

# Лекция 3

Углубленный JavaScript

# Продвинутая работа с функциями

Подробнее тут <https://learn.javascript.ru/advanced-functions>

## rest-оператор

Попытаемся написать функцию, которая складывает все переданные ей числа. Как это сделать?

```
1 function sumAll(a, b, c, d, e, f, ...) {  
2   // ЧИВО ?  
3 }  
4
```

# rest-оператор

Вуаля!!!!

```
1 function sumAll(...args) { // args – имя массива
2   let sum = 0;
3
4   for (let arg of args) sum += arg;
5
6   return sum;
7 }
8
```

# arguments

rest-оператор это новомодная штука. А как было раньше??

```
1 function sumAll() {  
2   let sum = 0;  
3  
4   for (let arg of arguments) sum += arg;  
5  
6   return sum;  
7 }  
8
```

\*arguments нет в стрелочных функциях

# spread-оператор

Передаем много аргументов в функцию. Как?

```
1 console.log(Math.max(0, -2, 1, 5, 100, 200, 300, -322));  
2
```

## spread-оператор

```
1 const arr1 = [0, -2, 1, 5];  
2  
3 const arr2 = [100, 200, 300, -322];  
4  
5 console.log(Math.max(...arr1, ...arr2));  
6
```

# Замыкание

```
1 let name = "John";  
2  
3 function sayHi() {  
4   alert("Hi, " + name);  
5 }  
6  
7 name = "Pete";  
8  
9 sayHi(); // что будет показано: "John" или "Pete"?
```



# Замыкание

```
1 function makeWorker() {
2   let name = "Pete";
3
4   return function() {
5     alert(name);
6   };
7 }
8
9 let name = "John";
10
11 // create a function
12 let work = makeWorker();
13
14 // call it
15 work(); // что будет показано? "Pete" (из места создания) или "John" (из места выполнения)
16
```

# Лексическое окружение

В JavaScript у каждой выполняемой функции, блока кода и скрипта есть связанный с ними внутренний (скрытый) объект, называемый лексическим окружением ***LexicalEnvironment***.

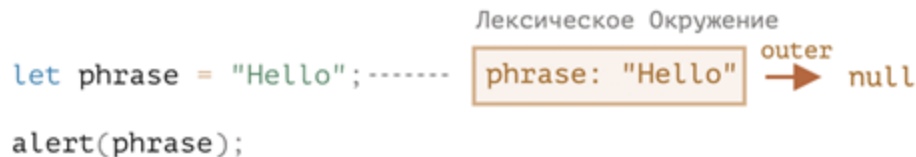
Объект лексического окружения состоит из двух частей:

- ***Environment Record*** – объект, в котором как свойства хранятся все локальные переменные (а также некоторая другая информация, такая как значение `this`).
- Ссылка на внешнее лексическое окружение – то есть то, которое соответствует коду снаружи (снаружи от текущих фигурных скобок).

# Лексическое окружение

"Переменная" – это просто свойство специального внутреннего объекта: *Environment Record*. «Получить или изменить переменную», означает, «получить или изменить свойство этого объекта».

На картинке прямоугольник означает *Environment Record* (хранилище переменных), а стрелка означает ссылку на внешнее окружение. У глобального лексического окружения нет внешнего окружения, так что она указывает на null.



# Лексическое окружение

Итого:

**Переменная** – это свойство специального внутреннего объекта, связанного с текущим выполняющимся блоком/функцией/скриптом.

**Работа с переменными** – это на самом деле работа со свойствами этого объекта.

# Лексическое окружение

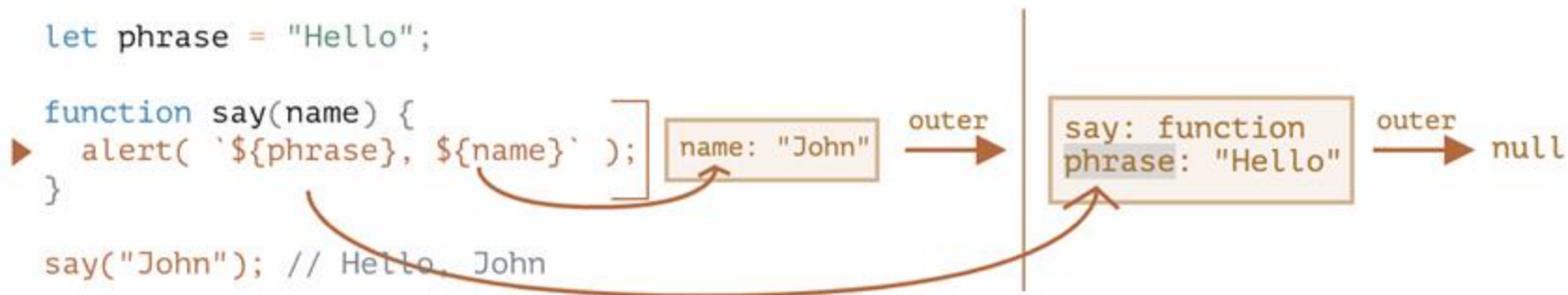
**В отличие от переменных, объявленных с помощью `let`, `Function Declaration` полностью инициализируются не тогда, когда выполнение доходит до них, а раньше, когда создаётся лексическое окружение.**

Для верхнеуровневых функций это означает момент, когда скрипт начинает выполнение.

Вот почему мы можем вызвать функцию, объявленную через `Function Declaration`, до того, как она определена.

# Лексическое окружение

Функция say замыкает на глобальное лексическое окружение, а глобальное лексическое окружение замыкается на null.



