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
```

Планета	Количество смертей	Есть JavaScript
	0	Нет
	0	Нет
	120 315 872 896+	Да
	0	Нет
	0	Нет
	0	Нет
	0	Нет
	0	Нет

Замыкание

```
let name = "John";

function sayHi() {
  alert("Hi, " + name);
}

name = "Pete";

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

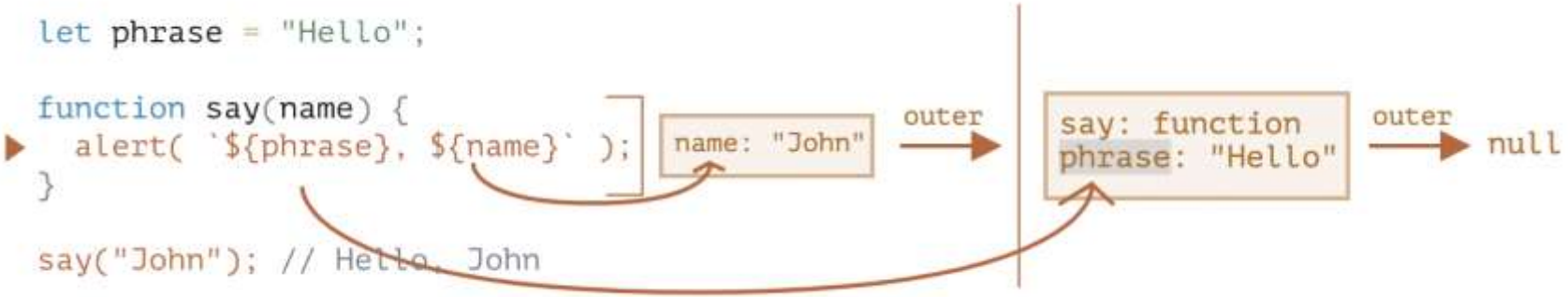
В отличие от переменных, объявленных с помощью `let`, Function Declaration полностью инициализируются не тогда, когда выполнение доходит до них, а раньше, когда создаётся лексическое окружение.

