JavaScript, полный конспект

Ссылки

```
JS, React, Redux
regexp101
Помогает составлять регулярные выражения

cors-anywhere.herokuapp.com
Позволяет делать cors-запросы, когда браузер блокирует их.

htmldom.dev
Очень годный ресурс с готовыми решениями вопросов по DOM

rajdee.gitbooks.io
Годно про Redux на русском



HTML, CSS
loading.io
Готовые индикаторы загрузки

www.w3schools.com
Простой спиннер

bootswatch.com
Темы для Bootstrap

heropatterns.com
svg-фоны для html.



NPM
$http-server
```

Стандартные встроенные объекты

Глобальные функции
eval()
uneval()
isFinite()
isNaN()
parseFloat()
parseInt()
decodeURI()
decodeURIComponent()
encodeURI()
encodeURIComponent()

Числа и даты
Number
Math
Date

Обработка текста
Объекты для манипулирования текстом.
String

RegExp

Структурированные данные
Буферы данных и Объектная нотация JavaScript.
ArrayBuffer (en-US)
DataView (en-US)
JSON

Объекты управляющих абстракций
Promise

Прочее
Аргументы функции (arguments)

</>

switch

Конструкция switch заменяет сразу несколько if. Она имеет один или более блок case и необязательный default. Несколько значений case можно группировать.

Переменная «x» поочерёдно проверяется на строгое равенство значению в первом case, затем значению во втором case и так далее.
Если равенство установлено, то выполнится и текущая директива, и все остальные директивы, расположенные ниже (без проверок). Чтобы выполнилась только текущая директива, после её инстуркций надо установить оператор **break**.
Если ни один case не совпал, то выполнится вариант default, если его указали.

```
const letter = 'a'

switch(letter) {
  case 'a':                // if (letter === a)
    console.log('Alpha');
    break;

  case 'b':
  case 'c':
  case 'd':
    console.log('Other');
    break;

  default:
    console.log("Nothing");
    break;
}

// Alpha
```

Если упустить break, то будет так:

```
const letter = 'a'

switch(letter) {
  case 'a':
    console.log('Alpha');

  case 'b':
  case 'c':
  case 'd':
    console.log('Other');
```

```
  default:
    console.log("Nothing");
}

// Alpha
// Other
// Nothing
```

Дестракчеринг

Документация здесь.
Деструктуризация (дестракчеринг) — специальный синтаксис, позволяющий извлекать части из составных данных. Можно использовать как синтаксический сахар для объявления переменных, обмена значениями и т.д.

Деструктуризация массива

Общий паттерн такой:

```
let [переменная, переменная] = массив[элемент1, элемент2]

let [a, b] = [1, 2, 3];

ненужные элементы могут быть пропущены
let[c, , d] = [1, 2, 3];

сразу в свойства объектов
[user.name, user.surname] = "Ilya Kantor".split(' ');

деструктурировать можно любой перебираемый объект
let [one, two, three] = new Set([1, 2, 3]);
let [a, b, c] = "abc";

Значения по умолчанию
let [name = "Guest", surname = "Anonymous"] = ["Julius"];

Обмен значениями
let a = 1;
let b = 2;
[a, b] = [b, a];

обойти объект по-взрослому
for (const [key, value] of Object.entries(obj) ) {
  console.log(key);
  console.log(value);
}
```

Деструктуризация объекта

Ну вот:

```
const/let { <ключ объекта>: <переменная>, в которую надо сохранить значение ключа} = объект

ключ объекта
Наименование существующего ключа в объекте, из которого будет скопировано значение
переменная
Создаваемая переменная, в которую будет скопировано значение из объекта
```

```
Наименование ключа и переменной совпадают:
let { width, height } = options


Значения по умолчанию
let {width = 100, height = 200} = options
let {incomingProperty: varName = defaultValue} = options


…Rest и дестракчеринг
Можно взять из объекта несколько необходимых свойств, а все остальные присвоить куда-нибудь
let options = { title: "Menu", height: 200, width: 100 };
let {title, ...rest} = options;


Можно присваивать в уже существующие переменные.
let title, width, height;
({title, width, height} = {title: "Menu", width: 200, height: 100});

Внимание на скобки, без скобок работать не будет.
JS обрабатывает {...} как самостоятельный блок кода, но на самом-то деле у нас
деструктуризация. Чтобы показать JavaScript, что это не блок кода, можно заключить выражение в
скобки ( {…} = {…} )
```

Деструктуризация параметров функции

Когда мы передаём в функцию аргумент при вызове, его значение присваивается формальному параметру функции. Это неявное автоматическое присвоение:

```
const foo = (x) => {
  // let x = arg
};
```

Для аргументов-объектов

Типичная ситуация, когда на вход функции приходит объект с большим количеством свойств. В таких случаях надо забрать только нужные значения:

```
const func = ({ name, surname } = {}) => {
  console.log(name);
  console.log(surname);
};

// внутри происходит это:
// let { name, surname } = { name: 'John', surname: 'Doe' };
```

Для аргументов-массивов

```
const func = ([first, second]) => { … };
const func = ([first = 1, second = 2]) => { … };
```

Прочая хуйня по деструктуризации

Дестракчеринг массива можно использовать в циклах:

```
const points = [ [4, 3], [0, -3] ];
for (const [x, y] of points) {
  console.log([x, y]);
}
// => [ 4, 3 ]
```

```
// => [ 0, -3 ]


То же самое для структуры map:
let user = new Map();
user.set("name", "John");
user.set("age", "30");

for (let [key, value] of user) {
  console.log(`${key}:${value}`); // name:John, затем age:30
}
```

Деревья и дестракчеринг массива
Если при дестракчеринге указать переменную так, что ей не будет соответствовать ни один элемент массива, то в переменную запишется значение undefined. Это можно использовать при работе с деревьями:

```
const node = ['a', ['b']];

// присваивание в какой-то функции
const [name, children] = node;

// b эту проверку не пройдёт, потому что он лист и у него нет потомков
if (!children) {
  return [name];
  }
```

Умные параметры функций
Если функция имеет много параметров, то вот так – плохой способ их объявлять:

```
function showMenu(title = "Untitled", width = 200, height = 100, items = []) { … }
```

1. невозможно запомнить порядок, в котором эти параметры передаются в функцию;
2. чтобы какие-то параметры в середине использовали значение по умолчанию, им надо будет вручную прописывать undefined:

```
showMenu("My Menu", undefined, undefined, ["Item1", "Item2"])
```

Эти проблемы решаются деструктуризацией. Нужно сделать так, чтобы функция на вход принимала объект из параметров.
Обратите внимание, что такое деструктурирование подразумевает, что в функцию обязательно будет передан объект. Если нам нужны все значения по умолчанию, следует передать пустой объект, а лучше сделать его передачу по умолчанию. Если весь объект аргументов по умолчанию равен {}, то всегда есть что-то, что можно деструктурировать.

Самый простой вариант реализации:

```
// объект с параметрами
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};

// функция извлекает свойства в переменные
function showMenu({title = "Untitled", width = 200, height = 100, items = []} = {}) {

  console.log( `${title} ${width} ${height}` );
  console.log( items );
}
```

```
showMenu(options);
// My Menu 200 100
// Item1, Item2
```

Более сложное деструктурирование со вложенными объектами и двоеточием:

```
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};


function showMenu({
  title = "Untitled",
  width: w = 100,         // width присваиваем в w
  height: h = 200,        // height присваиваем в h
  items: [item1, item2]   // первый элемент items присваивается в item1, второй в item2
} = {}) {

  console.log( `${title} ${w} ${h}` ); // My Menu 100 200
  console.log( item1 );  // Item1
  console.log( item2 );  // Item2
}

showMenu(options);
```

Вложенная деструктуризация
Если объект или массив содержит другие вложенные объекты или массивы, то мы можем использовать более сложные шаблоны с левой стороны, чтобы извлечь более глубокие свойства.

```
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
};

let {  size: {width, height}, items: [item1, item2], title = "Menu"} = options;

console.log(title);  // Menu
console.log(width);  // 100
console.log(height); // 200
console.log(item1);  // Cake
console.log(item2);  // Donut
```

Комбинирование деструктуризации
Деструктуризацию объекта можно комбинировать с деструктуризацией массива.
На практике эта возможность (именно глубокая деструктуризация сложных структур) используется очень редко, потому что такой код достаточно сложен для восприятия.

```
const x = { o: [1, 2, 3] };
const { o: [a, b, c] } = x;

console.log(a); // => 1
console.log(b); // => 2
console.log(c); // => 3
```

```
const y = { o: [[1, 2, 3], { what: 'WHAT' }] };
const { o: [[one, two, three], { what }] } = y;

console.log(one);   // => 1
console.log(two);   // => 2
console.log(three); // => 3
console.log(what);  // => 'WHAT'
```

Rest, Spread

Оператор «...» выполняет различные действия в зависимости от того, где применяется.
По сути, spread осуществляет итерацию. Поэтому он работает со всеми с итерируемыми объектами:

```
let str = "Привет";
alert( [...str] ); // П,р,и,в,е,т
```

Rest у функций документация здесь.
Spread документация здесь.

Функции, Rest и Spread

```
При объявлении функции
…rest упакует лишние аргументы в массив.
Rest может быть только один и обязательно в конце.
function foo(...rest) {…}
const func = (a, b, ...params) => {…}


При вызове функции
…spread извлекает элементы.
Может быть любое количество spread-операторов, они могут располагаться в любом порядке.
foo(...arr);
Math.max(...arr1, ...arr2);
```

arguments
К аргументам функции можно обращаться и по-старому — через псевдомассив arguments.
Отличие …rest от arguments (дубликат из раздела про функции):
Существует три основных отличия оставшихся параметров от объекта arguments:

1. оставшиеся параметры включают только те, которым не задано отдельное имя, в то время как объект arguments содержит все аргументы, передаваемые в функцию;
2. объект arguments не является массивом, в то время как оставшиеся параметры являются экземпляром Array;
3. объект arguments имеет дополнительную функциональность (например, свойство callee).
4. Стрелочные функции не имеют arguments.

См. также раздел по функциям с arguments.

Массивы, Rest и Spread

```
Rest
Разложить на первый элемент и все остальные
const [first, second, ...rest] = ['apple', 'orange', 'banana', 'pineapple'];

Если интересует только часть массива, то slice() лучше
```

```
const rest = fruits.slice(1);


Spread
Слияние нескольких массивов
let merged = [0, ...arr, 2, ...arr2];

Конвертация в массив
[...str]
[...args]
```

Array.from — более универсальный метод:
- Array.from работает как с псевдомассивами, так и с итерируемыми объектами
- Оператор расширения работает только с итерируемыми объектами

Объекты и Spread

```
Разложить объект на список пар «ключ:значение», можно делать копию объекта
const a = { key1: 'value1' };
const b = { ...a };


Запись новых и перезапись существующих свойств
const a = { key1: 'value1' };
const b = { ...a, newKey: 'newValue' };


Слияние объектов
const a = { key1: 'value1' };
const b = { key2: 'value2' };
const c = { ...a, ...b }


Обновление свойства объекта без мутации
const object = { key1: value1, key2: value2 };
const newObject = { ...object, key2: newValue };
```

</>

Функции

Функция – это фрагмент кода для выполнения определённых задач, к которому можно обратиться из другого места программы.
Функция – способ группировки команд.

Обычные значения, такие как строки или числа представляют собой *данные*. Функции можно воспринимать как «действия».

См. также:
Деструктуризация параметров функции

Обычные функции

Есть 2 способа объявить обычную функцию: declaration и expression.

```
```

```
declaration
всплывает, работает super
function foo(arg) {
  return arg;
}

expression
const foo = function(arg) {
  return arg;
};

named Function Expression
это для того, чтобы FE стал локальным и мог вызывать сам себя
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
};
```

Разница между ними:
1. Declaration всплывают
2. Declaration в классах имеет скрытое свойство [[HomeObject]]. Благодаря этому работает super, он в его прототипе ищет родительские методы.
3. У Expression ставится точка с запятой на конце, а у Declaration нет.

Пояснения по разнице (можно пропустить)
Когда движок JavaScript готовится выполнять код, он ищет в нём Function Declaration и создаёт их, поэтому они могут быть вызваны раньше своих определений. Expression создаются только в тот момент, когда выполнение доходит до них и только после этого могут использоваться.
Всплытие, поднятие, hoisting – это способность функций, объявленных через function declaration, быть вызванными ещё до их объявления.
Expression использует внутри себя инструкции присваивания let sayHi = ...; как значение. Это не блок кода, а выражение с присваиванием. Таким образом, точка с запятой лишь завершает инструкцию.

Есть ещё **Named Function Expression**
Делается это для того, чтобы FE стал локальным и мог вызывать сам себя. См. примеры ниже в «Объект функции».

Стрелочные функции

Анонимны, если их не присвоить какой-то переменной.
Мануал здесь.

```
однострочные:
() => {}
const foo = (argument) => argument;

многострочные
const foo = (argument) => {
  return argument;
};
```

Особенности стрелочных функций:

```
- не имеют super
- не имеют «arguments»
- привязаны к значению this (контекст) того места, где объявлены, а не вызваны.
- нельзя использовать как конструкторы с new (следствие отсутствия this).
```

Создание методов объектов

Методы объектов создаются такими бесстрелочными функциями:

```
const company = {
  name: 'Hexlet',

  // сокращённая запись метода
  getName() {
    return this.name;
  }

  getName: function getName() {
    return this.name;
  },
};
```

Доступ к переменным

Переменные, объявленные внутри функции, видны только внутри этой функции.
Функция обладает доступом к внешним переменным и может изменять их значение. Если одноимённая переменная объявляется внутри функции, тогда она перекрывает внешнюю.

Параметры функции

Мы можем передать внутрь функции любую информацию, используя параметры (аргументы) функции.

Передаваемые значения копируются в параметры функции и становятся локальными переменными. Функция получает именно копию.

```
const foo = (n) => {
  n = 'abc';     // перезаписывает параметр
  return n;
}

let a = 2;
foo(a);
console.log(a);  // 2 - значение не изменилось
```

Функции-коллбеки

Можно передавать функции как параметры. Идея в том, что мы передаём функцию и ожидаем, что она вызовется обратно (от англ. «call back» – обратный вызов) когда-нибудь позже, если это будет необходимо. Например, задаётся вопрос, и в зависимости от варианта ответа вызывается та или иная функция.

