var user = {
  [fn() + 2]: 'Some name'
};

console.log(user['foo 2']);  // Some name
```

Определение методов

Начиная с ECMAScript 6, существует короткий синтаксис для определения методов объектов. По сути, это сокращение для функции, которая назначена методом.

```
let obj = {
  getFoo() { return 'foo'; },
  getBar() { return 'bar'; }
};

console.log(obj.foo()); // 'foo'
console.log(obj.bar()); // 'bar'
```

Важно знать, что определенные таким способом методы, нельзя использовать в качестве конструктора:

```
let obj = {
  bar() { console.log('bar'); }
};

new obj.bar(); // ошибка
```

Новые методы объектов

В стандарте ES2015 доступны новые методы объектов. Object.is() и Object.assign().

Метод Object.is() определяет, являются ли два переданных значения одинаковыми.

```
Object.is('foo', 'foo'); // true
Object.is([], [])        // false
Object.is(null, null)    // true
```

Поведение этого метода не аналогично оператору строгого равенства ===.
Имеется ряд специальных случаев:

```
Object.is(-0, +0);       // false
Objec.is(NaN, NaN)       // true
```

Тогда как строгое равенство даст иной результат:

```
console.log(-0 === +0);   // true
console.log(NaN === NaN) // false;
```

Метод Object.assign() используется для копирования значений объектов. Он получает список объектов и копирует в первый свойства из остальных:

```
let obj1 = { a: 1 };
let obj2 = { b: 2 };
Object.assign(obj1, obj2);

console.log(obj1); // { a: 1, b: 2 }
```

Этот метод можно использовать для одноуровнего клонирования объектов:

```
let o1 = { a: 1 };
let o2 = { b: 2 };
let obj = Object.assign({}, o1, o2);

console.log(obj); // { a: 1, b: 2 }
```

Классы

Классы, появившиеся в ECMAScript2015, представляют собой синтаксический сахар для существующего в языке прототипного наследования. Таким образом классы не вводят новую модель наследования, но они предоставляют более понятный и простой способ создания объектов.

Классы - это просто функции, потому так же как мы определяем функции (function expression, function declaration), классы можно определять как class expression и class declaration. Класс имеет специальный метод constructor, необходимый для создания и инициализации объектов. В классе может быть только один конструктор.

Объявление класса

Первый способ определения класса - объявление класса (class declaration). Для этого необходимо воспользоваться ключевым словом class и указать имя класса.

```
class User() {
  consturctor(name) {
    this.name = name;
  }
}
```

Разница между объявлением функции (function declaration) и объявлением класса (class declaration) в том, что объявление функции совершает подъем (hoisted), в то время как объявление класса - нет. Поэтому вначале необходимо объявить наш класс и только затем работать с ним:

```
let foo = new Foo(); // Ошибка. Класс не определен

class Foo{};
```

Выражение класса

Второй способ определения классов - выражение класса (class expression). С помощью него можно задать именованные и безымянные классы.

```
// безымянный
let User = class {
  constructor(name) {
    this.name = name;
  }
};

// именованный
let User = class User {
  constructor(name) {
    this.name = name;
  }
};
```

Геттеры, сеттеры и вычисляемые свойства

В классах, как и в обычных объектах, можно объявить геттеры и сеттеры через get/set, а также использовать вычисляемые свойства.

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

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

```
  set fullName(newName) {
    [this.firstName this.lastName] =
newName.split(' ');
  }

  ['SAY'.toLowerCase()](message) {
   console.log(message);
  }
}

let foo = new Foo('bar', 'baz');
console.log(foo.fullName)   // bar baz
foo.fullName = 'baz quux';
console.log(foo.fullName); // baz quux
foo.say('I\'m alive!!!')   // I'm alive!!!
```

При чтении fullName будет вызван метод get fullName(), при присвоении - метод set fullName с новым значением.

Стоит заметить, что синтаксис создания класса позволяет определить метод класса, но не значение. Предполагается, что в прототипе должны быть только методы.

Статические методы

Класс, как и функция, является объектом. Статические методы класса, это методы самого класса и не могут быть вызваны у экземпляров. Ключевое слово static определяет для класса статический метод.

```
class Foo {
  constructor(name) {
    this.name = name;
  }

  static getVersion() {
    return '0.0.1';
  }
}
```

```
Foo.getVersion() // 0.0.1
```

Как правило, статические методы часто используются для созданий различных служебных функций.

Сравнение синтаксиса определения классов

Так как ES6 классы являются синтаксическим сахаром для более комфортного создания объектов, то следующие определения классов Foo и Bar примерно аналогичны:

```
class Foo {
  constructor(name) {
    this.name = name;
  }

  showName() {
    console.log(this.name);
  }

  static staticMethod() {}
};

var Bar = function(name) {
  this.name = name;
}

Bar.staticMethod = function() {};

Bar.prototype.showName = function() {
  console.log(this.name);
};
```

Наследование

Ключевое слово extends используется в объявлениях классов и выражениях классов для создания класса дочернего относительно другого класса. В примере ниже объявлено два класса: Foo и наследующего от него Bar:

```
class Foo {
  constructor(name) {
    this.name = name;
  }

  showName() {
    console.log(this.name);
  }
}

class Bar extends Foo {
  showName() {
    console.log('I\'m');
    super.showName();
  }
}

let bar = new Bar('bar');
bar.showName();  // I'm
                 // bar;
```

Как мы видим, в классе Bar достуаны как свои методы, так и методы родителя (через super). Т.е. методы родиля Foo можно переопределить в наследние. При этом для обращения к родительскому методу используется super.showName().

С методом constructor немного иначе. Конструктор родителя наследуется автоматически. Если в потомке не указан свой constructor, то используется родительский метод. В примере выше, таким образом Foo использует конструктор от Bar.

Если же у потомка свой constructor, то, чтобы в нем вызвать конструктор родителя используется метод super() с аргументами:

```
class Foo {
  constructor(name) {
    this.name = name;
  }

  showName() {
    console.log(this.name);
  }
}

class Bar extends Foo {
  constructor(name) {
    super(name);
  }

  showName() {
    console.log('I\'m');
    super.showName();
  }
}
```

Ряд ограничений состоит в том, что метод super() можно вызвать только из конструктора потомка. Нельзя вызвать super() из произвольного класса. Также в конструкторе мы обязаны вызвать super() до обращения к this т.к. до вызова метода super - this не существует.