# Лексическое окружение

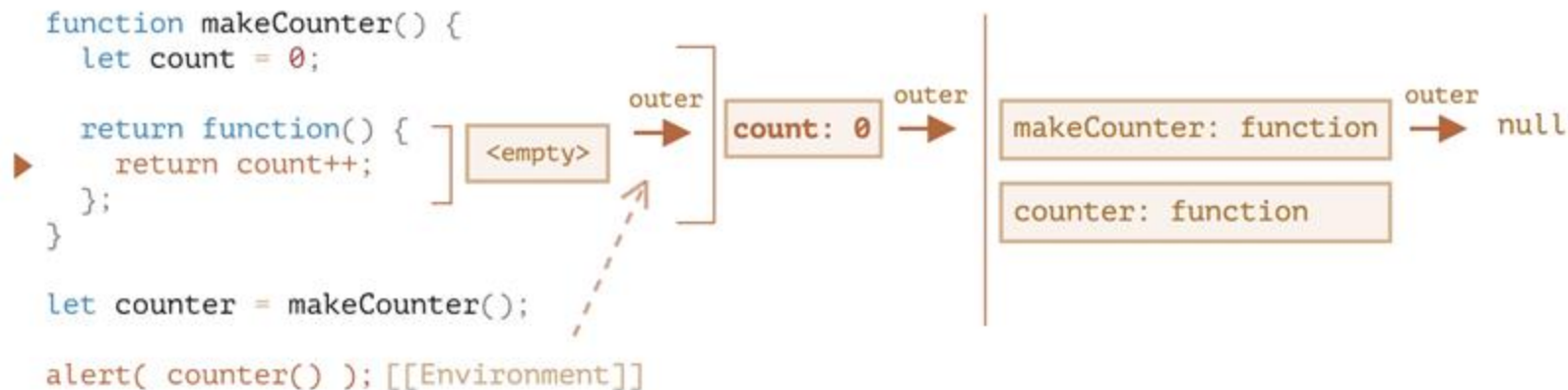
Что тогда происходит с вложенными функциями?

```
1 function makeCounter() {  
2   let count = 0;  
3  
4   return function() {  
5     return count++;  
6   };  
7 }  
8  
9 let counter = makeCounter();  
10  
11 alert( counter() ); // 0  
12 alert( counter() ); // 1  
13 alert( counter() ); // 2
```

# Лексическое окружение

Все функции «при рождении» получают скрытое свойство `[[Environment]]`, которое ссылается на лексическое окружение места, где они были созданы.

Мы ещё не говорили об этом, это то, каким образом функции знают, где они были созданы.

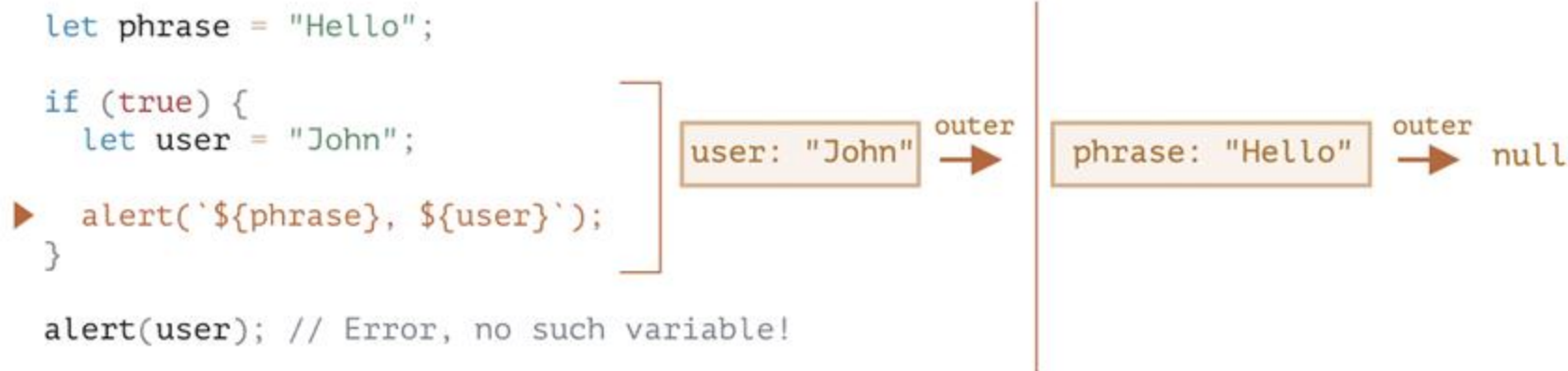




# Лексическое окружение

Где еще есть лексическое окружение?

# Лексическое окружение



# Лексическое окружение

```
1 for (let i = 0; i < 10; i++) {  
2   // У каждой итерации цикла своё собственное лексическое окружение  
3   // {i: value}  
4 }  
5  
6 alert(i); // Ошибка, нет такой переменной
```

# Лексическое окружение

```
1 {  
2   // сделать какую-нибудь работу с локальными переменными, которые не должны быть видны снаружи  
3  
4   let message = "Hello";  
5  
6   alert(message); // Hello  
7 }  
8  
9 alert(message); // Ошибка: переменная message не определена
```

# Лексическое окружение

Но так было не всегда...

```
1 if (true) {  
2     var name = "Леха";  
3 }  
4  
5 console.log(name); // выводит "Леха"  
6
```

# Лексическое окружение

**Immediately-Invoked Function Expressions** - функция, которая вызывается сразу после ее объявления.

```
1 (function() {  
2  
3   let message = "Hello";  
4  
5   alert(message); // Hello  
6  
7 })();  
8
```

# Лексическое окружение

```
1 (function() {  
2   if (true) {  
3     var name = "Леха";  
4   }  
5 })();  
6  
7 console.log(name); // ошибка  
8
```

# Устаревшее объявление переменных

А почему так происходит-то? Потому что старый стандарт и потому-что `var`!