Бросать коллбеки в функцию можно прям так, они всё равно будут вызываться:

```
const promise = ...
.catch(console.log);
```

Таймеры setTimeout и setInterval

Для планирования вызова существуют два метода:

setTimeout	вызвать функцию **один раз** через определённый интервал времени.
clearTimeout	отменить вызов

```
setInterval     вызывать функцию регулярно, повторяя вызов через определённый интервал времени.
clearInterval   отменить вызовы
```

Эти методы не являются частью спецификации JavaScript. Но большинство сред выполнения JS-кода имеют внутренний планировщик и предоставляют доступ к этим методам.

Функция, переданная в таймер, выполняется не в текущем стеке вызовов, поэтому к таймерам применимы все особенности АФ. try/catch не ловит ошибки. Ловить ошибки надо с помощью коллбеков.
При этом таймер не делает операцию, которая в него передана, асинхронной. Таймер только откладывает время её выполнения. Если сама операция синхронная, то после запуска она заблокирует основной поток выполнения программы, и все остальные будут ждать её завершения.

Таймеры срабатывают либо по заданному времени, либо с задержкой. Рантайм проверяет таймеры, когда в текущем стеке вызовов всё выполнено. Если происходит тяжелое вычисление, то колбеки и таймеры будут ждать, пока вычисление закончится. Время срабатывания сдвигается в большую сторону, поэтому справедливо говорить, что в таймерах задается минимальное время, после которого будет запущен код.

setTimeout в браузере устанавливает this=window для вызова функции. Для методов в отрыве от объекта он пытается получить window.метод, которого не существует.
В Node.js this становится объектом таймера.

setTimeout()

Вызов функции или выполнение фрагмента один раз через определённый интервал времени.
MDN.

```
let timerId = setTimeout(func|code, [delay], [arg1], [argN])

func|code
Функция или строка кода для выполнения.
delay
Задержка перед запуском в миллисекундах (1000 мс = 1 с). Значение по умолчанию – 0.
arg1, argN…
Аргументы, передаваемые в функцию (не поддерживается в IE9-)
```

Частая ошибка – передать в таймер не саму функцию, а её вызов.
Это не работает, потому что setTimeout ожидает ссылку на функцию.

Передать данные внутрь функции можно тремя способами:

```
const f = (a, b) => console.log(a + b);


1. дописать параметры в setTimeout
setTimeout(f, 1000, 5, 8);

2. функция-обёртка
setTimeout(() => foo(5, 8), 1000);

3. bind, частичное применение, первый параметр null потому что контекст не меняется
setTimeout(f.bind(null, 5, 8), 1000);
```

clearTimeout (id)
Вызов **setTimeout** возвращает «идентификатор таймера» timerId, который можно использовать для отмены дальнейшего выполнения. В браузере идентификатор таймера это числовое значение, в node.js это объект.
Синтаксис для отмены таймера:

```
let timerId = setTimeout(f, 1000); // таймер запущен
clearTimeout(timerId);             // отменён
```

setInterval()

setInterval автоматически запускает функцию через указанный промежуток времени до тех пор, пока её явно не остановят через clearInterval. Время между запусками равно переданному второму параметру.

Имеет такую же сигнатуру, как и setTimeout:

```
let timerId = setInterval(func|code, [delay], [arg1], [argN])
```

clearInterval (id)

Остановить setInterval можно через **clearInterval**, передав ему ID таймера.

Во время показа alert (в смысле, во время всплытия окошка) время тоже идёт и не ставится на паузу.
В большинстве браузеров, включая Chrome и Firefox, внутренний счётчик продолжает тикать во время показа alert/confirm/prompt.
Если подождать с закрытием alert несколько секунд, то следующий alert будет показан сразу, как только вы закроете предыдущий.

Использование двух таймеров вместе:
- setInterval каждые 5 секунд выполняет печать
- setTimeout через 16 секунд выключает интервальный таймер через clearInterval

```
const id = setInterval(() => console.log(new Date()), 1000);
setTimeout(() => clearInterval(id), 4000);  // автоматом отключить через 5 сек

// 2020-05-27T15:01:46.418Z
// 2020-05-27T15:01:47.424Z
// 2020-05-27T15:01:48.424Z
```

Таймер можно остановить изнутри, передав в колбек его *id*.

```
let counter = 0;

const foo = () => {
  counter += 1;

  if (counter === 4) {
    clearInterval(id);
    return;
  }

  console.log(new Date());
};


const id = setInterval(foo, 2000);
```

Рекурсивный setTimeout

Запускать что-то регулярно можно через интервальный setInterval или рекурсивный setTimeout.
Рекурсия проявляется не через return, а в том, что таймер через какое-то время вызывает такой же таймер.

```
```

```
// вместо setInterval

let timerId = setTimeout(function tick() {
  alert('tick');
  timerId = setTimeout(tick, 2000);

}, 2000);


// То же самое, только функция объявлена снаружи

function foo() {
  console.log('tick');
  id = setTimeout(foo, 1000);
}

let id = setTimeout(foo, 1000);
```

Рекурсивный setTimeout – более гибкий метод, чем setInterval. С его помощью последующий вызов может быть задан по-разному в зависимости от результатов предыдущего.
Например, необходимо написать сервис, который отправляет запрос для получения данных на сервер каждые 5 секунд, но если сервер перегружен, то необходимо увеличить интервал запросов до 10, 20, 40 секунд… Вот псевдокод:

```
let delay = 5000;

let timerId = setTimeout(function request() {
  // ...отправить запрос...

  if (ошибка запроса из-за перегрузки сервера) {
    // увеличить интервал для следующего запроса
    delay *= 2;
  }

  timerId = setTimeout(request, delay);

}, delay);
```

Можно запускать функции через 2 секунды строго после их выполнения. SetInterval запустит функцию и сразу же начнёт отсчёт для запуска новой, не дожидаясь окончания старой. SetTimeout можно установить так, что отсчитывать время он будет только после окончания операции:

```
let i = 1;

setTimeout(function run() {
  func(i);
  setTimeout(run, 100);
}, 100);
```

Рекурсивный setTimeout гарантирует фиксированную задержку не между запусками, а между исполнением и новым запуском функции (здесь 100 мс). Это потому, что новый вызов планируется в конце предыдущего.

```
const foo = () => {
  let i = 0;
  while (i < 1000000000) {
    i += 1;
  }
  console.log('!')
  let id = setTimeout(foo, 100);
```

```
  return
};

foo()
```

setTimeout с нулевой задержкой

Особый вариант использования: setTimeout(func, 0) или просто setTimeout(func).
Это планирует вызов func настолько быстро, насколько это возможно. Но планировщик будет вызывать функцию только после завершения выполнения текущего кода (стека).

```
setTimeout(() => console.log('привет'))

const foo = () => {
  let i = 0;
  while (i < 1000000000) {
    i += 1;
  }
  console.log('!')
  return
};

foo()

// !
// привет
```

Есть и более продвинутые случаи использования нулевой задержки в браузерах, которые мы рассмотрим в главе Событийный цикл: микрозадачи и макрозадачи.

Контекст, call, aply, bind

Это общая тема для разделов из учебника «переадресация вызова» и «привязка контекста».

При передаче методов объекта в качестве колбэков, например для setTimeout, возникает проблема – потеря this.
Это может быть в следующих случаях:
1. передача метода отдельно от объекта;
2. использование таймера. Таймер в браузере устанавливает this=window для вызова функции. В Node.js this становится объектом таймера. SetTimeout получает функцию отдельно от объекта и теряет контекст.

Примеры таких потерь:

```
let user = {
  firstName: "Вася",
  sayHi() {
    console.log(`Привет, ${this.firstName}!`);
  }
};

const a = user.sayHi;
a();  // Привет, undefined!

setTimeout(user.sayHi, 100); // Привет, undefined!
```

Функции для переадресации вызова

Передать контекст явно и переадресовать вызов могут методы .call, .apply и .bind.

```
const res = foo.call(this, x, y);
аргументы через запятую и сразу же выполняется

const res = foo.apply(this, [x, y]);
аргументы в массиве и сразу же выполняется

const res = (foo.bind(this, x, y))();
аргументы через запятую, но не выполняется
```

Пример:

```
let myObject = {
  name: 'myObjectect',
  method() {
    console.log(this.name);
  }
};

const a = myObject.method;
a();              // undefined
a.call(myObject); // correct
```

Стрелочная функция-обёртка

Стрелочная функция запоминает окружение, в котором была создана. Таким образом создаётся замыкание:

```
const obj = {
  foo() {
    console.log(this);
  }
}

setTimeout( () => obj.foo(), 1 ); // правильныйобъект
setTimeout( obj.foo, 1 );         // Timeout { ... }
```

В данном стрелочная функция замыкается на глобальном окружении и объект «obj» достаётся из него. В коде появилась уязвимость: до момента срабатывания setTimeout в переменную может быть записано другое значение. .bind гарантирует, что такого не случится.

.bind

Метод создаёт новую функцию и устанавливает ей контекст выполнения this. Кроме того, к функции можно применить все или часть аргументов (частичное применение).
В отличие от call и apply, bind не вызывает функцию сразу же, а возвращает "обёртку", которую можно вызвать позже.
Документация здесь.

Привязка выполняется только один раз и затем её нельзя отменить. Это связано с тем, что .bind возвращает экзотический объект bound function, возвращаемый при первом вызове. Следующий вызов bind будет устанавливать контекст уже для этого объекта. Это ни на что не повлияет.

```
let boundFoo = foo.bind(this);
let boundFoo = foo.bind(this, arg1, arg2...);

this
Значение, передаваемое в качестве this в целевую функцию при вызове привязанной функции

arg1, arg2...
```

```
Аргументы целевой функции («предустановленные»), передаваемые перед аргументами
привязанной функции
```

Методы объектов фиксируются так:

```
const obj = {
  name: 'correct',
  method() {
    console.log(this.name)
  }
};

const bindMethod = obj.method.bind(obj)
bindMethod(); // correct
```

Массовая привязка всех методов к объекту
Если у объекта много методов и их надо активно передавать куда-то ещё, то можно привязать их контекст в цикле. И они всё ещё остаются методами исходного объекта и исправно работают с ним.

```
for (let key in user) {
  if (typeof user[key] == 'function') {
    user[key] = user[key].bind(user);
  }
}
```

Некоторые JS-библиотеки предоставляют встроенные функции для удобной массовой привязки контекста, например _.bindAll(obj) в lodash.

Частичное применение

Когда мы привязываем аргументы, такая функция называется «частично применённой» или «частичной». Частичное применение удобно, когда мы не хотим повторять один и тот же аргумент много раз.

Контекст – это обязательный параметр для bind, так что нужно передать туда что-нибудь вроде null, это такая фикция. В примере ниже, функции multi не нужен контекст this, она работает сама по себе, поэтому нет смысла что-то указывать первым аргументом в качестве контекста.

```
const multi = (a, b) => a * b;
const double = multi.bind(null, 2);

console.log( double(3) ) // 6
```

Частичное применение для методов объекта
Иногда надо зафиксировать только аргументы без контекста. Null в данном случае не подойдёт, потому что методы могут использовать this. Можно создать такую вспомогательную функцию partial, которая привязывает только аргументы:

```
function partial(foo, ...agrsBound) {
  return function(...args) {
    return foo.call(this, ...agrsBound, ...args);
  }
}


let object = {
  name: 'Ivan',
  say(time, phrase) {
    console.log(`[${time}] ${this.name}: ${phrase}!`);
```

```
  }
};

const date = new Date();
// добавить новый метод в объект
// часы и минуты из даты - это 1-й аргумент, который $time
object.sayNow = partial(object.say, date.getHours() + ':' + date.getMinutes());

// 2-й аргумент - это phrase. Его надо добавить
object.sayNow('Hello')
// [15:24] name: Hello!
```

В этом примере используется call, а не bind, потому что call вызывается сразу же.
Также есть готовый вариант _.partial из библиотеки lodash.

JS - Объекты

Документация – здесь (объект как конструктор).
Статья в руководстве о работе с объектами здесь.

Объект (словарь, ассоциативный массив) – коллекция пар «ключ-значение», где каждый ключ уникален. Это набор свойств, и каждое свойство состоит из имени и ассоциированного с этим именем значения. Свойство объекта можно понимать как переменную, закрепленную за объектом.

Если значением свойства является функция, то её называют методом объекта. Различие свойства/метода это не более чем условность.

В качестве ключей – только строки и символы.
Другие типы данных будут автоматически преобразованы к строке. Зарезервированные слова разрешено использовать как имена свойств, такого ограничения нет. Но есть специальное свойство __proto__, которое по историческим причинам имеет особое поведение.

В качестве значений – любые типы данных.

Одним из фундаментальных отличий объектов от примитивных типов данных является то, что они хранятся и копируются «по ссылке». Переменная хранит не сам объект, а его «адрес в памяти», другими словами «ссылку» на него. См. раздел «Копирование объекта».
В этой связи, объект, объявленный через const, может быть изменён.
Объявление const защищает от изменений только само значение переменной. В случае с объектом, значение константы – это ссылка на объект, и это значение мы не меняем. Изменяя объект, мы действуем внутри объекта, а не переназначаем переменную.

Где использовать?
Объект подходит для хранения и обработки разнотипных данных, которые, как правило, описывают какую-то сущность. (Массив предназначен для хранения и обработки коллекций однотипных элементов.)
Кроме того, объекты используются как хранилища для конфигурационных параметров или как способ передать в функцию множество разнородных данных в виде одного параметра.

Синтаксис

Создание объекта

Документация здесь.
Объекты могут быть созданы с помощью:

```
{ key: value }   литеральной (инициирующей) нотации

new Object()
new Object({ key: value })

Object.create( Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj) );

Object.assign(target, source)
Object.fromEntries([ key, value ])
```

Литеральная:
Внутри скобок через запятую перечисляются пары «ключ-значение» в формате key: value.
Последнее свойство объекта может заканчиваться запятой. Это называется «висячая запятая». Но в json придётся запятую убрать, он с ней не работает.

