

Знакомство с ES6

ES6

- let&const
- Деструктуризация (массивов и объектов)
- ...Spread-оператор
- Стрелочные функции
- Шаблонные строки
- ООП. Классы. Наследование.

Классы

Объявление класса

```
1. class Название [extends Родитель] {  
2.     constructor  
3.     методы  
4. }
```

Объявление класса

```
1. class Polygon {  
2.     constructor(height, width) {  
3.         this.height = height;  
4.         this.width = width;  
5.     }  
6. }
```

Выражение класса

```
1. //БЕЗЫМЯННЫЙ
2. var Polygon = class {
3.     constructor(height, width) {
4.         this.height = height;
5.         this.width = width;
6.     }
7. };
8. //ИМЕНОВАННЫЙ
9. var Polygon = class Polygon {
10.    constructor(height, width) {
11.        this.height = height;
12.        this.width = width;
13.    }
14.};
```

Создание объекта и прототип

Constructor запускается при создании `new Object`, остальные методы записываются в `Object.prototype`

```
1. class User {  
2.     constructor(name) {  
3.         this.name = name;  
4.     }  
5.     sayHi() { console.log(this.name);}  
6. }  
7. let user = new User("Вася");
```

Создание объекта и прототип

```
1. function User(name) {  
2.     this.name = name;  
3. }  
4. User.prototype.sayHi = function() {  
5.     console.log(this.name);  
6. };
```


Всплытие (hoisting)

Разница между объявлением функции (function declaration) и объявлением класса (class declaration) в том, что объявление функции совершает подъём (hoisted), в то время как объявление класса — нет

1. `var p = new Polygon();`
2. `class Polygon {}`
3. `//Uncaught ReferenceError: Polygon is not defined`

Статические методы

```
1. class Point {
2.     constructor(x, y) {
3.         this.x = x;
4.         this.y = y;
5.     }
6.     static distance(a, b) {
7.         const dx = a.x - b.x;
8.         const dy = a.y - b.y;
9.         return Math.sqrt(dx*dx + dy*dy);
10.    }
11. }
12. const p1 = new Point(5, 5);
13. const p2 = new Point(10, 10);
14. console.log(Point.distance(p1, p2)); //7.07....
```

Геттеры, сеттеры

```
1. class User {
2.     constructor(firstName, lastName) {
3.         this.firstName = firstName;
4.         this.lastName = lastName;
5.     }
6.     get fullName() {
7.         return `${this.firstName} ${this.lastName}`;
8.     }
9.     set fullName(newValue) {
0.         [this.firstName, this.lastName] = newValue.split(' ');
1.     }
2. };
3. let user = new User('Maksim', 'Hladki');
4. console.log(user.fullName); //Maksim Hladki
5. user.fullName = "Ivan Ivanov";
6. console.log(user.fullName); //Ivan Ivanov
```

Пример

```
1. class Rectangle {
2.     constructor (width, height) {
3.         this._width = width
4.         this._height = height
5.     }
6.     set width (width) { this._width = width }
7.     get width () { return this._width }
8.     set height (height) { this._height = height }
9.     get height () { return this._height }
0.     get area () { return this._width * this._height }
1. }
2. let test = new Rectangle(50, 20);
3. console.log(test.area);//1000
```

Вычисляемые имена методов

```
1. class Foo() {  
2.     myMethod() {}  
3. }
```

```
1. class Foo() {  
2.     ['my'+'Method']() {}  
3. }
```

```
1. const m = 'myMethod';  
2. class Foo() {  
3.     [m]() {}  
4. }
```

Наследование

1. //Только один конструктор, прототип, базовый класс!
- 2.
3. class Child extends Parent {
- 4.
5. //TODO logic
- 6.
7. }

Пример

```
1. class Point {
2.     constructor(x, y) {
3.         this.x = x; this.y = y;
4.     }
5.     toString() {
6.         return '(' + this.x + ', ' + this.y + ')';
7.     }
8. }
9. class ColorPoint extends Point {
0.     constructor(color) {
1.         super(0, 0);
2.         this.color = color;
3.     }
4.     toString() {
5.         return super.toString() + ' in ' + this.color;
6.     }
7. }
8. let cPoint = new ColorPoint('red');
9. console.log(cPoint.toString()); //(0, 0) in red
```

Наследование статических методов

```
1. class Foo {  
2.     static classMethod() {  
3.         return 'hello';  
4.     }  
5. }  
6. class Bar extends Foo {  
7.     //TODO  
8. }  
9.  
0. console.log(Bar.classMethod());//hello
```