Существует 2 основных отличия `var` от `let/const`:

- Переменные `var` не имеют блочной области видимости, они ограничены, как минимум, телом функции.
- Объявления (инициализация) переменных `var` производится в начале исполнения функции (или скрипта для глобальных переменных).

С первым пунктом понятно (прошлый пример). А второй?



# Устаревшее объявление переменных

```
1 console.log(name); // undefined
2
3 var name = 123;
4
5 console.log(name); // 123
6
```



```
1 var name;
2
3 console.log(name); // undefined
4
5 name = 123;
6
7 console.log(name); // 123
8
```

## Функция-обертка (декорация)

```
1 function slow(x) {  
2   // здесь могут быть ресурсоёмкие вычисления  
3   alert(`Called with ${x}`);  
4   return x;  
5 }  
6  
7 function cachingDecorator(func) {  
8   let cache = new Map();  
9  
10  return function(x) {  
11    if (cache.has(x)) { // если кеш содержит такой x,  
12      return cache.get(x); // читаем из него результат  
13    }  
14  
15    let result = func(x); // иначе, вызываем функцию  
16  
17    cache.set(x, result); // и кешируем (запоминаем) результат  
18    return result;  
19  };  
20 }  
21  
22 slow = cachingDecorator(slow);  
23  
24 alert( slow(1) ); // slow(1) кешируем  
25 alert( "Again: " + slow(1) ); // возвращаем из кеша  
26  
27 alert( slow(2) ); // slow(2) кешируем  
28 alert( "Again: " + slow(2) ); // возвращаем из кеша  
29
```

# Функция-обертка

К сожалению, не работает с методами объектов.

```
1 // сделаем worker.slow кеширующим
2 let worker = {
3   someMethod() {
4     return 1;
5   },
6
7   slow(x) {
8     // здесь может быть страшно тяжёлая задача для процессора
9     alert("Called with " + x);
10    return x * this.someMethod(); // (*)
11  }
12 };
13
14 // тот же код, что и выше
15 function cachingDecorator(func) {
16   let cache = new Map();
17   return function(x) {
18     if (cache.has(x)) {
19       return cache.get(x);
20     }
21     let result = func(x); // (**)
22     cache.set(x, result);
23     return result;
24   };
25 }
26
27 alert( worker.slow(1) ); // оригинальный метод работает
28
29 worker.slow = cachingDecorator(worker.slow); // теперь сделаем его кеширующим
30
31 alert( worker.slow(2) ); // Ой! Ошибка: не удаётся прочитать свойство 'someMethod' из 'undefined'
```

# Функция-обертка

“Почему?!” - спросите вы, потеря контекста - скажу я. В коде ниже будет та же ошибка. При вызове метода `this` — это всегда объект перед точкой.

```
1 let func = worker.slow;  
2 func(2);  
3
```

# Привязываем контекст

Тогда нам нужно насильно указать функции, с каким контекстом она вызывается.

```
1 function say(phrase) {  
2   alert(this.name + ': ' + phrase);  
3 }  
4  
5 let user = { name: "John" };  
6  
7 // 'user' становится 'this', и "Hello" становится первым аргументом  
8 say.call( user, "Hello" ); // John: Hello  
9
```

```
1 let worker = {
2   someMethod() {
3     return 1;
4   },
5
6   slow(x) {
7     alert("Called with " + x);
8     return x * this.someMethod(); // (*)
9   }
10 };
11
12 function cachingDecorator(func) {
13   let cache = new Map();
14   return function(x) {
15     if (cache.has(x)) {
16       return cache.get(x);
17     }
18     let result = func.call(this, x); // теперь 'this' передаётся правильно
19     cache.set(x, result);
20     return result;
21   };
22 }
23
24 worker.slow = cachingDecorator(worker.slow); // теперь сделаем её кеширующей
25
26 alert( worker.slow(2) ); // работает
27 alert( worker.slow(2) ); // работает, не вызывая первоначальную функцию (кешируется)
28
```

## Детально, что произошло

1. После декорации `worker.slow` становится обернут в `function (x) { ... }`.
2. Так что при выполнении `worker.slow(2)` обёртка получает 2 в качестве аргумента и `this=worker` (так как это объект перед точкой).
3. Внутри обёртки, если результат ещё не кеширован, `func.call(this, x)` передаёт текущий `this (=worker)` и текущий аргумент (`=2`) в оригинальную функцию.



## Как еще привязать контекст?

- `func.apply(context, [])` - то же, что и `call`, только принимает аргументы массивом
- `func.bind(context)` - возвращает функцию, к которой привязан контекст, но не вызывает ее (пример дальше)

## Как еще привязать контекст?

```
1 let user = {  
2   firstName: "Вася"  
3 };  
4  
5 function func() {  
6   alert(this.firstName);  
7 }  
8  
9 let funcUser = func.bind(user);  
10 funcUser(); // Вася  
11
```

## Как еще привязать контекст?

```
1 let group = {
2   title: "Our Group",
3   students: ["John", "Pete", "Alice"],
4
5   showList() {
6     this.students.forEach(function(student) {
7       // Error: Cannot read property 'title' of undefined
8       alert(this.title + ': ' + student)
9     });
10  }
11 };
12
13 group.showList();
14
```

## Как еще привязать контекст?

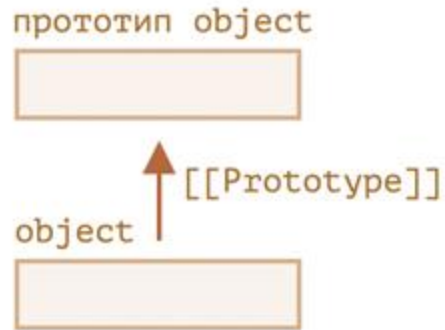
```
1 let group = {
2   title: "Our Group",
3   students: ["John", "Pete", "Alice"],
4
5   showList() {
6     this.students.forEach(
7       student => alert(this.title + ': ' + student)
8     );
9   }
10 };
11
12 group.showList();
13
```