Создание объекта через конструктор new Object()

```
let myCar = new Object();
let myCar = new Object({make: Ford, year: 1998,});
```

Создание объекта с помощью свойства Object.create()
Создаёт пустой объект со свойством [[Prototype]], указанным как proto, и необязательными дескрипторами свойств descriptors.

```
Object.create(proto[, propertiesObject])

proto
Объект, который станет прототипом вновь созданного объекта
propertiesObject
Необязательный. дескрипторы свойств.
```

Создание методов

Функцию, которая является свойством объекта, называют методом этого объекта.
Эти две записи не эквивалентны (см. функции):

```
const company = {
  name: 'Hexlet',

  // сокращённая запись метода
  getName() {
    return this.name;
  }

  getName: function getName() {
    return this.name;
  },
};
```

Копирование объектов

Примитивные типы: строки, числа, логические значения – присваиваются и копируются «по значению». В результате мы имеем две независимые переменные.

Переменные с объектами хранят не сам объект, а его «адрес в памяти», ссылку.
Когда переменная объекта копируется – копируется ссылка, сам же объект не дублируется. В результате две переменные ссылаются на один и тот же объект.

Копировать объект можно следующими способами:

```
{ …spread }
Object.assign(target, source)
Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj));
Object.fromEntries([ key, value ])
```

...spread

```
let copy = {...original};
```

Object.assign
Создаёт новый объект путём копирования значений всех собственных перечислимых свойств из одного или более исходных объектов в целевой объект.

```
Object.assign(target, ...sources)

var object = { key1: 'value1', key2: 'value2' };
var copy = Object.assign({}, object);


let user = { name: "Иван" };
let permissions1 = { canView: true };
let permissions2 = { canEdit: true };

Object.assign(user, permissions1, permissions2);
копируем все свойства в user
{ name: "Иван", canView: true, canEdit: true }
```

Object.create
Вызов создаёт точную копию объекта obj, включая все свойства: перечисляемые и неперечисляемые, геттеры/сеттеры для свойств – и всё это с правильным свойством [[Prototype]].

Тут комбинация сразу трёх методов: Object.create, Object.getPrototypeOf, Object.getOwnPropertyDescriptors

```
let clone = Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj));
```

Собрать новый объект из ключей и значений:

```
function getObjectCopy(originalObject) {

  let copy = {};
  for (let key of Object.keys(originalObject)) {
    copy[key] = originalObject[key];
  }

  return copy;
};
```

Через метод _.cloneDeep из библиотеки lodash:

```
// import _ from 'lodash';
import { cloneDeep } from 'lodash';
```

```
let original =  { key1: 'value1', key2: 'value2' };
let copy = cloneDeep(original);
```

Если объекты содержат в себе другие объекты, то простого копирования с ключами и свойствами не достаточно для создания независимой копии. Для этого надо использовать _.cloneDeep из библиотеки lodash или самому написать рекурсивную функцию. Такая функция должна проверять, является ли значение по ключу объектом, и если да – запускаться для него рекурсивно. Это называется «глубокое копирование».

Обращение к свойствам

Существует 2 способа доступа к элементам объекта:

```
точечная нотация
myObject.key1

скобочная нотация (см. property accessors)
myObject['key1']
```

Обращение к несуществующему возвращает undefined.

Через точку можно обращаться только к таким свойствам, которые начинаются с буквы, $, _.
Документация здесь и здесь.

Квадратные скобки дают больше возможностей, чем точечная нотация.
Если свойство состоит из нескольких слов через проблел:

```
let user = { "likes birds": true };
user['likes birds']
```

Если свойство было записано как число, к нему можно обратиться только так:

```
const myObject = {1: 'value1' };
myObject['1']
```

В квадратных скобках можно указать переменную, в которой хранится имя свойства для обращения:

```
const a = 'key1';
const object = { key1: 'value 1' };

object.a   ничего не произойдёт
object[a]  value 1
```

Изменение свойств

Чтобы изменить существующее свойство, нужно обратиться по ключу и изменить значение.

Удаление свойств
Удалить элемент из объекта можно с помощью оператора delete.

```
delete myObject['key2'];
```

Добавление свойств
через нотации:

```
myObject['newKey'] = 'new value';
myObject.newKey = 'new value';
```

Для создания пары свойство-значение, достаточно просто передать в объект переменную:

```
const x = 10;
const y = 20;

const obj = {x, y}
      obj = { x: 10, y: 20 }
```

Хороший пример использования этой фишки:

```
function makeUser(name, age) {
  return {
    name, // name: name
    age,  // age: age
  };
}
```

Присоздании объекта, можно указать наименование ключа из переменной:

```
const n = 'name';
const m = 'married';
const a = 'age';

const user = { [n]: 'Vasya', [m]: true, [a]: 25 };
// {
  name: 'Vasya',
  married: true,
  age: 25
  }
```

Можно делать префиксы:

```
const prefix = '_pre_';

const obj = {
  [prefix + 'first']: 'Glad',
  [prefix + 'second']: 'Valakas'
};

{
  _pre_first: 'Glad',
  _pre_second: 'Valakas'
}
```

Проверить наличие свойства

Оператор in

Оператор in возвращает true, если свойство содержится в указанном объекте или в его цепочке прототипов.

- Если свойству присвоено значение undefined, то для этого свойства in вернет значение true.
- Для свойств, которые унаследованы по цепочке прототипов, тоже возвращается true. Если вы хотите проверить только не наследованные свойства, используйте Object.hasOwnProperty().
- Ищет свойство в прототипе.
- Если нет кавычек, то значит указана переменная, в которой находится имя свойства.

```
"key" in object


let user = { name: "John", age: 30 };

console.log( "age" in user );    // true
console.log( "blabla" in user ); // false
```

Object.hasOwnProperty()
Возвращает true или false, указывающее, содержит ли объект указанное свойство.
Не проверяет существование свойств в цепочке прототипов объекта.

```
object.hasOwnProperty('propertyName')

const obj = { a: 1, b: 2, c: 3 };
console.log(obj.hasOwnProperty('a'))
// true
```

_.has из **Lodash**
Альтернатива встроенному методу Object.hasOwnProperty.

```
_.has(object, path)
```

Опциональная цепочка ?.

Новая фича в JS (октярь 2020) – это **опциональная цепочка** «**?.**»
Опциональная цепочка – это безопасный способ доступа к свойствам вложенных объектов.
Если обратиться к несуществующему свойству объекта, ошибки не будет и вернётся undefined.
Но если обратиться к несуществующему свойству несуществующего свойства, то скрипт упадёт.
Такая же ошибка будет, если нужно получить данные об HTML-элементе, который отсутствует на странице.
Optional chaining в MDN.

В случае использования опциональной цепочки, просто вернётся undefined.
Синтаксис имеет три формы:

```
obj?.prop
obj?.[prop]
obj.method?.()

также работает с функциями и квадратными скобками
arr?.[index]
func?.(args)
```

Вот безопасный способ обратиться к свойству user.address.street:

```
let user = {}; // пользователь без адреса

alert( user?.address?.street );
// undefined (без ошибки)
```

Чтение адреса с помощью конструкции «user?.address» выполняется без ошибок, даже если объекта user не существует:

```
let user = null;
```

```
alert( user?.address );         // undefined
alert( user?.address.street );  // undefined
```

Трансформации объекта

У объектов нет множества методов, которые есть в массивах, например map, filter и других.
Можно использовать Object.entries с последующим вызовом Object.fromEntries:

```
let prices = {
  banana: 1,
  orange: 2,
  meat: 4,
};

let doublePrices = Object.fromEntries(
  Object.entries(prices)
    .map( ([key, value]) => [key, value * 2])
);

alert(doublePrices.meat); // 8
```

Дескрипторы свойств

Свойство – это не только пара «ключ-значение». У каждого свойства есть ещё и дексрипторы. Обычно они скрыты:

```
(ключ: значение): дескрипторы
```

Просто так дексрипторы не получить и не записать. Для работы с ними нужны специальные методы:

```
Object.getOwnPropertyDescriptor()
Object.getOwnPropertyDescriptors(obj)

Object.defineProperty()
Object.defineProperties()
```

Подробная информация о типах дескрипторов свойств и их атрибутах может быть найдена в описании метода Object.defineProperty().

Дексрипторы описывают «служебные» свойства самого свойства объекта. Например, будет ли оно перезаписываемым, перечисляемым в цикле, есть ли у него сеттеры или геттеры.
Дескрипторы бывают двух основных типов: дескрипторы данных и дескрипторы доступа. Дескриптор может быть только чем-то одним из этих двух типов; он не может быть одновременно обоими.
Фактически, дескриптор – это объект с набором определённых ключей.

Если ты сам создаёшь свойства при инициализации объекта или дозаписываешь их в объект, то значение этих ключей по умолчанию – true.
Если свойства создаются через метод defineProperty, то значение флагов по умолчанию – false.

Обязательные ключи

```
const descriptor = {
  enumerable: true/false,
  configurable: true/false,
}
```

enumerable
true – свойство будет перечисляться в циклах.
false – циклы и метод Object.keys будут игнорировать свойство.

configurable
true – свойство можно удалить, а флаги можно изменять.
false:
- нельзя менять флаги этого дескриптора, в т.ч. сам configurable.
- свойство нельзя удалить (изменять value можно). Попытки сделать это приведут к ошибке в строгом режиме.
Определение свойства как неконфигурируемого – это дорога в один конец. Для некоторых встроенных объектов configurable иногда предустановлен в false. Например, для свойства Math.PI

Ключи дескриптора данных

```
const descriptor = {
  enumerable: true/false,
  configurable: true/false,

  writable: true/false
  value: <значение обычного свойства объекта>
}
```

writable
true – свойство объекта можно изменить
false – свойство только для чтения. Попытка изменить его приведёт к ошибке в строгом режиме.

value
это обычное значение свойства объекта. Его можно изменять и добавлять через дескриптор.

Дескрипторы доступа

Это функции, которые выполняют роль геттеров и сеттеров.
На практике можно реальзовать геттеры и сеттеры другими, более удобными способами. Обо всех способах написано ниже в разделе «Геттеры и сеттеры».

```
const descriptor = {
  get() {
    return `${this.name} ${this.surname}`;
  },

  set(value) {
    [this.name, this.surname] = value.split(' ');
  }
};

Object.defineProperty(myObject, 'fullName', descriptor);

myObject.fullName = 'Денис Детров';
myObject.fullName;// Денис Детров
```

Методы для дескрипторов

```
Object.getOwnPropertyDescriptor()
Object.getOwnPropertyDescriptors(obj)

Object.defineProperty()
```

```
Object.defineProperties()
```

Object.getOwnPropertyDescriptor()

Возвращает объект-дескриптор для собственного свойства объекта.
Собственное свойство – это которое находится в объекте, а не получено через цепочку прототипов.

```
descriptor = Object.getOwnPropertyDescriptor(obj, 'prop')

obj
Объект
'prop'
Это ключ, который нужно передать как строку
```

Пример:

```
const myObject = {
  key1: 'value1',
  key2: 'value2'
}

const descr = Object.getOwnPropertyDescriptor(myObject, 'key1');

{
  value: 'value1',
  writable: true,
  enumerable: true,
  configurable: true
}
```

Object.getOwnPropertyDescriptors(obj)

Получить все дескрипторы собственных свойств объекта сразу, включая свойства-символы.
Возвращается объект, в котором ключи – это наименование (ключи) исходного объекта, а значения этих ключей – объекты-дескрипторы.

```
Object.getOwnPropertyDescriptors(obj)

obj
Объект, чьи дескрипторы будут возвращены
```

Пример:

```
const myObject = {
  key1: 'value1',
  key2: 'value2'
};

const descr = Object.getOwnPropertyDescriptors(myObject);

{
  key1: {
    value: 'value1',
    writable: true,
    enumerable: true,
    configurable: true
  },
  key2: {
    value: 'value2',
    writable: true,
    enumerable: true,
```

```
    configurable: true
  }
}
```

Этот метод можно объединить с Object.defineProperties, чтобы клонировать объект вместе с дескрипторами. Обычное копирование объекта через [key, value] не копирует флаги, а так всё скопируется полностью, в т.ч. дескрипторы и свойства-символы:

```
let clone = Object.defineProperties({}, Object.getOwnPropertyDescriptors(obj));
```

Object.defineProperty()

Определяет новое или изменяет существующее свойство непосредственно на объекте.
Возвращает этот объект.
По умолчанию, ставит флагам дескриптора значение false, если их явно не прописать.

```
Object.defineProperty(obj, 'prop', descriptor)

obj
Объект, на котором определяется свойство
prop
Имя определяемого или изменяемого свойства
descriptor
Объект-дескриптор определяемого или изменяемого свойства.
```

Метод создаёт новое свойство с указанным value и флагами, или обновит флаги существующего свойства.
Можно перезаписывать все существующие свойства объекта, в т.ч. и скрытые.
Например, свойство .toString скрыто и не выводится при переборе в цикле, но к нему можно добраться через.defineProperty() и сделать так, чтобы оно было перечислимым.

Descriptor должен быть объектом, иначе будет ошибка. Например:

```
const descriptor = {
  value: 'new value',
  writable: true,
  enumerable: true,
  configurable: true,
}
```

В этом примере в объект добавляется новое свойство с флагами. Если флагам явно не прописать true, то их значением будет false:

```
const myObject = { key1: 'value1' };

const descriptor = {
  value: 'new value',
  writable: true,
  enumerable: true,
  configurable: true,
};

Object.defineProperty(myObject, 'newKey', descriptor);

myObject;
// { key1: 'value1', newKey: 'new value' }
```

Object.defineProperties()

Определяет новые или изменяет существующие свойства непосредственно на объекте, возвращая этот объект.
То же самое, что и .defineProperty, но позволяет менять флаги сразу нескольких свойств.

```
Object.defineProperties(obj, {props})


Object.defineProperties(obj, {
  prop1: descriptor1,
  prop2: descriptor2
});
```

```
const myObject = {};

const props = {
  key1: {
    value: 'value1',
    writable: true,
    enumerable: true,
    configurable: false,
  },

  key2: {
    value: 'value1',
    writable: true,
    enumerable: true,
    configurable: false,
  }
};

Object.defineProperties(myObject, props);

myObject;
// { key1: 'value1', key2: 'value1' }
```

Вместе с Object.getOwnPropertyDescriptors этот метод можно использовать для клонирования объекта вместе с его флагами. Обычное копирование объекта через [key, value] не копирует флаги, а так всё скопируется полностью, в т.ч. флаги и свойства-символы:

```
let clone = Object.defineProperties({}, Object.getOwnPropertyDescriptors(obj));
```

Глобальное запечатывание объекта

Дескрипторы свойств работают на уровне конкретных свойств.
Есть методы, которые ограничивают доступ ко *всему* объекту:

Object.preventExtensions(obj)
Запрещает добавлять новые свойства в объект.