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

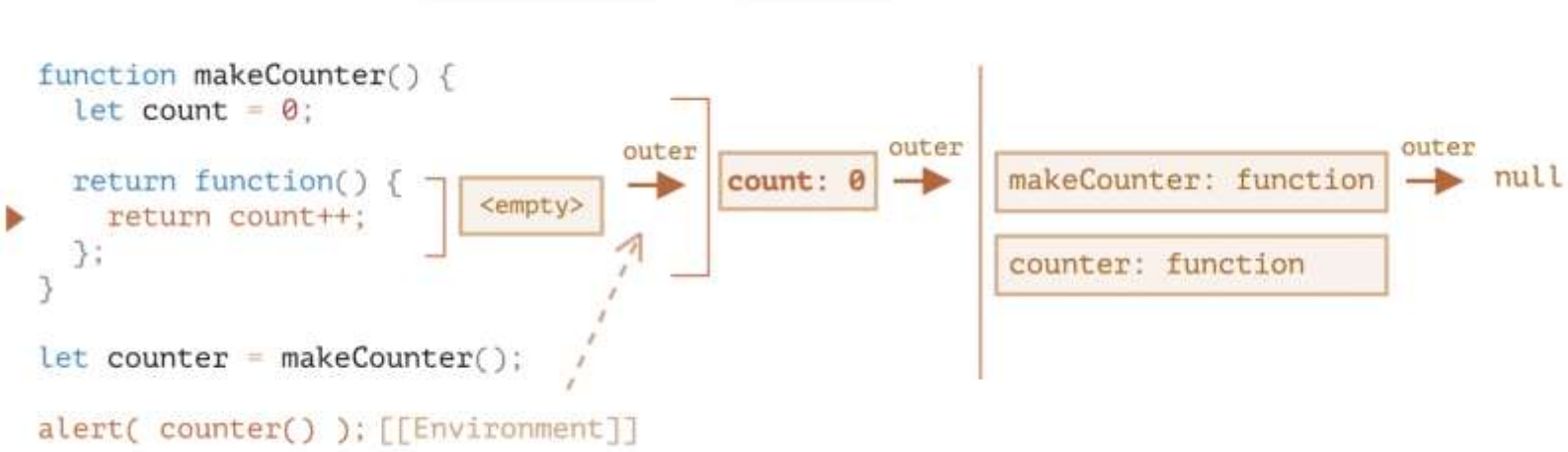
Что тогда происходит с вложенными функциями?

```
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.

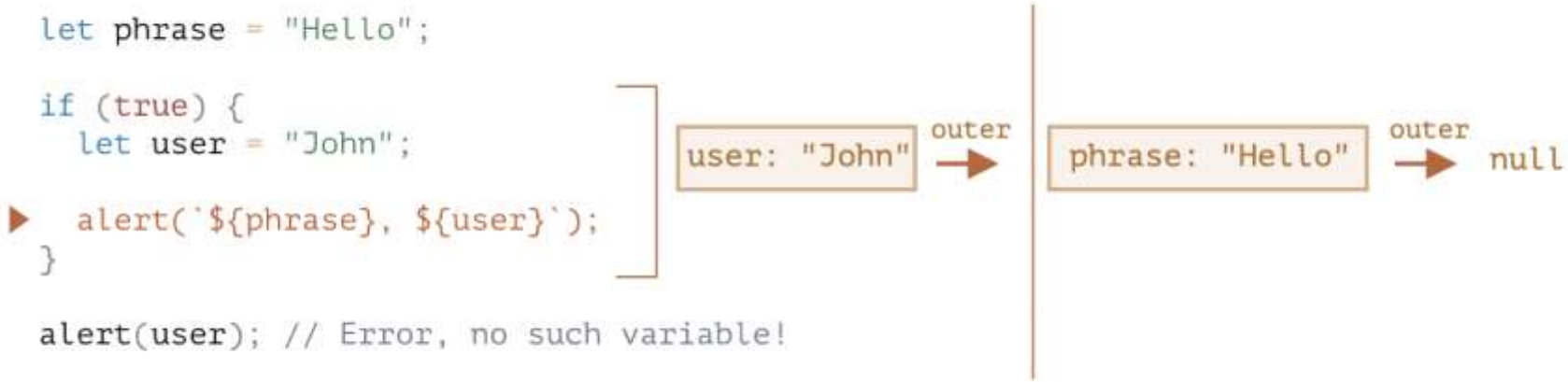
Мы ещё не говорили об этом, это то, каким образом функции знают, где они были созданы.



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

```
{
  // сделать какую-нибудь работу с локальными переменными, которые не должны быть видны снаружи

  let message = "Hello";

  alert(message); // Hello
}

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
```

Объявление переменных для дедушек и дураков

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?

Объявление переменных для дедушек и дураков

```
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
```


15 Fun Coding Projects!
Learn coding and build cool stuff!
JavaScript
FOR
KIDS


15 Fun Coding Projects!
Learn coding and build cool stuff!
JavaScript
FOR
KIDS
Build an Animated Robot App

Функция-обертка (декорация)

```
function slow(x) {
  // здесь могут быть ресурсоёмкие вычисления
  alert(`Called with ${x}`);
  return x;
}

function cachingDecorator(func) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) {    // если кеш содержит такой x,
      return cache.get(x); // читаем из него результат
    }

    let result = func(x); // иначе, вызываем функцию

    cache.set(x, result); // и кешируем (запоминаем) результат
    return result;
  };
}

slow = cachingDecorator(slow);

alert( slow(1) ); // slow(1) кешируем
alert( "Again: " + slow(1) ); // возвращаем из кеша

alert( slow(2) ); // slow(2) кешируем
alert( "Again: " + slow(2) ); // возвращаем из кеша

```

Функция-обертка

К сожаления, не работает с методами объектов.

```
// сделаем worker.slow кеширующим
let worker = {
  someMethod() {
    return 1;
  },

  slow(x) {
    // здесь может быть страшно тяжёлая задача для процессора
    alert("Called with " + x);
    return x * this.someMethod(); // (*)
  }
};

// тот же код, что и выше
function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
    let result = func(x); // (**)
    cache.set(x, result);
    return result;
  };
}

alert( worker.slow(1) ); // оригинальный метод работает

worker.slow = cachingDecorator(worker.slow); // теперь сделаем его кеширующим

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
```

```
let worker = {
  someMethod() {
    return 1;
  },

  slow(x) {
    alert("Called with " + x);
    return x * this.someMethod(); // (*)
  }
};

function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
    let result = func.call(this, x); // теперь 'this' передаётся правильно
    cache.set(x, result);
    return result;
  };
}

worker.slow = cachingDecorator(worker.slow); // теперь сделаем её кеширующей

alert( worker.slow(2) ); // работает
alert( worker.slow(2) ); // работает, не вызывая первоначальную функцию (кеширует)

```

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 контекс, но не вызывает ее (пример дальше)

Как еще привязать контекст?