# Прототипы, наследование

Подробнее тут <https://learn.javascript.ru/prototypes>

# [[Prototype]]

В JavaScript объекты имеют специальное скрытое свойство `[[Prototype]]` (так оно названо в спецификации), которое либо равно `null`, либо ссылается на другой объект. Этот объект называется «прототип»



# Прототипное наследование

Когда мы хотим прочитать свойство из object, а оно отсутствует, JavaScript автоматически берёт его из прототипа. Можно задать через `__proto__`.

```
1 let animal = {  
2   eats: true  
3 };  
4 let rabbit = {  
5   jumps: true  
6 };  
7  
8 rabbit.__proto__ = animal; // (*)  
9  
10 // теперь мы можем найти оба свойства в rabbit:  
11 alert( rabbit.eats ); // true (**)  
12 alert( rabbit.jumps ); // true  
13
```

# Прототипное наследование

Когда мы хотим прочитать свойство из object, а оно отсутствует, JavaScript автоматически берёт его из прототипа. Можно задать через `__proto__`.

Обход прототипов происходит от самого дочернего, вниз к родительским.

```
1 let animal = {  
2   eats: true  
3 };  
4 let rabbit = {  
5   jumps: true  
6 };  
7  
8 rabbit.__proto__ = animal; // (*)  
9  
10 // теперь мы можем найти оба свойства в rabbit:  
11 alert( rabbit.eats ); // true (**)  
12 alert( rabbit.jumps ); // true  
13
```



# F.prototype

Если в F.prototype содержится объект, оператор new устанавливает его в качестве [[Prototype]] для нового объекта.

```
1 let animal = {  
2   eats: true  
3 };  
4  
5 function Rabbit(name) {  
6   this.name = name;  
7 }  
8  
9 Rabbit.prototype = animal;  
10  
11 let rabbit = new Rabbit("White Rabbit"); // rabbit.__proto__ == animal  
12  
13 alert( rabbit.eats ); // true  
14
```

## F.prototype

На прошлом слайде установка `Rabbit.prototype = animal` буквально говорит интерпретатору следующее: "При создании объекта через `new Rabbit()` запиши ему `animal` в `[[Prototype]]`".

# prototype по-умолчанию

У каждой функции по умолчанию уже есть свойство `prototype`.

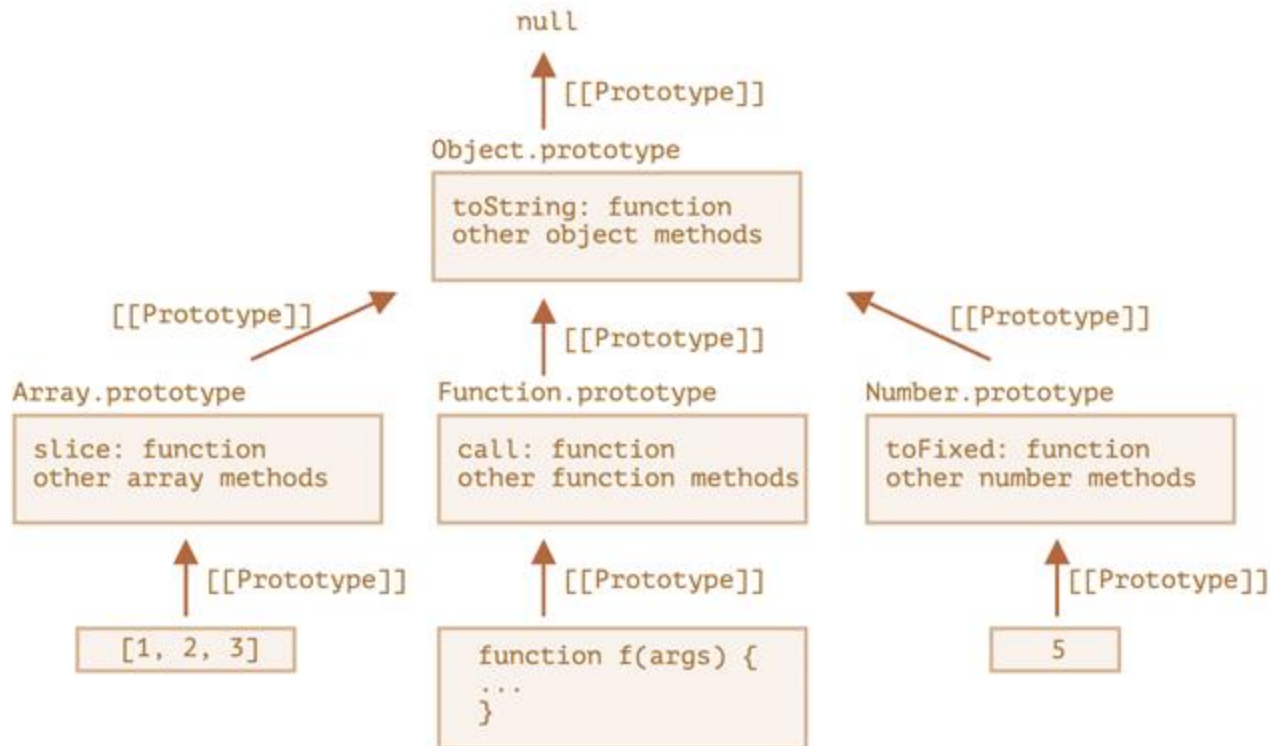
По умолчанию `prototype` – объект с единственным свойством `constructor`, которое ссылается на функцию-конструктор.

```
1 function Rabbit() {}  
2 // по умолчанию:  
3 // Rabbit.prototype = { constructor: Rabbit }  
4  
5 alert( Rabbit.prototype.constructor == Rabbit ); // true  
6  
7 let rabbit = new Rabbit(); // наследует от {constructor: Rabbit}  
8  
9 alert(rabbit.constructor == Rabbit); // true (свойство получено из прототипа)  
10
```