Object.seal(obj)
Запрещает добавлять/удалять свойства. Устанавливает configurable: false для всех существующих свойств.

Object.freeze(obj)
Запрещает добавлять/удалять/изменять свойства. Устанавливает configurable: false, writable: false для всех существующих свойств.

А также есть **методы для их проверки**:
Object.isExtensible(obj)
Возвращает false, если добавление свойств запрещено, иначе true.

Object.isSealed(obj)
Возвращает true, если добавление/удаление свойств запрещено и для всех существующих свойств установлено configurable: false.

Object.isFrozen(obj)
Возвращает true, если добавление/удаление/изменение свойств запрещено, и для всех текущих свойств установлено configurable: false, writable: false.

Дескрипторы доступа

Дескриптор аксессора может иметь:
get() - функция без аргументов, которая сработает при чтении свойства
set(value) - функция, принимающая один аргумент, вызываемая при присвоении свойства,
Записываются они также через Object.defineProperty()

В пример ниже добавляется геттер/сеттер fullname. Он не перечисляется, не выводится при печати объекта, но к нему можно обращаться и использовать. После использования всего одного геттера, перезапишутся одновременно два свойства объекта: this.name и this.surname

```
const myObject = { name: 'Glad', surname: 'Valakas' };

const descriptor = {
  get() {
    return `${this.name} ${this.surname}`;
  },
  set(value) {
    [this.name, this.surname] = value.split(' ');
  }
};


Object.defineProperty(myObject, 'fullName', descriptor);

myObject.fullName = 'Денис Детров';
myObject.fullName;  // Денис Детров
```

Использование для совместимости
См. учебник тут.

Умные геттеры/сеттеры
В геттеры и сеттеры можно добавлять инструкции для проверки значений, а само значение хранить в отдельном свойстве _name. Свойство _name изначально не создаётся и к нему ничто не обращается, кроме геттера-сеттера.

Технически, внешний код всё ещё может получить доступ к имени напрямую с помощью user._name, но существует широко известное соглашение: если перед свойством стоит нижнее подчёркивание _, значит что к нему не рекомендуется обращаться извне.

```
const myObject = {
  get name() {
    return this._name;
  },

  set name(value) {
    if (value.length < 4) {
      console.log('д.б. более 4 символов');
      return;
```

```
    }
    this._name = value;
  }
};

myObject.name = "Glad";
console.log(myObject.name);
// Glad

myObject.name = "PD";
// 'д.б. более 4 символов'

console.log(myObject)
// { name: [Getter/Setter], _name: 'Glad' }
```

То же самое, пример из курсеры с дескриптором:

```
const tweet = {
  _likes: 16
};


Object.defineProperty(tweet, 'likes', {
  get: function() {
    return this._likes;
  },

  set: function(value) {
    this._likes = parseInt(value) || 0;
  }
});

tweet.likes; // 16

tweet.likes = 17;
```

Геттеры и сеттеры

Геттеры и сеттеры ещё называют свойства-аксессоры (accessor properties). По сути это функции, которые используются для присвоения и получения значения, но во внешнем коде они выглядят как обычные свойства объекта. При литеральном объявлении объекта они обозначаются get и set

Геттеры – методы для извлечения данных из объекта.
Подставить перед методом «get»
Сеттеры – методы для изменения данных в объекте
Подставить перед методом «set»

Есть несколько способов реализовать дескрипторы:

1. При литеральном создании

Они создаются как обычные свойства при литеральном создании объекта с префиксами get и set.
При использовании свойство-аксессор выглядит как обычное свойство, **не метод!**
Геттер не вызывается как функция через скобки (). Обращение к геттеру – это как обращение к обычному свойству, типа .length. При создании функции слева ставится оператор get.
Сеттер тоже не имеет скобок при использовании, ему просто присваивается значение через оператор «=». При создании слева ставится оператор set.

В примере ниже я их записал в 2 разных свойства: getFullName и setFullName для удобства. На самом деле, можно назвать их одним и тем же именем и использовать его для обеих операций:

```
const myObject = {
  name: 'Glad',

  surname: 'Valakas',

  get getFullName() {
    return `${this.name} ${this.surname}`;
  },

  set setfullName(value) {
    [this.name, this.surname] = value.split(' ');
  }

};

// работают как обычные свойства, без вызовов ()
myObject.setfullName = 'Денис Детров';
myObject.getFullName; // Денис Детров
```

2. Через дескриптор доступа

Дескриптор доступа добавляется только через Object.defineProperty() и может иметь:
get() - функция без аргументов, которая сработает при чтении свойства
set(value) - функция, принимающая один аргумент, вызываемая при присвоении свойства,

В пример ниже добавляется геттер/сеттер fullname. Он не перечисляется, не выводится при печати объекта, но к нему можно обращаться и использовать. После использования всего одного геттера, перезапишутся одновременно два свойства объекта: this.name и this.surname

```
const myObject = { name: 'Glad', surname: 'Valakas' };

const descriptor = {
  get() {
    return `${this.name} ${this.surname}`;
  },
  set(value) {
    [this.name, this.surname] = value.split(' ');
  }
};


Object.defineProperty(myObject, 'fullName', descriptor);

myObject.fullName = 'Денис Детров';
myObject.fullName;  // Денис Детров
```

3. Обычными методами

Объявляются как функции, вызываются как функции: круглые скобки и всё такое:

```
const myObject = {
  name: 'Glad',

  surname: 'Valakas',

  getFullName() {
    return `${this.name} ${this.surname}`;
  },
```

```
  setFullName(value) {
    [this.name, this.surname] = value.split(' ');
  }

};

myObject.setFullName('Денис Детров');
myObject.getFullName();
// Денис Детров
```

```
const myObject = {
  name: 'Glad',
  surname: 'Valakas',

  getFullName() {
    return `${this.name} ${this.surname}`;
  },

  setFullName(value) {
    [this.name, this.surname] = value.split(' ');
  }
};

myObject.setFullName('Денис Детров');
myObject.getFullName();
// Денис Детров
```

Фишки с объектами

Задача
Реализуйте функцию pick, которая извлекает из переданного объекта все элементы по указанным ключам и возвращает новый объект. Аргументы:
1. Исходный объект
2. Массив ключей, по которым должны быть выбраны элементы (ключ и значение) из исходного объекта, и на основе выбранных данных сформирован новый объект

```
const pick = (obj, arr) => {
  const result = {};
  const pairs = Object.entries(obj);

  for (const [key, value] of pairs) {
    if (arr.includes(key)) {
      result[key] = value;
    }
  }
  return result;
};
```

Копия массива с объектами внутри:

```
var arr = [ { key1: 'value1', key2: 'value2' } ]

var copy = arr.map(function(element) {
  var newObject = {};
  var originalObjectKeys = Object.keys(element);
  var originalObjectValues = Object.values(element);
```

```
  for (var i = 0; i < originalObjectKeys.length; i += 1) {
    var currentKey = originalObjectKeys[i];
    var currentValue = originalObjectValues[i];
    newObject[currentKey] = currentValue;
  }
  return newObject;
})
```

Пересечение по значениям свойств
Иногда надо найти из нескольких объектов такие, у которых есть общие свойства (значения свойств). Можно это сделать так:

```
const arr1 = [
  { key1: 'value1', id: 2, key2: 'value2'},
  { key1: 'value1', id: 8, key2: 'value2'},
  { key1: 'value1', id: 100, key2: 'value2'}
];

const arr2 = [
  { key1: 'value1', id: 2, key2: 'value2'},
  { key1: 'value1', id: 100, key2: 'value2'},
  { key1: 'value1', id: 7, key2: 'value2'}
];

const commonID = arr1.filter(
  (element1) => arr2.some(
    (element2) =>
    element2.id === element1.id));

[ { key1: 'value1', id: 2, key2: 'value2' },
  { key1: 'value1', id: 100, key2: 'value2' } ]
```

</>

Конструкторы объектов, "new"

Много однотипных объектов можно создать при помощи функции-конструктора и оператора "new".
Для создания сложных объектов есть и более «продвинутый» синтаксис – классы.

Функции-конструкторы являются обычными функциями. Два требования к конструкторам:
1. Имя начинается с большой буквы.
2. Функция-конструктор должна вызываться при помощи оператора "new".

Синтаксис кратко:

```
function User(name) {
  создание свойства
  this.name = name;

  создание метода
  this.read = function() {
    console.log( "Меня зовут: " + this.name );
  };
}

создание
```

```
let user = new User('Valakas');
```

Как работает?
Такой вызов через «new» создаёт пустой this в начале выполнения и возвращает заполненный в конце:
1. Создаётся новый пустой объект, и он присваивается this.
2. Выполняется код функции. Обычно он модифицирует this, добавляет туда новые свойства.
3. Возвращается значение this.

Когда нам необходимо будет создать других пользователей, мы можем использовать new User("Маша). Данная конструкция удобнее, чем каждый раз создавать литерал объекта.

Отсутствие скобок ( )
Можно не ставить скобки после new, если вызов конструктора идёт без аргументов. Пропуск скобок считается плохой практикой, но синтаксис языка такое позволяет.

```
let user = new User;
// то же, что и
let user = new User();
```

Обычно у конструкторов нет return.
Конструкторы ничего не возвращают явно, только заполняют this. Но если return всё же есть, то применяется простое правило:
1. При вызове return с объектом, будет возвращён объект, а не this.
2. При вызове return с примитивным значением, примитивное значение будет отброшено.
#? Вот это я не понял.

Проверка на вызов в режиме конструктора: new.target
Свойство new.target позволяет определить была ли функция или конструктор вызваны с помощью оператора new. MDN.
В случае, если функция вызвана при помощи new, то в new.target будет сама функция, в противном случае undefined.
Свойство new.target это мета свойство которое доступно во всех функциях. В стрелочных - new.target ссылается на new.target внешней функции.

Это можно использовать, чтобы отличить обычный вызов от вызова «в режиме конструктора».

```
function Foo() {
  if (!new.target) throw "Foo() must be called with new";
  console.log("Foo instantiated with new");
}

new Foo(); // выведет "Foo instantiated with new"
Foo(); // ошибка "Foo() must be called with new"
```

Прототипы

Прототипное наследование — это возможность языка, которая позволяет создавать одни объекты на основе других. Learn.js здесь

Объекты имеют скрытое свойство [[Prototype]], которое либо равно null, либо ссылается на другой родительский объект-прототип.

Прототипное наследование – механизм, когда отсутствующее в объекте свойство берётся из прототипа.

Перечисление всех методов и свойств из этой главы:

```
obj.__proto__
Object.create(proto, [descriptors])
Object.getPrototypeOf(obj)
Object.setPrototypeOf(obj, prototype)

Function.prototype
Function.prototype.constructor
instance.constructor
instance[[Prototype]]
```

Методы для прототипов

Свойство [[Prototype]] является скрытым, поэтому обращение к нему и изменение осуществляется через методы.

```
obj.__proto__
Это геттер/сеттер для [[Prototype]]. По спецификации __proto__ должен поддерживаться только
браузерами, но по факту его поддерживают все среды. Это API не было стандартизовано.
Внимание! Это не то же самое, что [[Prototype]].


Object.getPrototypeOf(obj)
Метод возвращает прототип (то есть, внутреннее свойство [[Prototype]]) указанного объекта.


Object.setPrototypeOf(obj, prototype)
Метод устанавливает прототип (то есть, внутреннее свойство [[Prototype]]) указанного объекта
в другой объект или null.
obj          Объект, которому устанавливается прототип
prototype    Новый прототип объекта (объект или null)


Object.create(proto, [descriptors])
Создаёт пустой объект со свойством [[Prototype]], указанным как proto, и необязательными
дескрипторами свойств descriptors.
proto         Объект, который станет прототипом вновь созданного объекта
descriptors   Необязательный. Дескрипторы создаваемых свойств.
```

В примере создаётся новый объект, которому назначается прототип и создаётся новое свойство «jumps»:

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

let rabbit = Object.create(animal, {
  jumps: {
    value: true
  }
});
```

Object.create можно использовать для копирования объекта.
Такой вызов создаёт точную копию объекта obj, включая все свойства: перечисляемые и неперечисляемые, геттеры/сеттеры для свойств – и всё это с правильным свойством [[Prototype]].

Тут комбинация сразу трёх методов:
- Object.create
- Object.getPrototypeOf
- Object.getOwnPropertyDescriptors

```
let clone = Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj));
```

Свойство [[Prototype]]

Как было сказано ранее, смысл прототипов – это наследование свойств. Если у объекта нет какого-то свойства, то движок JS ищет это свойство в прототипе объекта. У прототипа может быть свой прототип и так далее по цепочке. Объект child можно так же назначить прототипом для третьего объекта, и третьему будут доступны все свойства parent.
Такие свойства называются **унаследованными**.

Есть несколько ограничений при создании прототипов:
1. Ссылки не могут идти по кругу, JavaScript выдаст ошибку;
2. Прототип – это либо Object, либо null. Другие типы игнорируются;
3. У объекта может быть только один прототип, он не может наследовать от двух одновременно.

В примере ниже объекту child будут доступны все свойства parent:

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

let child = { jumps: true };

child.__proto__ = parent;   // назначить прототип для child
child.__proto__             // геттер прототипа


let child = {
  jumps: true,
  __proto__: parrent
};
```

В примере выше я назначил прототип вне объекта. Альтернативный способ – назначить прототип прямо в объекте, но я не уверен, что так правильно делать.

Прототип используется только для чтения свойств. Операции записи новых свойств и удаления существующих работают напрямую с объектом, но не с прототипом.
Если дочернему объекту прописать такое же свойство, какое есть у прототипа, то это свойство добавится в дочерний элемент. Свойство в прототипе не перезапишется.

Работа с this и другими операторами
Если мы вызываем obj.method(), а метод при этом взят из прототипа, то this всё равно ссылается на obj. Таким образом, методы всегда работают с текущим объектом, даже если они наследуются.

Поиск свойства и исполнение кода – два разных процесса:
Методы, использующие this, ищут свойство в дочернем объекте. Если не находят – ищут в прототипе.
Сеттеры с this записывают по общему правилу: в объект перед точкой, т.е. не в прототип.