```
let user = {
  firstName: "Вася"
};

function func() {
  alert(this.firstName);
}

let funcUser = func.bind(user);
funcUser(); // Вася

```

Как еще привязать контекст?

```
let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],

  showList() {
    this.students.forEach(function(student) {
      // Error: Cannot read property 'title' of undefined
      alert(this.title + ': ' + student)
    });
  }
};

group.showList();

```

Как еще привязать контекст?

```
let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],

  showList() {
    this.students.forEach(
      student => alert(this.title + ': ' + student)
    );
  }
};

group.showList();

```

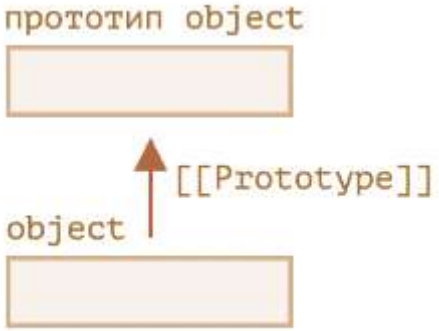

JavaScript...
> '5' - 3
2
> '5' + 3
'53'
> '5' + - '2'
'5-2'
...буду проституткой

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__`.

```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal; // (*)

// теперь мы можем найти оба свойства в rabbit:
alert( rabbit.eats ); // true (**)
alert( rabbit.jumps ); // true

```

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.

```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal; // (*)

// теперь мы можем найти оба свойства в rabbit:
alert( rabbit.eats ); // true (**)
alert( rabbit.jumps ); // true

```

F.prototype

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

```
let animal = {
  eats: true
};

function Rabbit(name) {
  this.name = name;
}

Rabbit.prototype = animal;

let rabbit = new Rabbit("White Rabbit"); //  rabbit.__proto__ == animal

alert( rabbit.eats ); // true

```

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`.

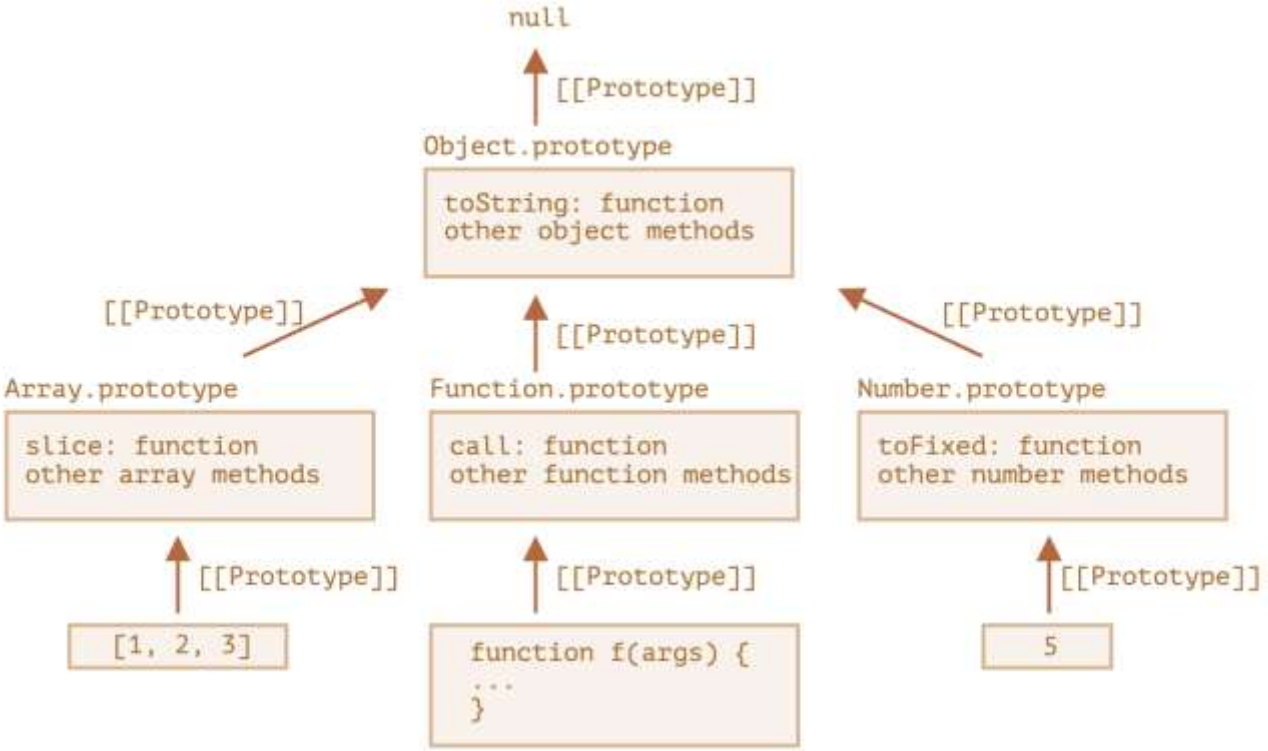
По умолчанию `prototype` – объект с единственным свойством constructor, которое ссылается на функцию-конструктор.