Super

1. //Используется для вызова функций, принадлежащих
2. //родителю объекта
- 3.
4. `super([arguments]);`//вызов родительского конструктора
- 5.
6. `super.functionOnParent([arguments]);`

Пример: ВЫЗОВ КОНСТРУКТОРА

```
1. class Polygon {
2.     constructor(height, width) {
3.         this.height = height;
4.         this.width = width;
5.     }
6. }
7. class Square extends Polygon {
8.     constructor(length) {
9.         super(length, length);
0.     }
1.     get area() {
2.         return this.height * this.width;
3.     }
4. }

5. Let square = new Square(10);
```

ВЫЗОВ МЕТОДА

```
1. class Foo {
2.     static classMethod() {
3.         return 'hello';
4.     }
5. }
6. class Bar extends Foo {
7.     static classMethod() {
8.         return super.classMethod() + ', too';
9.     }
0. }
1. Bar.classMethod();//hello, too
```

Mixins

Mixins

```
1. // Абстрактные подклассы (mix-ins) — это шаблоны для классов.
2. //У класса может быть только один родительский класс, поэтому
3. // множественное наследование невозможно.
4. //Функциональность должен предоставлять родительский класс
5.
6. class B extends A, M {}//      Uncaught SyntaxError: Unexpected token ,
7.           //      множественного наследования нет

8. const mixin = base => class extends base {
9.     /* свойства и методы */
0. }
```

Пример

```
1. class Person { ... }
2. const Storage = Sup => class extends Sup {
3.     save(database) { ... }
4. };
5. const Validation = Sup => class extends Sup {
6.     validate(schema) { ... }
7. };
8. class Employee extends Storage(Validation(Person)) { ... }
```

Пример

```
1. let MyMixin = (superclass) => class extends superclass {
2.   test() {
3.     console.log('test from MyMixin');
4.   }
5. };
6.
7. class MyClass extends MyMixin(MyBaseClass) {
8.   /* ... */
9. }
0. let c = new MyClass();
1. c.test();//test from MyMixin
```

Symbol

Тип данных Symbol

Уникальный и неизменяемый тип данных, который может быть использован как идентификатор для свойств объектов

Символьный объект — это объект-обертка для примитивного символьного типа

```
let sym = Symbol("foo");  
  
console.log(typeof sym);//symbol  
  
let symObj= Object(sym);  
  
console.log(typeof symObj);//object  
  
console.log(Symbol("name") == Symbol("name"));//false
```

Тип данных Symbol

```
1. let isAdmin = Symbol("isAdmin");  
  
2. let user = {  
  
3.   name: "Вася",  
  
4.   [isAdmin]: true  
  
5. };  
  
6. console.log(user[isAdmin]); //true
```

Тип данных Symbol

Свойство, объявленное через символ, не будет видно в `for-in`, `Object.keys`, `Object.getOwnPropertyNames`, также не будет добавлено при использовании `JSON.stringify`

Тип данных Symbol

```
1. let user = {
2.   name: "Вася",
3.   age: 30,
4.   [Symbol.for("isAdmin")]: true
5. };
6.
7. //в цикле for..in не будет символа
8.
9. console.log(Object.keys(user));//[ "name", "age" ]
0.
1. //доступ к свойству через ГЛОБАЛЬНЫЙ СИМВОЛ — работает
2.
3. console.log(user[Symbol.for("isAdmin")]);//true
```

Глобальные символы

Глобальный реестр символов позволяет иметь общие глобальные символы, которые можно получить из реестра по имени.

Используется метод `for`

```
//создание символа в реестре
```

```
let name = Symbol.for("name");
```

```
//символ уже есть, чтение из реестра
```

```
console.log(Symbol.for("name") == name);//true
```

Встроенные символы (Well-known)

Symbol.iterator

возвращающий итератор для объекта

Symbol.match

сопоставление объекта со строкой
(String.prototype.match)

Symbol.replace

заменяет совпавшие подстроки в строке
(String.prototype.replace)

Symbol.search

возвращает индекс вхождения подстроки,
соответствующей регулярному выражению
(String.prototype.search)

Symbol.split

разбивает строку на части в местах,
соответствующих регулярному выражению
(String.prototype.split)

Symbol.for(key)