Операция delete применяется только к собственным свойствам объекта. Если свойства нет в дочернем объекте, но есть в прототипе, то ничего не произойдёт.

Методы Object.keys, .values и прочие тоже работают только с собственными свойствами объекта.

Цикл for…in (не of!)
Цикл перебирает и собственные, и унаследованные свойства объекта.

Чтобы сделать обход через for...in с собственными ключами, можно использовать проверку методом obj.hasOwnProperty(key):

```
for (let i in child) {
  if (!child.hasOwnProperty(i)) {
    continue;
  }
  console.log(i)
}
```

Все методы, которые доступны объектам из коробки, наследуются от Object.prototype. Это «верховный» объект в иерархии. Эти методы не отображаются в цикле for..in, как другие унаследованные свойства, потому что им прописаны соответствующие дескрипторы: они неперечисляемые, внутренний флаг enumerable = false. Почти все остальные методы получения ключей/значений игнорируют унаследованные таким образом свойства.

Итого:

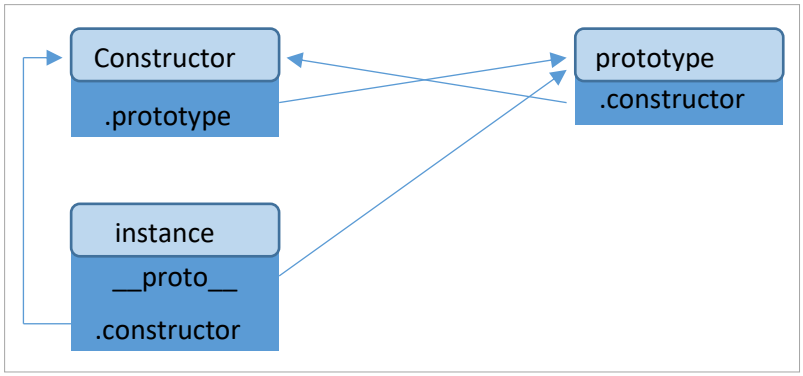
- В JavaScript все объекты имеют скрытое свойство [[Prototype]], которое является либо другим объектом, либо null.
- Объект, на который ссылается [[Prototype]], называется «прототипом».
- Если мы хотим прочитать свойство obj или вызвать метод, которого не существует у obj, тогда JavaScript попытается найти его в прототипе.
- Операции записи/удаления работают непосредственно с объектом, они не используют прототип (если это обычное свойство, а не сеттер).
- Если мы вызываем obj.method(), а метод при этом взят из прототипа, то this всё равно ссылается на obj. Таким образом, методы всегда работают с текущим объектом, даже если они наследуются.
- Цикл for..in перебирает как свои, так и унаследованные свойства. Остальные методы получения ключей/значений работают только с собственными свойствами объекта.

Конструктор и прототип

Глава в учебнике находится тут и называется Foo.prototype.
В конспекте я ещё что-то написал от себя.
Важно! Свойство функции .prototype и геттер-сеттер .__proto__ – это разные вещи. У функции есть свой прототип и он доступен через обычные геттеры.

Объекты можно создавать с помощью функций-конструкторов.
У функции-конструктора есть свойство Function**.prototype**. Это именно наименование ключа. По этому ключу хранится объект, который будет автоматически назначаться в качестве прототипа для вновь создаваемых конструктором объектов. Экземпляры объекта Function (т.е. функции-конструктора) наследуют методы и свойства из объекта Function.prototype. У обычных объектов свойства .prototype нет.
По умолчанию этот прототип имеет только одно свойство: **.constructor**. По этому ключу записана ссылка на функцию-конструктор.

MDN:
Object.prototype
Свойство представляет объект прототипа Object. В докментации написано Object, но речь идёт именно о функции-конструкторе Foo(), потому что обычные объекты не имеют свойства .prototype.
Object.constructor
Свойство возвращает ссылку на функцию Object, создавшую прототип экземпляра. Обратите внимание, что значение этого свойства является ссылкой на саму функцию, а не строкой, содержащей имя функции. Для примитивных значений, вроде 1, true или "test", значение доступно только для чтения.
Все объекты наследуют свойство constructor из своего прототипа.



Поскольку этот объект будет назначаться вновь создаваемым объектам в качестве прототипа, то все создаваемые объекты также будут иметь свойство .constructor.
Если у тебя есть какой-то инстанс и ты не знаешь, как называется его конструктор, то можно вызвать конструктор прямо из инстанса и создать новый объект:

```
function Func() {}
const instance = new Func();

Func === Func.prototype.constructor; // true
Func === instance.constructor;       // true

const instance2 = new instance.constructor();
Func === instance2.constructor;      // true
```

Конструкторы работают следующим образом:
1. прописывают создаваемому объекту будущий прототип из Function.prototype;
2. дозаписывают в создаваемый объект остальные свойства из своего тела;
3. создают новый объект.

Объект, указанный в Function.prototype, можно **заменять**.
Установка Function.prototype = object буквально говорит следующее: при создании объекта через new Function() запиши ему object в свойство [[Prototype]].
Объект по ссылке Function.prototype используется только в момент создания инстанса. Если этот объект в дальнейшем заменить на другой (т.е. указать в свойсте Function.prototype другой объект), то он по прежнему останется прототипом для старых инстансов, но не для новых.
Если прототип менялся, важно вручную прописать новому объекту-прототипу свойство .constructor, иначе не будет ссылки на функцию-конструктор.
Ещё один способ переназначить прототип для вновь создаваемых через конструктор объектов – внутрь конструктора записать this.__proto__, но я не уверен, что так правильно делать.

Рекомендуется не менять прототип после создания объекта. Движок JS хорошо оптимизирован в том случае, когда прототип не меняется. Object.setPrototypeOf или obj.__proto__= – очень медленная операция, которая ломает внутренние оптимизации для операций доступа к свойствам объекта. Так что лучше избегайте этого.

Дополнение в прототип новых свойств.
В объект в Function.prototype можно добавлять и удалять новые свойства, которые будут доступны всем ранее созданным инстансам. Этот способ предпочтительнее, чем его полная замена.

Примеры замены объекта-прототипа.

```
const parent = { eats: true, д.б. ещё свойство .constructor };

function Function(name) {
  this.name = name;
//this.__proto__ = parent;
}

Function.prototype = parent;
```

```
const instance = new Function("Name");
instance.eats; // true
```

Прототипы встроенных объектов

Все встроенные объекты следуют одному шаблону:
- Методы хранятся в прототипах (Array.prototype, Object.prototype, Date.prototype и т.д.).
- Сами объекты хранят только данные (элементы массивов, свойства объектов, даты).

Примитивы также хранят свои методы в прототипах объектов-обёрток: Number.prototype, String.prototype, Boolean.prototype. Только у значений undefined и null нет объектов-обёрток.

Встроенные прототипы могут быть изменены или дополнены новыми методами. Но не рекомендуется менять их. Единственная допустимая причина – полифил, т.е. добавление нового метода из стандарта, который ещё не поддерживается движком JavaScript.

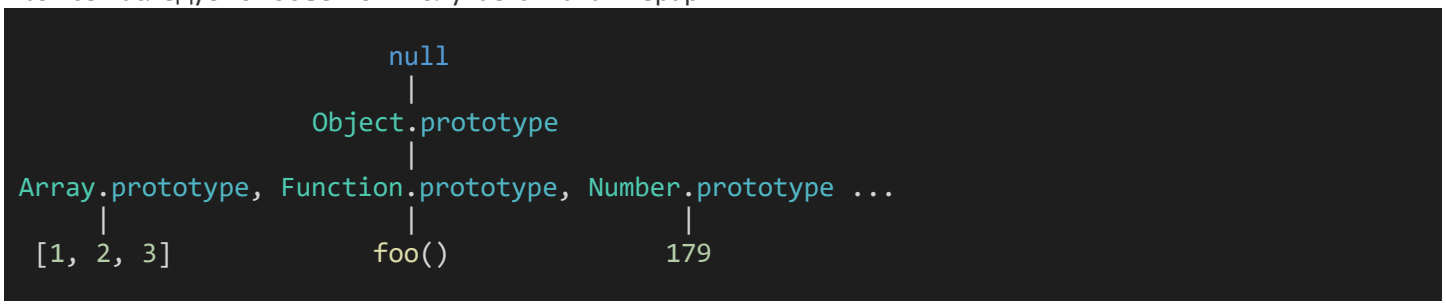
Объект всегда создаётся с помощью обычной функции-конструктора. Использование краткой нотации – это просто синтаксический сахар, всё равно работает конструктор:

```
obj = new Object()
obj = {}
```

Функция-конструктор, как обычно, имеет свойство .prototype. Этот Object.prototype – объект, который хранит все встроенные в JS функции, которые подтягивают созданные объекты. Когда вызывается new Object(), свойство [[Prototype]] этого объекта устанавливается на Object.prototype по правилам, которые обсуждены в предыдущей части. У самого Object.prototype своего .prototype нет, он равен 'null'.
В браузерных инструментах, таких как консоль разработчика, можно посмотреть цепочку наследования через console.dir.

Array, Date, Function – это всё объекты и они имеют свои конструкторы и прототипы.
Например, при создании массива [1, 2, 3] используется конструктор Array, поэтому прототипом массива становится Array.prototype, который и предоставляет свои методы.

В JS всё наследует от объектов. Получается такая иерархия:



Некоторые методы в прототипах могут пересекаться.
У Array.prototype есть свой метод toString, который выводит элементы массива через запятую, а у Object.prototype метод toString тоже есть и работает по-другому. Инстанс берёт ближайший метод, поэтому ничего страшного не происходит.

Примитивы
Хоть это и не объекты, но если мы попытаемся получить доступ к их свойствам, то тогда будет создан временный объект-обёртка с использованием встроенных конструкторов String, Number и Boolean, который предоставит методы и после этого исчезнет. Методы этих объектов также находятся в прототипах, доступных как String.prototype, Number.prototype и Boolean.prototype.
Как было сказано ранее, значения null и undefined не имеют объектов-обёрток, так что методы и свойства им недоступны. Также у них нет соответствующих прототипов.

Изменение встроенных прототипов

Встроенные прототипы можно изменять. Если что-то добавить к встроенному прототипу для строк, то все строки будут иметь этот метод. Если что-то удалить из общего прототипа, то свойство исчезнет.

Добавлять и изменять методы считается плохой практикой, потому что разные библиотеки могут переписать или добавить один и тот же метод и будет коллизия.
Единственное исключение – это полифилы.

```
if (!String.prototype.repeat) {
  // Если такого метода нет, то добавляем его в прототип

  String.prototype.repeat = function(n) {
    return new Array(n + 1).join(this);
  };
}

console.log( "La".repeat(3) ); // LaLaLa
```

на самом деле код должен быть немного более сложным, полный алгоритм можно найти в спецификации, но даже неполный полифил зачастую достаточно хорош для использования.

Заимствование методов у прототипов

В главе Декораторы и переадресация вызова, call/apply мы говорили о заимствовании методов, когда мы берём метод из одного объекта и копируем его в другой.

С прототипами есть такая же практика. Можно одолжить какие-то методы, которые есть у встроенных прототипов, для невстроенных объектов. Заимствование методов – гибкий способ, позволяющий смешивать функциональность разных объектов по необходимости.

Например, если мы создаём объект, похожий на массив (псевдомассив), мы можем скопировать некоторые методы из Array в этот объект. Например, .join, потому что для алгоритма join важны только корректность индексов и свойство length, он не проверяет, является ли объект на самом деле массивом. И многие встроенные методы работают так же:

```
let obj = {
  0: "Hello",
  1: "world!",
  length: 2,
};

obj.join = Array.prototype.join;

console.log( obj.join(',') ); // Hello,world!
```

Ультра вариант – унаследовать все вообще методы, указав в качестве прототипа объекта – Array.prototype: obj.__proto__ = Array.prototype. Но Важно помнить, что прототип у инстанса может быть только один.

Объекты без прототипа

Если хочется создать простейший объект, то можно создать объект и прописать ему прототип null.

```
let obj = Object.create(null);
```

Недостаток в том, что у таких объектов не будет встроенных методов объекта. Но обычно это нормально для ассоциативных массивов. Кроме того, большая часть методов, связанных с объектами, имеют вид свойства Object.something(...), как Object.keys(obj). Подобные методы не находятся в прототипе (они - свойства Object), так что они продолжат работать для таких объектов.

Такие чистые объекты нужны для того, чтобы избежать уязвимостей.
В случае наследования, объект получает сеттер **proto** и это представляет уязвимость, потому что кто-то может сделать так:

```
let obj = {};

let key = prompt("What's the key?", "__proto__");
obj[key] = "some value";

console.log(obj[key]);
// [object Object], не "some value"
```

Если мы присваиваем объектные значения, то прототип будет изменён. В результате дальнейшее выполнение пойдёт совершенно непредсказуемым образом. Это типа уязвимость. Неожиданные вещи могут случаться также при присвоении свойства toString, которое по умолчанию функция, и других свойств, которые тоже на самом деле являются встроенными методами.