# Встроенные прототипы

```
1 let arr = [1, 2, 3];  
2  
3 // наследует ли от Array.prototype?  
4 alert( arr.__proto__ === Array.prototype ); // true  
5  
6 // затем наследует ли от Object.prototype?  
7 alert( arr.__proto__.__proto__ === Object.prototype ); // true  
8  
9 // и null на вершине иерархии  
10 alert( arr.__proto__.__proto__.__proto__ ); // null  
11
```

# Встроенные прототипы

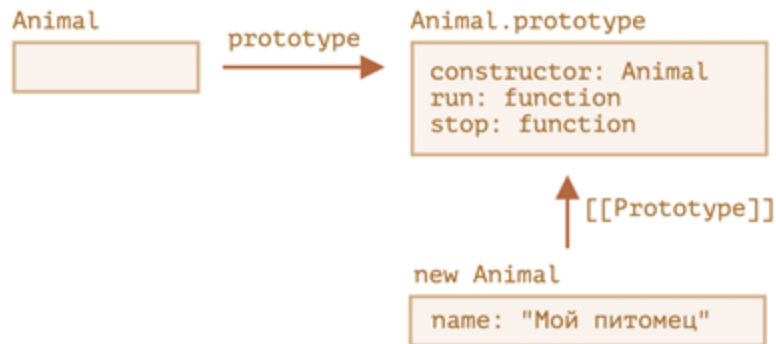


# Продвинутые классы

Подробнее тут <https://learn.javascript.ru/classes>

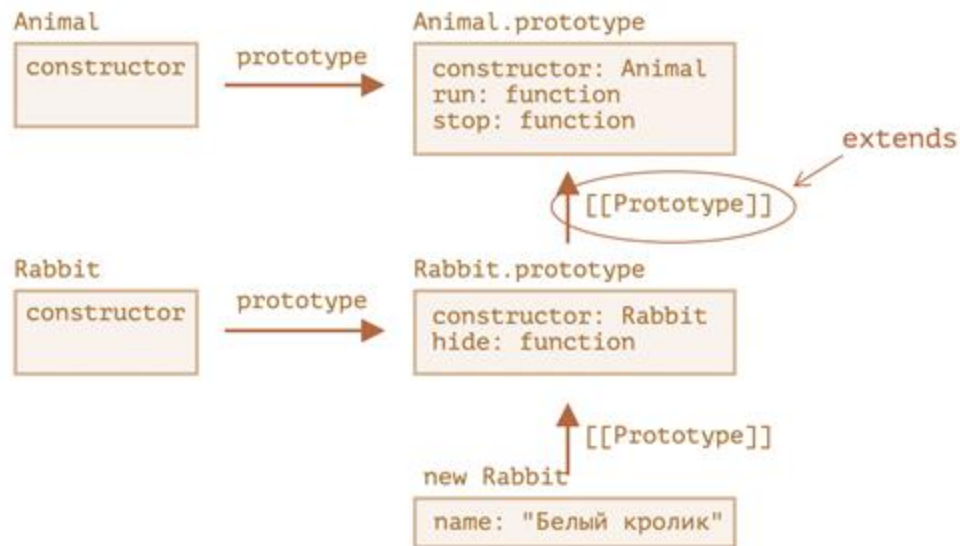
# Сразу к наследованию

```
1 class Animal {  
2   constructor(name) {  
3     this.speed = 0;  
4     this.name = name;  
5   }  
6   run(speed) {  
7     this.speed = speed;  
8     alert(`${this.name} бежит со скоростью ${this.speed}.`);  
9   }  
10  stop() {  
11    this.speed = 0;  
12    alert(`${this.name} стоит неподвижно.`);  
13  }  
14 }  
15  
16 let animal = new Animal("Мой питомец");  
17
```



# Сразу к наследованию

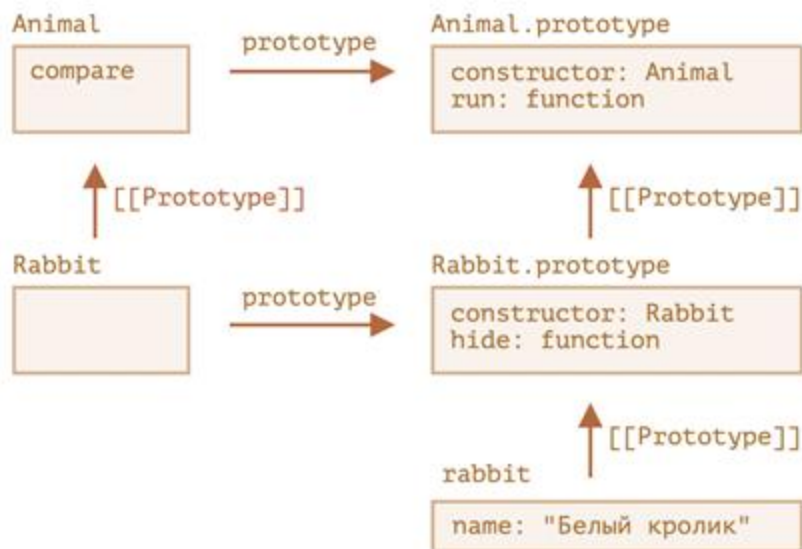
```
1 class Rabbit extends Animal {  
2   hide() {  
3     alert(`${this.name} прячется!`);  
4   }  
5 }  
6  
7 let rabbit = new Rabbit("Белый кролик");  
8  
9 rabbit.run(5); // Белый кролик бежит со скоростью 5.  
10 rabbit.hide(); // Белый кролик прячется!  
11
```