```
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

```
let arr = [1, 2, 3];

// наследует ли от Array.prototype?
alert( arr.__proto__ === Array.prototype ); // true

// затем наследует ли от Object.prototype?
alert( arr.__proto__.__proto__ === Object.prototype ); // true

// и null на вершине иерархии
alert( arr.__proto__.__proto__.__proto__ ); // null

```

Встроенные прототипы

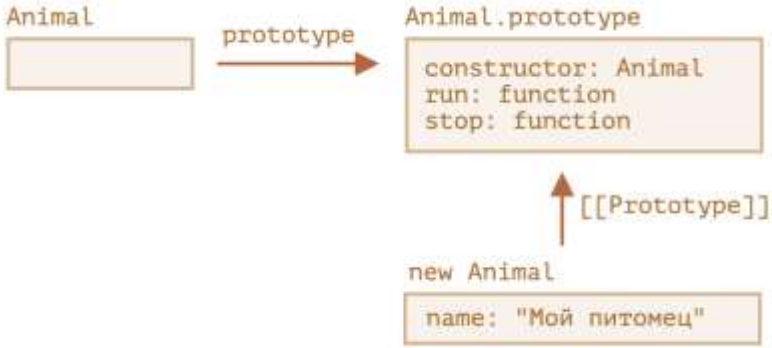


Продвинутые классы

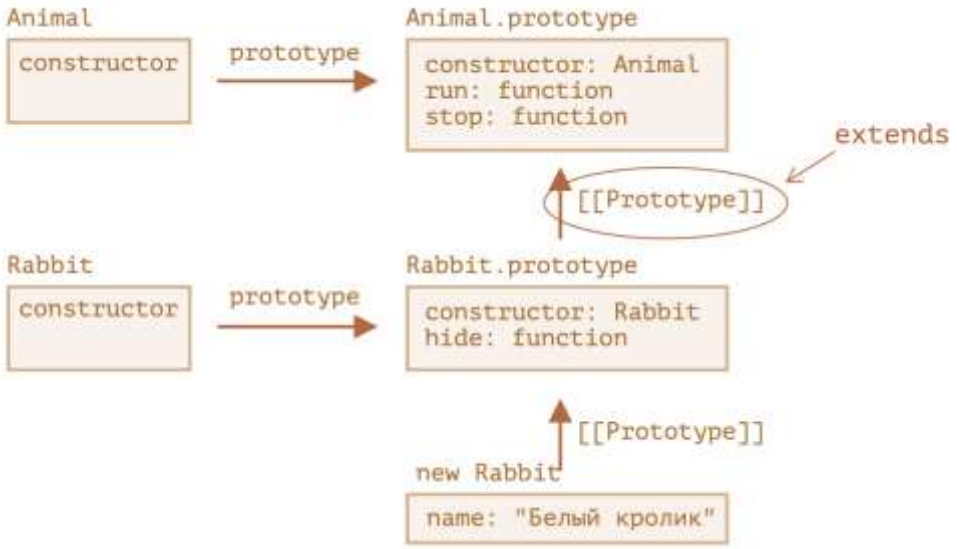
Подробнее тут https://learn.javascript.ru/classes