Встроенный геттер/сеттер __proto__ не безопасен, если мы хотим использовать созданные пользователями ключи в объекте. Как минимум потому, что пользователь может ввести "__proto__" как ключ. Так что мы можем использовать либо Object.create(null) для создания «простейшего» объекта, либо использовать коллекцию Map.
</>

Udemy, прототипы

Если у нескольких объектов есть одно и то же свойство, имеет смысл его вынести в прототип. Сделать это можно несколькими способами:

1. С помощью свойства Object.setPrototypeOf().
Использование свойства Object.setPrototypeOf() очень ресурсоёмко, его не рекомендует использовать даже MDN.

```
const animal = {
  say: function() {
    console.log(`${this.name} say ${this.voice}`);
  }
};

const dog = {
  name: 'dog',
  voice: 'woof'
};

const cat = {
  name: 'cat',
  voice: 'meow'
};

Object.setPrototypeOf(dog, animal);
Object.setPrototypeOf(cat, animal);
```

2. С помощью Object.create()
Вместо этого можно использовать метод Object.create.
Из минусов – придётся переписывать все свойства вручную.

```
const animal = {
  say: function() {
    console.log(`${this.name} say ${this.voice}`);
  }
};
```

```
const dog = Object.create(animal);
dog.name = 'dog';
dog.voice = 'woof';
```

3. Через обычную функцию
Чтобы не вбивать все свойства вручную, можно сделать функцию, которая будет создавать и возвращать новый объект с использованием Object.create():

```
const animal = {
  say: function() {
    console.log(`${this.name} say ${this.voice}`);
  }
};

function createAnimal(name, voice) {
  const result = Object.create(animal);
  result.name = name;
  result.voice = voice;
  return result;
}

const dog = createAnimal('dog', 'woof');
const cat = createAnimal('cat', 'meow');
```

4. Через функцию-конструктор
Ещё удобнее будет использовать синтаксис функции-конструктора. Их имена пишутся с большой буквы, а при использовании нужно ставить ключевое слово «new». Кажется, что это то же самое, но нет:
1. JS будет оптимизировать процесс создания нового объекта, не надо использовать Object.create();
2. Ссылки на свойства создаваемого объекта (в примере выше это был result) теперь даются через ключевое слово this.property.
3. Новый объект не надо возвращать через return.
4. Методы можно дописывать вручную через свойство Animal.prototype (а можно сразу записывать в функцию-конструктор). Прототип создаётся автоматом для конструктора.

```
function Animal(name, voice) {
  this.name = name;
  this.voice = voice;
}

Animal.prototype.say = function() {
  console.log(`${this.name} say ${this.voice}`);
}

const dog = new Animal('dog', 'woof');
const cat = new Animal('cat', 'meow');
```

Тем не менее, этот паттерн не очень хороший по нескольким причинам:

- Мы показываем связи между объектами вместо того, чтобы показывать смысл. А смысл очень простой: у всех объектов Animal должно быть имя, голос и функция.
- Наследование и управление цепочками прототипов. Работая с прототипами напрямую, код для вызова функции по цепочке прототипов выше выглядит сложно и громоздко.

Синтакс классов – это синтаксический сахар, делают работу с прототипами просто более удобную.

Классы

Классы в справочнике MDN здесь.

Свойства могут записываться:
- в экземпляры класса (собственные свойства);
- в прототип (унаследованные свойства);
- в функцию-конструктор (статические свойства).

Класс в объектно-ориентированном программировании – это расширяемый шаблон кода для создания объектов, который устанавливает в них начальные значения (свойства) и реализацию поведения (методы).

На практике приходится создавать много однотипных объектов (пользователи, товары и т.д.). С этим позволяет справляться конструктор new Foo(), но есть более продвинутый синтаксис: класс. Класс – это надстройка над конструктором, которая позволяет сразу же закидывать методы в прототип создаваемых объектов, а не делать это отдельными операциями через Foo.prototype и instance._proto.
Если прототип (который _proto) при создании через конструктор имел только один метод .constructor, то в классах ему добавляются все остальные необходимые методы «из коробки».

Отличия классов от обычных конструкторов:
1. функция, созданная с помощью class, помечена специальным внутренним свойством `[[FunctionKind]]:"classConstructor"`. У созданных вручную функций такого свойства нет.

2. Методы класса являются неперечислимыми. Определение класса устанавливает флаг enumerable в false для всех методов в "prototype". В отличие от обычного конструктора.

3. Классы всегда используют use strict. Весь код внутри класса автоматически находится в строгом режиме.

4. Отличие от функций – классы не всплывают (проверить самому).

В остальном, class – это «синтаксический сахар» (синтаксис для улучшения читаемости кода, но не делающий ничего принципиально нового), потому что мы можем сделать всё то же самое без конструкции class, докинув нужные методы в прототип отдельной строкой.

Дописать
При объявлении класса, функции (методы), объявленные внутри, уходят в прототип. Если метод использует this, то, в отрыве от инстанса, он может не найти нужное свойство. Например, так теряется this.props в React.

Чтобы этого избежать, можно использовать поля класса (новая фишка) или конструктор (старая, не правильно).

```
// метод уходит в прототип и потеряет контекст
// будет работать только с bind
class TodoListItem extends Component {

  onLabelClick() {
    console.log('Done: ${this.props.label}');
  }

  render() {
    <span
      onClick={ this.onLabelCkick } />
}



// прописать в конструкторе
```

```
// всё, что указывается в конструкторе, будет храниться в инстансе, а не прототипе
class TodoListItem extends Component {

  constructor() {
    super();
    this.onLabelClick() = () => {
      console.log('Done: ${this.props.label}');
    };
  }

  render() {
    <span
      onClick={ this.onLabelClick } />
}



// поля класса
// методы, объявленные через поля, останутся в инстансе, а не прототипе
class TodoListItem extends Component {

  onLabelClick() = () => {
    console.log('Done: ${this.props.label}');
  };

  render() {
    <span
      onClick={ this.onLabelClick } />
}
```

Udemy, классы

- В классе первым делом создаётся конструктор объекта.
- Ключевое слово extends говорит о том, что данный класс включается в цепочку прототипов расширяемого класса.
- Если extends-объекту не создать свой конструктор, то он будет наследовать конструктор прототипа.

```
class Animal {
  constructor(name, voice) {
    this.name = name;
    this.voice = voice;
  }

  say() {
    console.log(`${this.name} say ${this.voice}`);
  }
}

class Bird extends Animal {
}

// duck --> Bird.prototype --> Animal.prototype --> Object.prototype --> null
const duck = new Bird('duck', 'quack');
```

Если для дочернего класса надо добавить дополнительную функциональность и ему понадобится конструктор, то внутри этого конструктора первой строкой должен сначала вызываться родительский конструктор через ключевое слово super.

```
class Animal {
```

```
  constructor(name, voice) {
    this.name = name;
    this.voice = voice;
  }

  say() {
    console.log(`${this.name} say ${this.voice}`);
  }
}

class Bird extends Animal {
  constructor(name, voice, canFly) {
    super(name, voice);
    this.canFly = canFly;
  }
}

const duck = new Bird('duck', 'quack', true);
```

Ключевое слово super даёт доступ не только к super-конструктору, но и к любому методу из super-класса.

```
class Bird extends Animal {
  constructor(name, voice, canFly) {
    super(name, voice);
    super.say();
    this.canFly = canFly;
  }
}
```

Если необходимо, в дочернем классе можно полностью переопределить метод родительского класса.
В примере ниже переопределяется метод say(). При этом сначала сработает родительский метод, к которому обратились через super, а потом дочерний, переопределённый.

```
class Animal {
  constructor(name, voice) {
    this.name = name;
    this.voice = voice;
  }

  say() {
    console.log(`${this.name} say ${this.voice}`);
  }
}

class Bird extends Animal {
  constructor(name, voice, canFly) {
    super(name, voice);
    super.say();
    this.canFly = canFly;
  }

  say() {
    console.log('birds don\'t like to talk');
  }

}

const duck = new Bird('duck', 'quack', true);
duck.say();
```

```
// duck say quack
// birds don't like to talk
```

Свойства класса

Можно инициализировать свойства прямо в теле класса.
Если свойства не зависят от внешних параметров, не имеет смысла передавать их через конструктор, можно сразу записать их в тело класса. Так же можно создавать методы.

```
class Counter {
  count = 1;

  inc = () => {
    this.count += 1;
    console.log(this.count);
  }
}


let a = new Counter();
a.inc();  // 2

let b = new Counter();
b.inc();  // 2

setTimeout(a.inc, 1000);  // 3
```

Если в примере выше создать ещё инстансы этого класса, то свойство count для каждого из них будет своим, независимым.

Статические свойства
Статические свойства и методы принадлежат всему классу. Правило простое: обычные свойства – для конкретных экземпляров, статические – для всех экземпляров.
Обращение к ним – ClassName.property

```
class Counter {
  count = 1;

  inc = () => {
    this.count += Counter.incStep;
    console.log(this.count);
  }

  static incStep = 2;

  static incrementAll = function(arr) {
    arr.forEach((instance) => instance.increment());
  };
}
```

Пример выше можно написать на старом синтаксисе, без static.
Те свойства, которые инициализируются в теле класса и принадлежат конкретному экземпляру, можно создать в конструкторе. Что касается методов экземпляров, то Arrow-функция сохранит лексический this – это сам объект, который мы создаём.
Статические поля и функции выносятся за конструктор и прописываются вручную для конструктора класса.
Функции – это тоже объекты, поэтому им можно записать свойства.

```
class Counter {

  constructor() {
    this.count = 0;
    this.increment = () => {
      this.count += Counter.incrementStep;
    }
  }
}

Counter.incrementStep = 2;

Counter.incrementAll = function(arr) {
  arr.forEach((inst) => inst.increment());
}
```

Весь синтаксис кратко

Базовый. Конструктор вызывается автоматом при создании инстанса, в него передаются аргументы, которые становятся собственными свойствами создаваемого инстанса.
Между методами не ставится запятая.

```
class MyClass {

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

  property = "Value";

  method() {
    console.log(this.name);
  }
  method2() {
    console.log(this.color);
  }
}

// Использование:
let instance = new MyClass('Иван', 'Red');
instance.method();
```

Геттеры и сеттеры

```
class MyClass {
  constructor(name) {
    // вызывает сеттер instance.name =
    this.name = name;
  }

  get name() {
    return this._name;
  }

  set name(value) {
    this._name = value;
  }
}
```

Наследование классов, переопределение конструктора и переопределение метода:

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

  method() {
    return `${this.name} стоит`;
  }
}

class Child extends Parent {
  constructor(name, surnane) {
    super(name);
    this.surnane = surnane;
  }

  method() {
    return super.method() + ` и ${this.surnane} прячется`;
  }
};

const instance = new Child('Валера', 'Жмых');
console.log( instance.method() ); // Валера стоит и Жмых прячется
```

Ключевое слово super используется для вызова функций, принадлежащих родителю объекта. Документация здесь.

```
super([arguments]);
// вызов родительского конструктора

super.functionOnParent([arguments]);
```

Ключевое слово static определяет статический метод класса. Обращение к ним происходит через конструктор.метод, они не могут быть вызваны у экземпляров (*instance*) класса. Статические методы часто используются для создания служебных функций для приложения.

```
class MyClass {
  static method() {
    console.log('this is static method');
    return;
  }
}

MyClass.method()
// this is static method
```

Это фактически то же самое, что присвоить метод напрямую как свойство функции:
`MyClass.staticMethod = function() {...};`

Создание приватного свойства, которое не наследуется и недоступно извне функции-конструктора класса:

```
#field
// приватное
```

```
field
// публичное
```

Синтаксис

Создание классов

```
class MyClass {

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

  method() {
    console.log(this.name);
  }
  method2() {
    console.log(this.color);
  }
}

// Использование:
let instance = new MyClass('Иван', 'Red');
instance.method();
```

В JavaScript класс – это разновидность функции.
Секция **constructor** идёт первой в описании класса
В нём надо инициализировать объект. Сюда записываются такие свойства создаваемого инстанса, которые являются уникальными лично для него.

Теоретически, в конструктор можно пихнуть не только уникальные для инстанса свойства, но и общие для всех инстансов методы, но это будет неправильно с т.з. семантики. У инстансов должны быть только такие свойства, которые индивидуально характеризуют именно их.
Например, для класса User это может быть имя, фамилия, которые у каждого инстанса свои. А вот методы у них всех общие - посчитать возраст, вывести полное имя и т.д. Общие методы можно один раз положить в прототип и все, они работают одинаково для всех инстансов с разными собственными данными.

Общие методы и свойства для всего класса
Если в тело функции-класса после конструктора записать метод, то он будет передан в прототип.
Если после конструктора записать обычное свойство, то такое свойство будет записано именно в инстанс, а не в прототип.

В примере видно, что свойство отправилось в инстанс, а метод – в прототип.
Этот код работает в браузере и немного иначе работает в ноде:

```
class User {
  name = "Аноним";

  sayHi() {
    alert(`Привет, ${this.name}!`);
  }
}

const instance = new User();

instance;
// Object { name: "Аноним" }

Object.getPrototypeOf(instance);
```

```
// constructor: class User {}
// sayHi: function sayHi()
```

Геттеры, сеттеры в классах

Все методы, в т.ч. геттеры и сеттеры записываются в Class.prototype.

В примере ниже интересный случай, который работает не очевидно: откуда берётся _name в коде ниже и куда пропадает name, объявленный в конструкторе?
При создании инстанса, constructor вызывает сеттер this.name. Т.е. this.name – это не запись переменной в свойство name, а вызов функции-сеттера, куда передаётся значение, которое сеттер записывает в новое свойство _name. Соответственно, сеттер name создаёт свойство _name. А свойства name не существует, это – метод-сеттер.

```
class MyClass {

  constructor(name) {
    // вызывает сеттер instance.name =
    this.name = name;
  }

  get name() {
    return this._name;
  }

  set name(value) {
    if (value.length < 4) {
      console.log("Имя слишком короткое.");
      return;
    }
    this._name = value;
  }
}


let instance = new MyClass("Иван");

console.log(instance.name);
// Иван

instance.name = '1234';
console.log(instance.name);
// 1234

instance = new MyClass("");
// Имя слишком короткое.
```

В том, что геттер и сеттер name лежит в прототипе, можно убедиться с помощью этих команд:

```
console.log( Object.getOwnPropertyDescriptors(instance) );
{
  _name: {
    value: 'Иван',
    writable: true,
    enumerable: true,
    configurable: true
  }
}

console.log( Object.getOwnPropertyDescriptors(instance.__proto__) );
{
  constructor: {
    value: [Function: MyClass],
```

```
    writable: true,
    enumerable: false,
    configurable: true
  },

  name: {
    get: [Function: get name],
    set: [Function: set name],
    enumerable: false,
    configurable: true
  }
}
```

Это равносильно тому, если бы геттеры и сеттеры были записаны в имеющийся прототип через дескрипторы свойств:

```
Object.defineProperties(User.prototype, {
  name: {
    get() {
      return this._name
    },
    set(name) {
      // ...
    }
  }
});
```

Class Expression

Как и функции, классы можно определять внутри другого выражения, передавать, возвращать, присваивать и т.д.
Пример Class Expression (по аналогии с Function Expression):

```
let User = class {
  sayHi() {
    console.log("Привет");
  }
};

const a = new User();
console.log(a)  // User {}
a.sayHi()       // Привет
```

Class Expression может иметь имя, как и Named Function Expression.
Если у Class Expression есть имя, то оно видно только внутри класса и не видно снаружи:

```
let User = class MyClass {
  sayHi() {
    console.log(MyClass); // имя MyClass видно только внутри класса
  }
};

new User().sayHi();
// работает: [Function: MyClass]

console.log(MyClass);
// ошибка, имя MyClass не видно за пределами класса
```

Можно создавать класс динамически:
Тут речь идёт о том, что класс можно возвращать из функции.