# А статические методы?

Так что `Rabbit extends Animal` создаёт две ссылки на прототип: функция `Rabbit` прототипно наследует от функции `Animal`, `Rabbit.prototype` прототипно наследует от `Animal.prototype`.



```
1 class Animal {
2
3   constructor(name, speed) {
4     this.speed = speed;
5     this.name = name;
6   }
7
8   run(speed = 0) {
9     this.speed += speed;
10    alert(`${this.name} бежит со скоростью ${this.speed}.`);
11  }
12
13  static compare(animalA, animalB) {
14    return animalA.speed - animalB.speed;
15  }
16
17 }
18
19 // Наследует от Animal
20 class Rabbit extends Animal {
21   hide() {
22     alert(`${this.name} прячется!`);
23   }
24 }
25
26 let rabbits = [
27   new Rabbit("Белый кролик", 10),
28   new Rabbit("Чёрный кролик", 5)
29 ];
30
31 rabbits.sort(Rabbit.compare);
32
33 rabbits[0].run(); // Чёрный кролик бежит со скоростью 5.
34
```

# Я ничего не понял

Это нормально!!!

Как правило мы редко сталкиваемся с этим на работе или учебе. Эта информация нужна для более комплексного понимания того, как устроен javascript (ну и на собеседованиях спрашивают).

# Модули

Подробнее тут <https://learn.javascript.ru/modules>

# Модули

- AMD - одна из самых старых модульных систем, изначально реализована библиотекой `require.js`.
- CommonJS – модульная система, созданная для сервера `Node.js`.
- UMD - ещё одна модульная система, предлагается как универсальная, совместима с AMD и CommonJS.

# Модули (ESM)

Модуль – это просто файл. Один скрипт – это один модуль.

Модули могут загружать друг друга и использовать директивы `export` и `import`, чтобы обмениваться функциональностью, вызывать функции одного модуля из другого:

- `export` отмечает переменные и функции, которые должны быть доступны вне текущего модуля
- `import` позволяет импортировать функциональность других модулей

# Модули (импорты, экспорты)

```
1 // 📁 sayHi.js
2 export function sayHi(user) {
3   alert(`Hello, ${user}!`);
4 }
```

```
1 // 📁 main.js
2 import {sayHi} from './sayHi.js';
3
4 alert(sayHi); // function...
5 sayHi('John'); // Hello, John!
```

## Модули (браузер)

```
1 <!doctype html>
2 <script type="module">
3   import {sayHi} from './say.js';
4
5   document.body.innerHTML = sayHi('John');
6 </script>
```

## Модули (область видимости)

```
1 <!doctype html>
2 <script type="module" src="user.js"></script>
3 <script type="module" src="hello.js"></script>
```

```
1 // user.js
2 let user = "John";
```

```
1 // hello.js
2 alert(user);
```



## Модули (выполняется 1 раз)

```
1 // 📁 alert.js  
2 alert("Модуль выполнен!");
```

```
1 // Импорт одного и того же модуля в разных файлах  
2  
3 // 📁 1.js  
4 import `./alert.js`; // Модуль выполнен!  
5  
6 // 📁 2.js  
7 import `./alert.js`; // (ничего не покажет)
```

# Модули (выполняется 1 раз)

```
1 // 📁 admin.js
2 export let admin = {
3   name: "John"
4 };
```

```
1 // 📁 1.js
2 import {admin} from './admin.js';
3 admin.name = "Pete";
4
5 // 📁 2.js
6 import {admin} from './admin.js';
7 alert(admin.name); // Pete
8
9 // Оба файла, 1.js и 2.js, импортируют один и тот же объект
10 // Изменения, сделанные в 1.js, будут видны в 2.js
```

## Модули (“голые” модули)

```
1 import {sayHi} from 'sayHi'; // Ошибка, "голый" модуль  
2 // путь должен быть, например './sayHi.js' или абсолютный
```

# Модули (export)

```
1 // экспорт массива
2 export let months = ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
3
4 // экспорт константы
5 export const MODULES_BECAME_STANDARD_YEAR = 2015;
6
7 // экспорт класса
8 export class User {
9   constructor(name) {
10     this.name = name;
11   }
12 }
```

## Модули (export)

```
1 // 📁 say.js
2 function sayHi(user) {
3   alert(`Hello, ${user}!`);
4 }
5
6 function sayBye(user) {
7   alert(`Bye, ${user}!`);
8 }
9
10 export {sayHi, sayBye}; // список экспортируемых переменных
```

## Модули (import)

```
1 // 📁 main.js
2 import {sayHi, sayBye} from './say.js';
3
4 sayHi( 'John' ); // Hello, John!
5 sayBye( 'John' ); // Bye, John!
```

## Модули (import)

```
1 // 📁 main.js
2 import * as say from './say.js';
3
4 say.sayHi( 'John' );
5 say.sayBye( 'John' );
```

## Модули (export, import as)

```
1 // 📁 main.js
2 import {sayHi as hi, sayBye as bye} from './say.js';
3
4 hi('John'); // Hello, John!
5 bye('John'); // Bye, John!
```

```
1 // 📁 say.js
2 ...
3 export {sayHi as hi, sayBye as bye};
```



## Модули (export default)

```
1 // 📁 user.js
2 export default class User { // просто добавьте "default"
3   constructor(name) {
4     this.name = name;
5   }
6 }
```

```
1 // 📁 main.js
2 import User from './user.js'; // не {User}, просто User
3
4 new User('John');
```