ищет существующие символы по заданному
ключу и возвращает его, если он найден

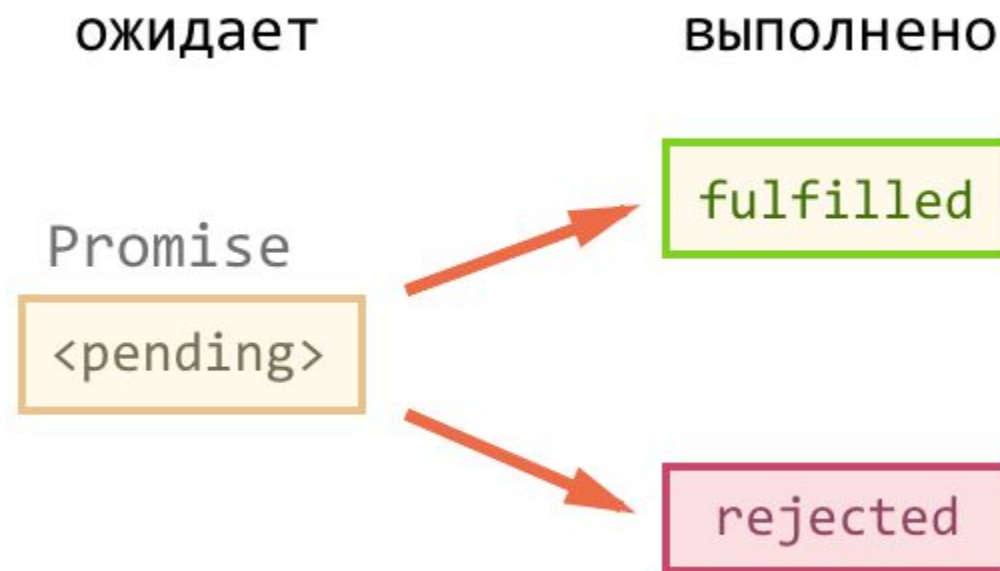
Symbol.species

определяет конструктор для порожденных
объектов

Promise

Понятие промиса

**Объект, используемый как заглушка для результата
некоего отложенного (и асинхронного) вычисления**



callback hell

```
1.  getData(function(a){
2.    getMoreData(a, function(b){
3.      getMoreData(b, function(c){
4.        getMoreData(c, function(d){
5.          getMoreData(d, function(e){
6.            ...
7.          });
8.        });
9.      });
10.    });
11.  });
```

Синтаксис

```
new Promise(executor);
```

```
new Promise(function(resolve, reject) { ... });
```

```
// onFulfilled – срабатывают, когда promise в состоянии  
«выполнен успешно»
```

```
// onRejected – срабатывают, когда promise в состоянии  
«выполнен с ошибкой»
```

Алгоритм

Код, которому надо сделать что-то асинхронно, создаёт объект `promise` и возвращает его

Внешний код, получив `promise`, навешивает на него обработчики

После завершения процесса асинхронный код переводит `promise` в состояние `fulfilled` (с результатом) или `rejected` (с ошибкой). При этом автоматически вызываются соответствующие обработчики во внешнем коде

Навешивание обработчиков

1. `promise.then(onFulfilled, onRejected);`
2. `//onFulfilled` – функция, которая будет вызвана с результатом при `resolve`
3. `//onRejected` – функция, которая будет вызвана с ошибкой при `reject`

1. `//onFulfilled` сработает при успешном выполнении
2. `promise.then(onFulfilled);`
3. `//onRejected` сработает при ошибке
4. `promise.then(null, onRejected);`

Метод then

Метод возвращает Promise и принимает два аргумента, callback-функции для случаев выполнения и отказа
соответственно

```
1. p.then(onFulfilled, onRejected);
2. p.then(
3.   function(value) {
4.     //выполнение
5.   },
6.   function(reason) {
7.     //отказ
8.   }
9. );
```

Метод catch

Для того, чтобы поставить обработчик только на ошибку, вместо `.then(null, onRejected)` можно написать `.catch(onRejected)`

1. `promise.catch(onRejected);`
2. `promise.catch(function(reason) {`
3. `//отказ`
4. `});`
5. `// catch всегда возвращает положительный промис`

throw в промисах

Синхронный throw – то же самое, что reject

1. `let p = new Promise((resolve, reject) => {`
2. `//то же что reject(new Error("Ошибка"))`
3. `throw new Error("Ошибка");`
4. `});`
5. `p.catch(console.log);//Ошибка`

Эмуляция асинхронности

```
1. let promise = new Promise((resolve, reject) => {
2.     setTimeout(() => {
3.         resolve("result");
4.     }, 1000);
5. });
6. promise.then(
7.     result => {
8.         console.log("Fulfilled: " + result);
9.         //result - аргумент resolve
10.     },
11.     error => {
12.         console.log("Rejected: " + error);
13.         //error - аргумент reject
14.     }
15. );
16. //Fulfilled: result
```