```
function makeClass(phrase) {
  // объявляем класс и возвращаем его
  return class {
    sayHi() {
      console.log(phrase);
    };
  };
}

// Создаём новый класс
let User = makeClass("Привет");

new User().sayHi();
// Привет
```

Наследование классов

extends и super

Для того, чтобы наследовать класс от другого, мы должны использовать ключевое слово "extends" и указать название родительского класса перед {..}.

```
class Child extends Parent {...}
```

Ключевое слово extends работает, используя прототипы. Оно устанавливает Child.prototype.[[Prototype]] в Parent.prototype. Если метод не найден в Child.prototype, JavaScript берёт его из Parent.prototype.

После extends разрешены любые выражения.
Синтаксис создания класса допускает указывать после extends не только класс, но любое выражение.
В примере ниже обычная функция создаёт родительский класс, к которому привязывается дочерний.
Это может быть полезно для продвинутых приёмов проектирования, где мы можем использовать функции для генерации классов в зависимости от многих условий и затем наследовать их.
Здесь class User наследует от результата вызова f("Привет"):

```
function f(phrase) {
  return class {
    sayHi() { alert(phrase) }
  }
}

class User extends f("Привет") {}

new User().sayHi(); // Привет
```

У дочернего класса может не быть своего конструктора.
Согласно спецификации, если класс расширяет другой класс и не имеет своего конструктора, то автоматически создаётся «пустой» конструктор, который использует конструктор родительского класса.
Сбор аргументов и вызов родительского конструктора условно выглядит так:

```
class Rabbit extends Animal {
  constructor(...args) {
    super(...args);
  }
}
```

Первый пример – самый примитивный. Здесь нет дочернего конструктора и нет переопределения методов. Так как у наследника нет собственного конструктора, будет автоматом использоваться родительский конструктор. В результате дочернему классу будет доступно всё из родительского:

```
class MyClass {
  constructor(name) {
    this.name = name;
  }
  m1() {
    console.log('Метод родителя');
  }
}

class Child extends MyClass {
  m2() {
    console.log('Метод наследника');
  }
}


const instance = new Child('Valakas');
instance.name;  // Valakas
instance.m1();  // Метод родителя
instance.m2();  // Метод наследника
```

Super

Super – это ссылка на родителя, как this – ссылка на себя. Ключевое слово super используется для вызова родительского конструктора и родительских методов в дочернем классе. Стрелочные функции не имеют super. Документация здесь.

```
super([arguments]);
// вызов родительского конструктора

super.method([arguments]);
// вызов родительского метода
```

Переопределение конструктора

Если у потомка есть свой конструктор, то конструктор обязан вызывать super(...) первой строчкой:

```
Родительский
constructor(param1, param2) {
  this.param1 = param1;
  this.param2 = param2;
}

Дочерний
constructor(param1, param2, param3) {
  super(param1, param2);
  this.param3 = param3;
}
```

Это связано с тем, что когда выполняется обычный конструктор, он создаёт пустой объект и присваивает его this . Но если запускается конструктор унаследованного класса, он этого не делает. Пустой объект создаёт конструктор родительского класса, а дочерний докидывает туда что-то. Если не вызвать родительский конструктор первой строкой, то объект для this не будет создан, и мы получим ошибку.
В наследующем классе собственная функция-конструктор помечена специальным внутренним свойством `[[ConstructorKind]]:"derived"`.

Если в родительский конструктор ничего не надо класть (для него не предусотрено параметров), то всё равно надо его прописать вот так: super().

Переопределение методов
В дочернем классе можно вызывать родительские методы через super.method. Сам по себе родительский метод может использовать инстанс без всяких super, но бывают ситуации, когда технически нужно вызвать родительский метод внутри дочерней функции-конструктора. Например, если используется одноимённый метод, который делает то же самое, что и родительский и что-то дополнительно ещё. Мы делаем что-то в нашем методе и вызываем родительский метод в процессе.

```
class Parent {
  method() {
    return `стоит`;
  }

class Child extends Parent {
  method() {
    return super.method() + ` и прячется`;
  }
};
```

Пример работы super
Тут используется и заинствование конструктора, и заинствование метода:

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

  method() {
    return `${this.name} стоит`;
  }
}

class Child extends Parent {
  constructor(name, surnane) {
    super(name);
    this.surnane = surnane;
  }

  method() {
    return super.method() + ` и ${this.surnane} прячется`;
  }
};

const instance = new Child('Валера', 'Жмых');
console.log( instance.method() ); // Валера стоит и Жмых прячется
```

У стрелочных функций нет super
стрелочные функции не имеют super.
При обращении к super стрелочной функции он берётся из внешней функции.
В примере super в стрелочной функции тот же самый, что и в stop(), поэтому метод отрабатывает как и ожидается.
Если бы мы указали здесь «обычную» функцию, была бы ошибка.

```
class Rabbit extends Animal {
  stop() {
    // вызывает родительский stop после 1 секунды
    setTimeout( () => super.stop(), 1000);
  }
}
```

```
// Unexpected super
setTimeout( function() { super.stop() }, 1000);
```

Устройство super, [[HomeObject]]

Почему this не заменит super

Чтобы использовать родительские методы, использовать .this не получится: .this указывает на сам объект, и если пытаться подняться вверх по цепочке прототипов выше, чем на 1 уровень вверх, то попадёшь в бесконечную рекурсию.

Первый наследник работает, второй попадает в рекурсию:

```
let parent = {
  name: 'parent',
  method() {
    console.log(this.name);
  }
};


let child1 = {
  __proto__: parent,
  name: 'child',
  method() {
    this.__proto__.method.call(this); // (1)
  }
};


let child2 = {
  __proto__: child1,
  name: 'child2',
  method() {
    this.__proto__.method.call(this); // (2)
  }
};


child1.method();
// child

child2.method();
// RangeError: Maximum call stack size exceeded
```

Метод вызывается в контексте текущего объекта (поэтому присутствует .call). Простой вызов this.__proto__.method() будет выполнять родительский метод в контексте прототипа, а не текущего объекта.

Получается, что в обеих строках (1) и (2) значение this – это текущий объект. Таким образом, метод снова и снова вызывает себя, не поднимаясь по цепочке вызовов.
1. Внутри child2 строка (2) вызывает child1.method со значением this = child2
2. В строке (1) в child1.method мы хотим передать вызов выше по цепочке, но this = child2, поэтому this.__proto__.method снова равен child1.eat.
3. child1.method вызывает себя в бесконечном цикле, потому что не может подняться дальше по цепочке.

Интересно, что так получается только в том случае, если методы называются одинаково. Если бы они назывались method1, method2 и method3, то такого бы не произошло, потому что поиск нужного метода сразу бы лез наверх, а одноимёный запускается на второй ступени в контексте объекта this.

[[HomeObject]]

Когда функция объявлена как метод внутри класса или объекта, её свойство [[**HomeObject**]] становится равно этому объекту. Затем super использует его, чтобы получить прототип родителя и его методы.

Каждый метод знает свой [[HomeObject]] и получает метод родителя из его прототипа.
Методы запоминают свой объект во внутреннем свойстве [[HomeObject]]. Благодаря этому работает super, он в его прототипе ищет родительские методы.

```
let parent = {
  name: 'parent',
  method() {  // [[HomeObject]] == parent
    console.log(this.name);
  }
};

let child1 = {
  __proto__: parent,
  name: 'child1',
  method() {  // [[HomeObject]] == child1
    super.method(); // (1)
  }
};

let child2 = {
  __proto__: child1,
  name: 'child2',
  method() {  // [[HomeObject]] == child2
    super.method(); // (2)
  }
};

child1.method(); // child1
child2.method(); // child2
```

Методы не «свободны»

Функции в JavaScript не привязаны к объектам: их можно копировать между объектами и вызывать с любым this. Существование [[HomeObject]] нарушает этот принцип, так как методы запоминают свои объекты. [[HomeObject]] нельзя изменить, эта связь – навсегда.

Единственное место в языке, где используется [[HomeObject]] – это super. Поэтому если метод не использует super, то мы все ещё можем считать его свободным и копировать между объектами. А вот если super в коде есть, то возможны побочные эффекты.

Вот пример неверного результата super после копирования.
В примере ниже animal – это прототип для rabbit, a plant – прототип для tree. Tree хочет вызвать метод sayHi, который лежит в rabbit, но этот метод реализован через super, поэтому tree в конечном итоге переходит в animal:

```
let animal = {
  sayHi() {
    console.log("Я животное");
  }
};


// rabbit наследует от animal
let rabbit = {
  __proto__: animal,
  sayHi() {
    super.sayHi();
  }
```

```
};


let plant = {
  sayHi() {
    console.log("Я растение");
  }
};


// tree наследует от plant
let tree = {
  __proto__: plant,
  sayHi: rabbit.sayHi // (*)
};

tree.sayHi();  // Я животное
```

В строке (*), метод tree.sayHi скопирован из rabbit. Возможно, мы хотели избежать дублирования кода?
Его [[HomeObject]] – это rabbit, ведь он был создан в rabbit. Свойство [[HomeObject]] никогда не меняется.
В коде tree.sayHi() есть вызов super.sayHi(). Он идёт вверх от rabbit и берёт метод из animal.

Методы, а не свойства-функции

Свойство [[HomeObject]] определено именно для методов. Но методы должны быть объявлены именно как method(), а не «method: function()».

Для JavaScript это две большие разницы.
В примере ниже используется синтаксис свойства-функции. У него нет [[HomeObject]] и наследование не работает:

```
let parent = {
  foo: function() { // намеренно пишем так, а не foo() { ...
    console.log('foo');
  }
};

let child = {
  __proto__: parent,
  foo: function() {
    super.foo();
  }
};

child.foo();
// SyntaxError: 'super' keyword unexpected here
// потому что нет [[HomeObject]])
```

Методы запоминают свой объект во внутреннем свойстве [[HomeObject]]. Благодаря этому работает super, он в его прототипе ищет родительские методы.

Статические свойства и методы

Статические свойства – это общие свойства для всего класса. Они хранятся в конструкторе класса и доступны через него же. К ним нельзя обратиться через инстанс, только через класс.
В классе такие свойства и методы обозначаются ключевым словом static.

Статическими их можно назвать только в контексте того, что если использовать функции как конструкторы, то в создаваемых объектах это свойство будет отсутствовать, и **доступ к ним будет осуществляться только по имени класса**.

Т.к. конструкторы являются функциями, а функции – объектами, то в конструкторы можно добавлять свойства. Кроме свойств экземпляра и свойств прототипа (в конструктор записываются свойства конкретного инстанса!), JavaScript позволяет определить общие свойства класса в функции-конструкторе.

Статические свойства и методы наследуются через extends. Так происходит потому, что у функций есть свои прототипы и они наследуют друг от друга. Исключение – приватные свойства, о них ниже соответствующий раздел.

Если я правильно понял, в статические свойства записываются:
- универсальные свойства всего класса, существующие в одном экземпляре: Number.MAX_VALUE, Math. PI.
- универсальные методы для работы с экземплярами класса: String.parse(), Object.keys(obj).
- методы, легко создающие пустые шаблонные инстансы класса.

Статические методы
Документация про статические методы:
Ключевое слово static, определяет статический метод для класса. Статические методы вызываются без инстанцирования их класса и не могут быть вызваны у экземпляров (*instance*) класса. Статические методы, часто используются для создания служебных функций для приложения.

Статичный метод – это метод, который не использует собственные свойства объекта. Например, если метод печатает одну и ту же строку, а не подставляет какое-нибудь свойство этого объекта.

Статические методы наследуются. Исключение – встроенные классы. Встроенные классы не наследуют статические методы друг друга.

Статические свойства
Эта возможность была добавлена в язык недавно. Примеры работают в последнем Chrome. В ноде не работают.
Статические свойства выглядят как обычные свойства класса, но со static в начале.

```
class MyClass {

  static method() {
    console.log('this is static method');
    return;
  }

  static name = 'Valakas';
}


MyClass.method();           // this is static method
console.log(MyClass.name);  // Valakas


// Это фактически то же самое, что присвоить метод снаружи через точку
// MyClass.staticMethod = function() {...};
// MyClass.name = 'Valakas'
```

.this и статический метод
Если в статическом методе используется .this, то он **равен самой функции-конструктору**, а не экземпляру (правило «объект до точки»):

```
class MyClass {
  static method() {
    console.log(this === MyClass);
  }
}
```

```
MyClass.method();
// true
```

Наследование статических свойств и методов

Статические свойства и методы наследуются только при использовании extends.
Например, у Animal есть статический метод Animal.compare, а у Rabbit его нет, но он ему всё равно доступен, как если бы он у него был.

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

  run(speed = 0) {
    this.speed += speed; // нахуя?
    console.log(`${this.name} бежит со скоростью ${this.speed}`);
  }

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


class Rabbit extends Animal {
  hide() {
    console.log(`${this.name} прячется!`);
  }
}


let instances = [
  new Rabbit('Белый', 10),
  new Rabbit('Серый', 9)
];

instances.sort(Rabbit.compare);
instances[0].run();
// Серый бежит со скоростью 9
```

Это работает с использованием прототипов: функция `Rabbit` прототипно наследует от функции `Animal`.
Функции – это объекты, а объекты могут прототипно наследовать друг от друга

И их prototype тоже наследуют:

```
     Animal()            Animal.prototype {}
(static methods)           (methods)
       |                       |
    Rabbit()            Rabbit.prototype {}
```

Пруфы:

```
console.log(Object.getPrototypeOf(Rabbit) === Animal);
// true

console.log(Object.getOwnPropertyNames(Animal))
// [ 'length', 'prototype', 'compare', 'name' ]
```

```
// для обычных методов
console.log(Rabbit.prototype.__proto__ === Animal.prototype);
// true
```

То, что написано дальше – это задача после темы. Но в ней ещё немного теории и примеров, поэтому я её помещу сюда. Важно не забывать, что всё, что здесь написано, касается статических методов, а не просто методов объектов.

Примеры использования

Класс Article может создавать объекты-статьи, и эти объекты нужно сравнивать между собой. Для этого создаётся статический метод Article.compare:

```
class Article {
  constructor(title, date) {
    this.title = title;
    this.date = date;
  }

  static compare(articleA, articleB) {
    return articleA.date - articleB.date;
  }
}

// использование
let articles = [
  new Article("HTML", new Date(2019, 1, 1)),
  new Article("CSS", new Date(2019, 0, 1)),
  new Article("JavaScript", new Date(2019, 11, 1))
];

articles.sort(Article.compare);

console.log( articles[0].title ); // CSS
```

Ещё пример – «фабричный метод».
Допустим, надо создавать статьи различными способами:
1. через заданные параметры и вводные данные – это в конструктор
2. создание пустой статьи с сегодняшней датой – это в статику.

Статический метод в коде ниже - Article.createTodays(). Он использует возврат экземпляра класса из функции.