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
```



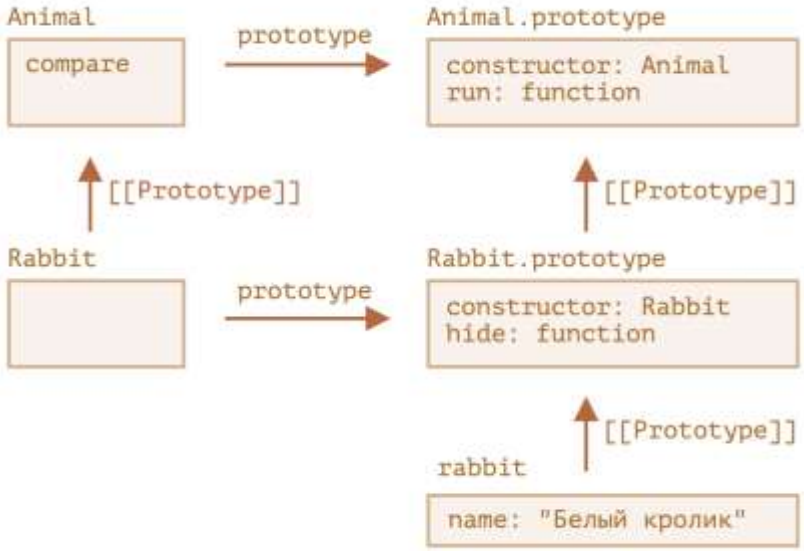
Сразу к наследованию




```
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`.




```
class Animal {

  constructor(name, speed) {
    this.speed = speed;
    this.name = name;
  }

  run(speed = 0) {
    this.speed += speed;
    alert(`${this.name} бежит со скоростью ${this.speed}.`);
  }

  static compare(animalA, animalB) {
    return animalA.speed - animalB.speed;
  }

}

// Наследует от Animal
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} прячется!`);
  }
}

let rabbits = [
  new Rabbit("Белый кролик", 10),
  new Rabbit("Чёрный кролик", 5)
];

rabbits.sort(Rabbit.compare);

rabbits[0].run(); // Чёрный кролик бежит со скоростью 5.

```

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).

Я люблю JavaScript

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)

```
<!doctype html>
<script type="module">
  import {sayHi} from './say.js';

  document.body.innerHTML = sayHi('John');
</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)

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

```
// 📁 1.js
import {admin} from './admin.js';
admin.name = "Pete";

// 📁 2.js
import {admin} from './admin.js';
alert(admin.name); // Pete

// Оба файла, 1.js и 2.js, импортируют один и тот же объект
// 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)

```
// 📁 user.js
export default class User { // просто добавьте "default"
  constructor(name) {
    this.name = name;
  }
}
```

```
// 📁 main.js
import User from './user.js'; // не {User}, просто User

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)

```
// say.js
export function hi() {
  alert(`Привет`);
}

export function bye() {
  alert(`Пока`);
}
```

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

hi();
bye();
```

Обработка ошибок

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

Ошибка (try catch)

Ошибка (try catch)

```
try {

  alert('Начало блока try');  // (1) <--

  lalala; // ошибка, переменная не определена!

  alert('Конец блока try (никогда не выполнится)');  // (2)

} catch(err) {

  alert(`Возникла ошибка!`); // (3) <--

}
```

Ошибка (try catch)

```
let json = "{ некорректный JSON }";

try {

  let user = JSON.parse(json); // <-- тут возникает ошибка...
  alert( user.name ); // не сработает

} catch (e) {
  // ...выполнение прыгает сюда
  alert( "Извините, в данных ошибка, мы попробуем получить их ещё раз." );
  alert( e.name );
  alert( e.message );
}
```

Ошибка (try catch)

```
try {
  setTimeout(function() {
    noSuchVariable; // скрипт упадёт тут
  }, 1000);
} catch (e) {
  alert( "не сработает" );
}
```

Ошибка (try catch finally)

```
try {
  alert( 'try' );
  if (confirm('Сгенерировать ошибку?')) BAD_CODE();
} catch (e) {
  alert( 'catch' );
} finally {
  alert( 'finally' );
}
```