# Модули (export default)

```
1 // 📁 user.js
2 export default class User { // просто добавьте "default"
3   constructor(name) {
4     this.name = name;
5   }
6 }
```

```
1 // 📁 main.js
2 import User, { sayHi } from './user.js'; // не {User}, просто User
3
4 new User('John');
5 sayHi('John'); // Hello, John!
```

## Модули (re-export)

```
1 export {sayHi} from './say.js'; // реэкспортировать sayHi
2
3 export {default as User} from './user.js'; // реэкспортировать default
```

# Модули (dynamic import)

```
1 //  say.js
2 export function hi() {
3   alert('Привет');
4 }
5
6 export function bye() {
7   alert('Пока');
8 }
```

```
1 let {hi, bye} = await import('./say.js');
2
3 hi();
4 bye();
```

# Обработка ошибок

Подробнее тут <https://learn.javascript.ru/error-handling>

# Ошибка (try catch)



## Ошибка (try catch)

```
1 try {  
2  
3   alert('Начало блока try'); // (1) <--  
4  
5   lalala; // ошибка, переменная не определена!  
6  
7   alert('Конец блока try (никогда не выполнится)'); // (2)  
8  
9 } catch(err) {  
10  
11   alert(`Возникла ошибка!`); // (3) <--  
12  
13 }
```

# Ошибка (try catch)

```
1 let json = "{ некорректный JSON }";
2
3 try {
4
5   let user = JSON.parse(json); // <-- тут возникает ошибка...
6   alert( user.name ); // не сработает
7
8 } catch (e) {
9   // ...выполнение прыгает сюда
10  alert( "Извините, в данных ошибка, мы попробуем получить их ещё раз." );
11  alert( e.name );
12  alert( e.message );
13 }
```



## Ошибка (try catch)

```
1 try {  
2     setTimeout(function() {  
3         noSuchVariable; // скрипт упадёт тут  
4     }, 1000);  
5 } catch (e) {  
6     alert( "не работает" );  
7 }
```

## Ошибка (try catch finally)

```
1 try {  
2     alert( 'try' );  
3     if (confirm( 'Сгенерировать ошибку?' )) BAD_CODE();  
4 } catch (e) {  
5     alert( 'catch' );  
6 } finally {  
7     alert( 'finally' );  
8 }
```

# Ошибка (объект ошибки)

- `name` - имя ошибки.
- `message` – сообщение ошибки.
- `stack` - текущий стек вызова.

```
1 try {  
2   // ...  
3 } catch(err) { // <-- объект ошибки  
4   // ...  
5 }
```

## Ошибка (генерация ошибок)

```
1 throw new Error(message);  
2 throw new SyntaxError(message);  
3 throw new ReferenceError(message);
```

# Ошибка (пользовательские ошибки)

```
1 class ValidationError extends Error {
2   constructor(message) {
3     super(message); // (1)
4     this.name = "ValidationError"; // (2)
5   }
6 }
7
8 function test() {
9   throw new ValidationError("Ync!");
10 }
11
12 try {
13   test();
14 } catch(err) {
15   alert(err.message); // Ync!
16   alert(err.name); // ValidationError
17   alert(err.stack); // список вложенных вызовов с номерами строк для каждого
18 }
```

## Ошибка (пользовательские ошибки)

```
1 class ValidationError extends Error {  
2   constructor(message) {  
3     super(message);  
4     this.name = "ValidationError";  
5   }  
6 }  
7  
8 class PropertyRequiredError extends ValidationError {  
9   constructor(property) {  
10    super("Нет свойства: " + property);  
11    this.name = "PropertyRequiredError";  
12    this.property = property;  
13  }  
14 }
```

## Ошибка (пользовательские ошибки)

```
1 try {  
2     validateUser(user);  
3 } catch (err) {  
4     if (err instanceof ValidationError) {  
5         throw new ReadError("Ошибка валидации", err);  
6     } else {  
7         throw err;  
8     }  
9 }
```

# Регулярные выражения

Подробнее тут <https://learn.javascript.ru/regular-expressions>



# Регулярные выражения

```
1 regexr = new RegExp( "шаблон", "флаги" );  
2 regexr = /шаблон/; // без флагов  
3 regexr = /шаблон/gmi; // с флагами gmi
```

## Регулярные выражения (str.match)

```
1 let str = "Любо, братцы, любо!";  
2  
3 let result = str.match(/любо/i); // без флага g  
4  
5 alert( result[0] );    // Любо (первое совпадение)  
6 alert( result.length ); // 1  
7  
8 // Дополнительная информация:  
9 alert( result.index ); // 0 (позиция совпадения)  
10 alert( result.input ); // Любо, братцы, любо! (исходная строка)
```

# Регулярные выражения (str.replace)

```
1 // без флага g
2 alert( "We will, we will".replace(/we/i, "I") ); // I will, we will
3
4 // с флагом g
5 alert( "We will, we will".replace(/we/ig, "I") ); // I will, I will
```

## Регулярные выражения (regex.test)

```
1 let str = "Я ЛюБлю JavaScript";  
2 let regexp = /люблю/i;  
3  
4 alert( regexp.test(str) ); // true
```

# События

Подробнее тут <https://learn.javascript.ru/events>

# События

## События мыши:

- `click` – происходит, когда кликнули на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании).
- `contextmenu` – происходит, когда кликнули на элемент правой кнопкой мыши.
- `mouseover` / `mouseout` – когда мышь наводится на / покидает элемент.
- `mousedown` / `mouseup` – когда нажали / отжали кнопку мыши на элементе.
- `mousemove` – при движении мыши.

## События на элементах управления:

- `submit` – пользователь отправил форму `<form>`.
- `focus` – пользователь фокусируется на элементе, например нажимает на `<input>`.