Еще пример

```
1. var promise = new Promise(function(resolve, reject) {
2.     console.log('in Promise constructor function');
3.     setTimeout(function() {
4.         console.log('in setTimeout callback');
5.         resolve('foo');
6.     }, 100);
7. });
8. console.log('created promise');
9. promise.then(function(result) {
10.    console.log('promise returned: ' + result);
11. });
12. console.log('hooked promise.then()');
13. //in Promise constructor function
14. //created promise, hooked promise.then()
15. //in setTimeout callback
16. //promise returned: foo
```

Promise после reject/resolve – неизменны

```
1. let promise = new Promise((resolve, reject) => {
2.   // через 1 секунду готов результат: result
3.   setTimeout(() => resolve("result"), 1000);
4.   // через 2 секунды — reject с ошибкой, он будет
   проигнорирован
5.   setTimeout(() => reject(new Error("ignored")), 2000);
6. });
7. promise.then(
8.   result => console.log("Fulfilled: " + result), //сработает
9.   error => console.log("Rejected: " + error) //не сработает
10. );
11. //Fulfilled: result
```

Промисификация

Процесс, при котором из асинхронной функции делают обертку, возвращающую promise

```
1. function httpGet(url) {
2.     return new Promise(function(resolve, reject) {
3.         var xhr = new XMLHttpRequest();
4.         xhr.open('GET', url, true);
5.         xhr.onload = function() {
6.             if (this.status == 200) {
7.                 resolve(this.response);
8.             }
9.             else { var error = new Error(this.statusText);
10.                 error.code = this.status;
11.                 reject(error); } };
12.         xhr.onerror = function() {
13.             reject(new Error("Network Error")); };
14.         xhr.send(); });}
```

Метод Promise.resolve

Метод возвращает Promise, выполненный с переданным значением. Если переданное значение является thenable (имеет метод then), возвращаемое обещание будет следовать thenable - объекту, принимая свое состояние; в ином случае возвращаемое обещание будет выполнено с переданным значением

- `Promise.resolve(value);`
- `Promise.resolve(promise);`
- `Promise.resolve(thenable);`

Пример 1

```
1. Promise.resolve("Success").then(  
2.     function(value) {  
3.         console.log(value); //"Success"  
4.     },  
5.     function(value) {  
6.         //не будет вызвана  
7.     }  
8. );
```

Пример 2

```
1. var p = Promise.resolve([1,2,3]);  
2. p.then(function(v) {  
3.     console.log(v[0]);//1  
4. });
```

Пример 3

```
1. var original = Promise.resolve(true);
2. console.log(original); //Promise {resolved: true}
3. var cast = Promise.resolve(original);
4. cast.then(function(v) {
5.     console.log(v); //true
6. });
```

Метод Promise.reject

1. // Метод возвращает объект Promise,
2. //который был отклонен по указанной причине
3. `Promise.reject(reason);`

Метод Promise.all

Возвращает обещание, которое выполнится тогда, когда будут выполнены все обещания, переданные в виде перечисляемого аргумента, или отклонено любое из переданных обещаний

1. `Promise.all(iterable);`
2. `//iterable` - перечисляемый объект, например, массив (`Array`)

Пример 1

```
1. var p1 = Promise.resolve(3);
2. var p2 = 1337;
3. var p3 = new Promise((resolve, reject) => {
4.   setTimeout(resolve, 100, "foo");
5. });
6. Promise.all([p1, p2, p3]).then(values => {
7.   console.log(values);
8. });
9. //[3, 1337, "foo"]
```

Пример 2

```
1.  var p1 = Promise.resolve(3);
2.  var p2 = Promise.reject("Promise Reject");
3.  var p3 = new Promise((resolve, reject) => {
4.    setTimeout(resolve, 100, "foo");
5.  });
6.  Promise.all([p1, p2, p3]).then(
7.    values => console.log(values),
8.    error => console.log(error)
9.  );
10. //Promise Reject
```

Метод Promise.race

Метод возвращает выполненное или отклоненное обещание, в зависимости от того, с каким результатом завершится первое из переданных обещаний, со значением или причиной отклонения этого обещания

1. `Promise.race(iterable);`
2. `//iterable` - итерируемый объект, такой как (`Array`)

Пример 1

```
1.  var p1 = new Promise(function(resolve, reject) {
2.      setTimeout(resolve, 500, "один");
3.  });
4.  var p2 = new Promise(function(resolve, reject) {
5.      setTimeout(resolve, 100, "два");
6.  });
7.  Promise.race([p1, p2]).then(function(value) {
8.      console.log(value); //"два"
9.      //Оба вернули resolve, однако p2 вернул результат
      первым
10. });
```