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("Упс!");
10 }
11
12 try {
13   test();
14 } catch(err) {
15   alert(err.message); // Упс!
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)

```
try {
    validateUser(user);
  } catch (err) {
    if (err instanceof ValidationError) {
      throw new ReadError("Ошибка валидации", err);
    } else {
      throw err;
    }
  }
```

Регулярные выражения

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

Регулярные выражения

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

Регулярные выражения (str.match)

```
let str = "Любо, братцы, любо!";

let result = str.match(/любо/i); // без флага g

alert( result[0] );     // Любо (первое совпадение)
alert( result.length ); // 1

// Дополнительная информация:
alert( result.index );  // 0 (позиция совпадения)
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 (regexp.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)

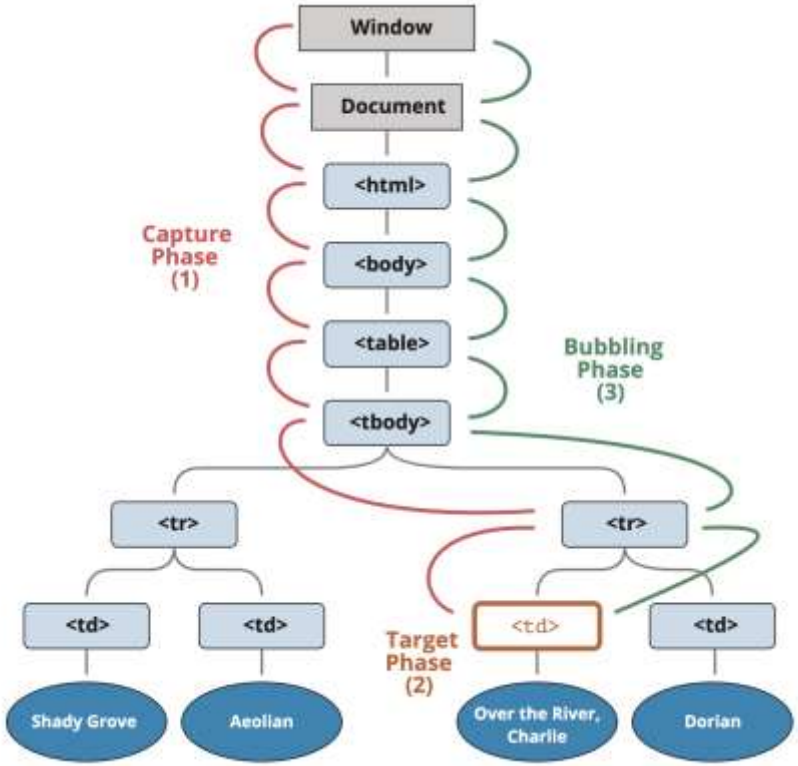
- `event.type` - тип события.
- `event.currentTarget` – элемент, на котором сработал обработчик.
- `event.clientX / event.clientY` - координаты курсора.

```
<input type="button" value="Нажми меня" id="elem">

<script>
  document.getElemntById('elem').onclick = function(event) {
    // вывести тип события, элемент и координаты клика
    alert(event.type + " на " + event.currentTarget);
    alert("Координаты: " + event.clientX + ":" + event.clientY);
  };
</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 .
3. Затем на внешнем <form>.
4. И так далее вверх по цепочке до самого document.

```
<style>
  body * {
    margin: 10px;
    border: 1px solid blue;
  }
</style>

<form onclick="alert('form')">FORM
  <div onclick="alert('div')">DIV
    <p onclick="alert('p')">P</p>
  </div>
</form>
```

События (всплытие и погружение)

```
<style>
  body * {
    margin: 10px;
    border: 1px solid blue;
  }
</style>

<form>FORM
  <div>DIV
    <p>P</p>
  </div>
</form>

<script>
  for(let elem of document.querySelectorAll('*')) {
    elem.addEventListener("click", e => alert(`Погружение: ${elem.tagName}`), true);
    elem.addEventListener("click", e => alert(`Всплытие: ${elem.tagName}`));
  }
</script>
```

События (всплытие и погружение)

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

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

```
<button id="elem" onclick="alert('Клик!');">Автоклик</button>

<script>
  let event = new Event("click");
  document.getElementById('elem').dispatchEvent(event);
</script>
```

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