## Клавиатурные события:

- `keydown` и `keyup` – когда пользователь нажимает / отпускает клавишу.

# События (обработчики)

```
1 <input value="Нажми меня" onclick="alert('Клик!')" type="button">
```

```
1 <input type="button" id="button" value="Кнопка">
2 <script>
3   document.getElementById('button').onclick = function() {
4     alert('Клик!');
5   };
6 </script>
```

## События (обработчики)

```
1 function handler() {  
2     alert( 'Спасибо!' );  
3 }  
4  
5 input.addEventListener("click", handler);  
6 // ....  
7 input.removeEventListener("click", handler);
```



# События (объект события)

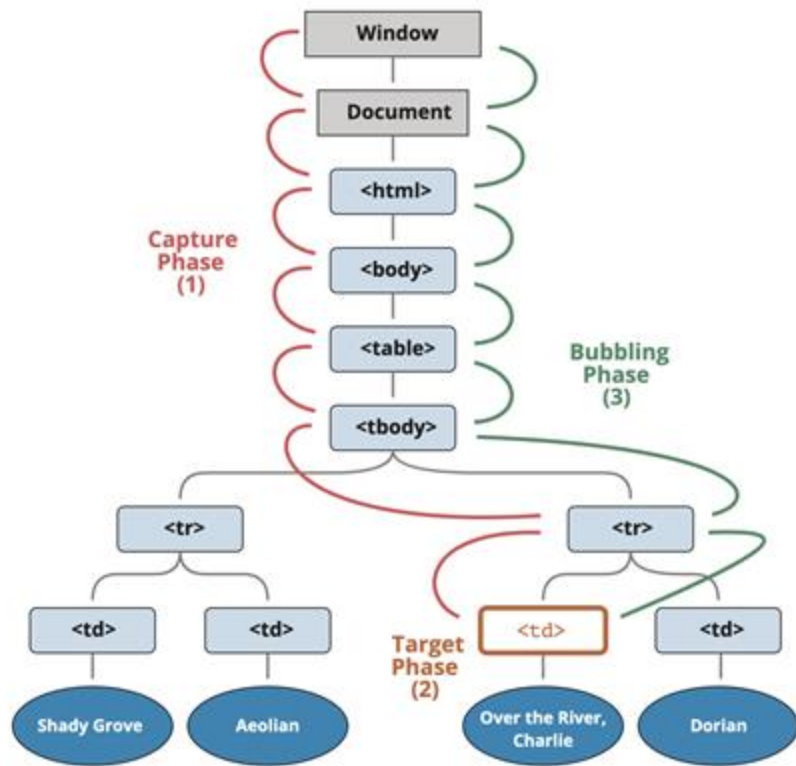
- `event.type` - тип события.
- `event.currentTarget` – элемент, на котором сработал обработчик.
- `event.clientX` / `event.clientY` - координаты курсора.

```
1 <input type="button" value="Нажми меня" id="elem">
2
3 <script>
4   document.getElementById('elem').onclick = function(event) {
5     // вывести тип события, элемент и координаты клика
6     alert(event.type + " на " + event.currentTarget);
7     alert("Координаты: " + event.clientX + ":" + event.clientY);
8   };
9 </script>
```

# События

```
1 <form onclick="alert( 'form' )" >FORM
2   <div onclick="alert( 'div' )" >DIV
3     <p onclick="alert( 'p' )" >P</p>
4   </div>
5 </form>
```

# События (всплытие и погружение)



# События (всплытие и погружение)

Клик по внутреннему `<p>` вызовет обработчик `onclick`:

1. Сначала на самом `<p>`.
2. Потом на внешнем `<div>`.
3. Затем на внешнем `<form>`.
4. И так далее вверх по цепочке до самого `document`.

```
1 <style>
2   body * {
3     margin: 10px;
4     border: 1px solid blue;
5   }
6 </style>
7
8 <form onclick="alert('form')">FORM
9   <div onclick="alert('div')">DIV
10     <p onclick="alert('p')">P</p>
11   </div>
12 </form>
```

# События (всплытие и погружение)

```
1 <style>
2   body * {
3     margin: 10px;
4     border: 1px solid blue;
5   }
6 </style>
7
8 <form>FORM
9   <div>DIV
10    <p>P</p>
11  </div>
12 </form>
13
14 <script>
15   for(let elem of document.querySelectorAll('*')) {
16     elem.addEventListener("click", e => alert(`Погружение: ${elem.tagName}`), true);
17     elem.addEventListener("click", e => alert(`Всплытие: ${elem.tagName}`));
18   }
19 </script>
```

# События (всплытие и погружение)

```
1 <body onclick="alert(`сюда всплытие не дойдёт`)">
2   <button onclick="event.stopPropagation()">Кликни меня</button>
3 </body>
```

```
1 <body onclick="alert(`сюда всплытие не дойдёт`)">
2   <button onclick="event.stopImmediatePropagation()">Кликни меня</button>
3 </body>
```

## События (всплытие и погружение)

```
1 elem.addEventListener(..., {capture: true})  
2 // или просто "true", как сокращение для {capture: true}  
3 elem.addEventListener(..., true)
```

## События (по умолчанию)

```
1 <a href="/" onclick="return false">Нажми здесь</a>
```

```
2 или
```

```
3 <a href="/" onclick="event.preventDefault()">здесь</a>
```



# События (пользовательские события)

```
1 <button id="elem" onclick="alert('Клик!');">Автоклик</button>
2
3 <script>
4   let event = new Event("click");
5   document.getElementById('elem').dispatchEvent(event);
6 </script>
```

```
1 document.addEventListener("hello", function(event) {
2   alert("Привет от " + event.target.tagName);
3 });
```