Пример 2

```
1.  var p1 = Promise.resolve(3);
2.  var p2 = Promise.reject("Promise Reject");
3.  var p3 = new Promise((resolve, reject) => {
4.    setTimeout(resolve, 100, "foo");
5.  });
6.  Promise.race([p1, p2, p3]).then(
7.    values => console.log(values),
8.    error => console.log(error)
9.  );
10. //3
```

Пример 3

```
1. var p3 = new Promise(function(resolve, reject) {
2.   setTimeout(resolve, 100, "три");
3. });
4. var p4 = new Promise(function(resolve, reject) {
5.   setTimeout(reject, 500, "четыре");
6. });
7. Promise.race([p3, p4]).then(function(value) {
8.   console.log(value); //"три"
9.   //p3 быстрее, поэтому выведется значение его resolve
10. }, function(reason) {
11.   //Не вызывается
12. });
```

Пример 4

```
1. var p5 = new Promise(function(resolve, reject) {
2.   setTimeout(resolve, 500, "пять");
3. });
4. var p6 = new Promise(function(resolve, reject) {
5.   setTimeout(reject, 100, "шесть");
6. });
7. Promise.race([p5, p6]).then(
8.   function(value) {
9.     //Не вызывается
10.  },
11.  function(reason) {
12.    console.log(reason); //"шесть"
13.    //p6 быстрее, выводится его rejects
14.  }
15. );
```


Цепочки промисов

```
something(...)
```

```
.then(...)
```

```
.then(...)
```

```
.then(...);
```

Если очередной `then` вернул промис, то далее по цепочке будет передан не сам этот промис, а его результат

Пример 1

```
1. Promise.resolve("Success")
2.   .then(x => { console.log("THEN1", x); })
3.   .catch(x => { console.log("CATCH1", x); })
4.   .then(x => { console.log("THEN2", x); })
5.   .catch(x => { console.log("CATCH2", x); });
6. //THEN1 Success
7. //THEN2 undefined
```

Пример 2

```
1. Promise.reject("Fail")
2.   .then(x => { console.log("THEN1", x); })
3.   .catch(x => { console.log("CATCH1", x); })
4.   .then(x => { console.log("THEN2", x); })
5.   .catch(x => { console.log("CATCH2", x); });
6. //CATCH1 Fail
7. //THEN2 undefined
```

Пример 3

```
Promise.reject("Fail")
  .catch(x => {
    console.log("CATCH1", x);
    return Promise.reject("B");
  })
  .then(x => { console.log("THEN1", x); })
  .catch(x => { console.log("CATCH2", x); })
  .then(x => { console.log("THEN2", x); });
//CATCH1 Fail
//CATCH2 B
//THEN2 undefined
```

Fetch API

//Метод fetch: замена XMLHttpRequest

```
let promise = fetch(url[, options]);
```

//url – URL, на который сделать запрос,

//options – необязательный объект с настройками запроса.

СВОЙСТВА options

method	метод запроса
headers	заголовки запроса (объект)
body	тело запроса: <code>arrayBuffer</code> , <code>blob</code> , <code>formData</code> , <code>json</code> или <code>text</code>
mode	указывает, в каком режиме кросс-доменности предполагается делать запрос: <code>same-origin</code> , <code>no-cors</code> , <code>cors</code>
credentials	указывает, пересылать ли куки и заголовки авторизации вместе с запросом: <code>omit</code> , <code>same-origin</code> , <code>include</code>
cache	указывает, как кешировать запрос: <code>default</code> , <code>no-store</code> , <code>reload</code> , <code>no-cache</code> , <code>force-cache</code> , <code>only-if-cached</code>
redirect	« <code>follow</code> » для обычного поведения при коде 30х (следовать редиректу) или « <code>error</code> » для интерпретации редиректа как ошибки

Пример 1

```
1.  fetch('/article/fetch/user.json')
2.    .then(function(response) {
3.      console.log(response.headers.get('Content-Type'));
4.      //application/json; charset=utf-8
5.      console.log(response.status);//200
6.      return response.json();
7.    })
8.    .then(function(user) {
9.      console.log(user.name);//Maks
10.   })
11.   .catch(console.log);
```

Пример 2

```
1. var form = new FormData(document.getElementById('login-form'));
2. fetch("/login", {
3.     method: "POST",
4.     body: form
5. });
```