```
class Article {
  constructor(title, date) {
    this.title = title;
    this.date = date;
  }

  static createTodays() {
    return new this('Сегодняшний дайджест', new Date());
  //return new Article('Сегодняшний дайджест', new Date());
  }
}

const instance = Article.createTodays();
console.log(instance.title)
// Сегодняшний дайджест
```

Статические методы также используются в классах, относящихся к базам данных, для поиска/сохранения/удаления вхождений в базу данных, например:

```
// предположим, что Article - это специальный класс для управления статьями
// статический метод для удаления статьи:
Article.remove({id: 12345});
```

Приватные и защищённые методы и свойства

Внутренний и внешний интерфейсы

Один из важнейших принципов объектно-ориентированного программирования – разделение внутреннего и внешнего интерфейсов. Это обязательная практика в разработке чего-либо сложнее, чем «hello world».

В объектно-ориентированном программировании **свойства и методы разделены на 2 группы**:
Внутренний интерфейс – методы и свойства, доступные из других методов класса, но не снаружи класса. Они не доступны для пользователя, но могут использоваться в доступных для пользователя методах в качестве вспомогательных механизмов.
Внешний интерфейс – методы и свойства, доступные снаружи класса.

Внутренний интерфейс. Если провести аналогию с кофеваркой – это то, что скрыто внутри: трубка кипятильника, нагревательный элемент и т.д. Внутренний интерфейс используется для работы объекта, его детали используют друг друга.
Внешний интерфейс – это то, что будет использовать пользователь. Снаружи кофеварка закрыта защитным кожухом, детали скрыты и недоступны. Мы можем использовать их функции через внешний интерфейс.

В JavaScript есть **два типа полей** (свойств и методов) объекта:
Публичные: доступны отовсюду. Они составляют внешний интерфейс. До этого момента мы использовали только публичные свойства и методы.
Приватные: доступны только внутри класса. Они для внутреннего интерфейса.

Можно сказать, что внутренний интерфейс реализуется через приватные и защищённые поля. Для сокрытия внутреннего интерфейса мы используем защищённые или приватные свойства:
_ Защищённые поля
имеют префикс _. Это хорошо известное соглашение, не поддерживаемое на уровне языка. Программисты должны обращаться к полю, начинающемуся с _, только из его класса и классов, унаследованных от него.
 # Приватные поля
имеют префикс #. JavaScript гарантирует, что мы можем получить доступ к таким полям только внутри класса. Такие свойства не наследуются. В настоящее время приватные поля не очень хорошо поддерживаются в браузерах, но можно использовать полифил.

Во многих других языках также существуют «защищённые» поля, доступные только внутри класса или для дочерних классов (то есть, как приватные, но разрешён доступ для наследующих классов) и также полезны для внутреннего интерфейса. В некотором смысле они более распространены, чем приватные, потому что мы обычно хотим, чтобы наследующие классы получали доступ к внутренним полям. Защищённые поля не реализованы в JavaScript на уровне языка, но на практике они очень удобны, поэтому их эмулируют.

_ защищённое свойство

Защищённые свойства начинаются с префикса _.
Это не синтаксис языка, это такое соглашение между кодерами, что такие свойства и методы не должны быть доступны извне. Т.к. это всего лишь соглашение, а не специальный синтаксис, такие свойства наследуются.

Допустим, есть класс-кофеварка, в нём есть такие свойства, которые сейчас являются публичными:

```
class CoffeeMachine {
  waterAmount = 0;
  constructor(power) {
    this.power = power;
```

```
  }
}
```

В примере ниже эти же свойства становятся защищёнными через префикс _.
Изменим свойство waterAmount на защищённое. Пусть оно не устанавливается ниже нуля.
Здесь есть опять интересная штука с геттерами-сеттерами: если геттер waterAmount вернёт this.waterAmount, то он будет бесконечно возвращать сам себя, потому что waterAmaunt – это геттер. Поэтому запись делается в _waterAmount.

```
class CoffeeMachine {
  _waterAmount = 0;

  setWaterAmount(value) {
    if (value < 0) throw new Error("Отрицательное количество воды");
    this._waterAmount = value;
  }

  getWaterAmount() {
    return this._waterAmount;
  }
  constructor(power) {
    this._power = power;
  }
}

// создаём новую кофеварку
let coffeeMachine = new CoffeeMachine(100);

// устанавливаем количество воды
coffeeMachine.waterAmount = -10; // Error: Отрицательное количество воды
```

Один из вариантов организации **защиты от перезаписи** – не создавать сеттер.
Иногда нужно, чтобы свойство устанавливалось только при создании объекта и после этого никогда не изменялось.
Так как у кофеварки мощность никогда не меняется, мощность имеет смысл указать только при создании объекта. Для этого нам нужно создать только геттер, но не сеттер (ахуенная защита).
Уровень воды, напротив, всегда меняется, для него есть и сеттер, и геттер:

```
class CoffeeMachine {

  constructor(power) {
    this._power = power;
  }

  get power() {
    return this._power;
  }

  setWaterAmount(value) {
    if (value < 0) throw new Error("Отрицательное количество воды");
    this._waterAmount = value;
  }

  getWaterAmount() {
    return this._waterAmount;
}

new CoffeeMachine().setWaterAmount(100);
```

приватное свойство

Эта возможность была добавлена в язык недавно. В движках JavaScript пока не поддерживается или поддерживается частично, нужен полифил.

Приватные свойства и методы инстансов должны начинаться с #. На уровне языка # является специальным символом, который означает, что поле приватное. Мы не можем получить к нему доступ извне или из наследуемых классов. К ним можно без проблем обращаться внутри функции-конструктора класса, но снаружи функции к ним уже обратиться нельзя. Для этого нужно написать специальные геттеры и сеттеры.
Приватные поля не наследуются.

В отличие от защищённых, функциональность приватных полей обеспечивается самим языком. Но если мы унаследуем что-то, то мы не получим прямого доступа к #приватным полям. Мы будем вынуждены полагаться на функции геттер/сеттер, которые создаются без # и, соответственно, наследуются.

Во многих случаях такое ограничение слишком жёсткое. Раз уж мы расширяем класс, у нас может быть вполне законная причина для доступа к внутренним методам и свойствам. Поэтому защищённые свойства используются чаще, хоть они и не поддерживаются синтаксисом языка.

Приватные поля не конфликтуют с публичными. У нас может быть #приватное и публичное поля с одинаковым именем:

```
#field
// приватное

field
// публичное
```

Квадратные скобки тоже не дают доступ к приватным полям:

```
#value = 0;
simpleValue = 123;

getValue() {
  // return this.#value
  const key = '#value'
  return this[key];      // undefined
}
```

Геттеры-сеттеры и приватные поля
Получить доступ к приватному свойству можно только через специально написанные для него геттеры-сеттеры.
В примере ниже создаётся класс, в котором есть приватное свойство и обычное.

```
class MyClass {
  #value = 0;
  simpleValue = 123;

  getValue() {
    return this.#value
  }
  setValue(num) {
    this.#value = num;
  }
}

const instance = new MyClass();

instance;             // MyClass {simpleValue: 123, #value: 0}
instance.simpleValue; // 123
```

```
instance.#value;      // Private field '#value' must be declared in an enclosing class

// через геттеры и сеттеры всё работает
instance.getValue();  // 0
instance.setValue('new Value');
instance;             // MyClass {simpleValue: 123, #value: "new Value"}
```

Расширение встроенных классов

От встроенных классов, таких как Array, Map и других, тоже можно наследовать.

Если от них наследовать и применить .map .filter и другие функции на экземпляре наследующего потомка, они будут возвращать экземпляр класса-потомка. Так происходит потому, что для создания пропущенного через родительскую функцию дочернего экземпляра вновь используется свойство дочернего экземпляра .constructor.

Т.е. JS сначала ищет у родителя функцию, прогоняет дочерний экземпляр через неё, а для получившегося результата вызывает instance.constructor.

Здесь речь идёт не о статических методах (таких как Object.keys, например), а о методах экземпляров. Если наследуется класс, то и прототип со всеми методами тоже наследуется.

В примере ниже видно, что использование родительского .filter возвращает экземпляр дочернего класса. Через свойство .constructor видно, что оба инстанса созданы дочерним классом.
Нужно вспомнить, что дочернему классу не обязательно иметь свой конструктор.

```
// расширим array и добавим метод
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }
}

const arr = new PowerArray(1, 2, 5, 10, 50);
console.log(arr.isEmpty()); // false

let filtered = arr.filter((item) => item >= 10);
console.log(filtered);      // PowerArray(2) [ 10, 50 ]

console.log(arr.constructor)       // [Function: PowerArray]
console.log(filtered.constructor)  // [Function: PowerArray]
```

Symbol.species

Этим поведением можно управлять при помощи статического геттера Symbol.species.
Symbol.species
Документация здесь.
Symbol.species — известный символ, позволяющий определить конструктор, использующийся для создания порождённых объектов. Свойство Symbol.species, содержащее аксессор (геттер), позволяет подклассам переопределить конструктор, используемый по умолчанию для создания новых объектов.

Понятно, как это работает с родительскими функциями высшего порядка. А если бы у дочернего класса были свои функции, обрабатывающие и возвращающие новый экземпляр, они бы тоже становились родительскими?

```
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }
```

```
  // перегружаем species для использования родительского конструктора Array
  static get [Symbol.species]() {
    return Array;
  }
}
```

Наследование статических свойств встроенных классов

У встроенных объектов есть собственные статические методы, например Object.keys, Array.isArray и т. д. Обычно, когда один класс наследует от другого, то наследуются и статические методы.
Исключение – встроенные классы. Встроенные классы **не наследуют** статические методы друг друга.
Например, классы Array и Date наследуют от Object. Их экземплярам доступны методы из Object.prototype.
Но Array.[[Prototype]] не ссылается на Object, поэтому нет статических методов Array.keys() или Date.keys().

Проверка класса

В старой версии статьи есть про утиную типизацию.

Проверить, к какому классу принадлежит экземпляр, можно с помощью:

1. оператора **instanceof**
Оператор будет искать наименование класса в цепочке прототипов. Если экземпляр наследует от какого-то прототипа, то вернёт true.
Схема поиска будет другая, если используется статический метод static [Symbol.hasInstance](obj).

2. оператора **isPrototypeOf**
Это как instanceof, только наоборот.

3. метода **.toString**
Встроенный метод toString может быть позаимствован у объекта и вызван в контексте любого другого значения. И результат зависит от типа этого значения. Можно использовать {}.toString.call вместо instanceof для встроенных объектов, когда мы хотим получить тип в виде строки, а не просто сделать проверку.
Например, для массивов это будет так:

```
Object.prototype.toString.call(arr) => [object Array]
```

4. **Утиная типизация**
в ООП-языках – определение факта реализации определённого интерфейса объектом без явного указания или наследования этого интерфейса, а просто по реализации полного набора его методов.

Смысл утиной типизации – в проверке отдельных необходимых методов и свойств.
Например, мы можем проверить, что объект – массив, не вызывая Array.isArray, а просто уточнив наличие важного для нас метода, например splice:

```
var something = [1, 2, 3];

if (something.splice) {
  console.log('Это массив');
}
```

instanceof

instanceof
Документация здесь.
Оператор проверяет, принадлежит ли объект к определённому классу (конструктору).
Он проверяет, присутствует ли объект «constructor.prototype» в цепочке прототипов object.

Важно: instanceof не учитывает саму функцию-конструктор при проверке, а только prototype, который проверяется на совпадения в прототипной цепочке.

Работает как с классами, так и с конструкторами.
Оператор вернёт true, если obj принадлежит классу Class **или наследующему** от него.

Синтаксис

```
obj instanceof Class
obj instanceof Constructor

obj
Проверяемый объект

Class
Класс или функция-конструктор
```

Пример работы:

```
// с классами
class MyClass {}
const myObject = new MyClass();

myObject instanceof MyClass; // true


// с конструкторами
function MyConstructor() {}
const myObject2 = new MyConstructor();

myObject2 instanceof MyConstructor; // true
```

Алгоритм работы instanceof описан чуть ниже.

Обычно оператор instanceof просматривает для проверки цепочку прототипов. Это работает так: «экземпляр» instanceof «Класс» означает, что JS будет искать «Класс» в цепочке прототипов всё выше и выше, пока не найдёт. Например, [array] instanceof Object => true, потому что класс Array наследует от Object.

Если изменится constructor.prototype, то эта цепочка может нарушиться. Результат оператора instanceof зависит от свойства constructor.prototype (у функции-конструктора), поэтому результат оператора может поменяться после изменения этого свойства. Также результат может поменяться после изменения прототипа object (или цепочки прототипов) с помощью Object.setPrototypeOf или __proto__ (у экземпляра).

```
                              .__proto__ = Object? Да
                                   |
                                   |
[array] instanceof Object  [].__proto__ = Object? Нет
// true
```

Но это поведение может быть изменено при помощи статического метода Symbol.hasInstance. Если этот статический метод-символ присутствует у конструктора и отрабатывает, то JS не пойдёт выше по цепочке.

```
 [array] instanceof Object  Array.[Symbol.hasInstance](obj)? Да
// true
```

Symbol.hasInstance

Symbol.hasInstance

Это символ, который используется для определения является ли объект экземпляром конструктора. О нём подробнее написано в «Типы данных». Используется для изменения поведения оператора instanceof. Документация здесь.

Алгоритм instanceof

Алгоритм obj instanceof Class работает в таком порядке:
1. Если имеется статический метод Symbol.hasInstance, тогда вызвать его: Class[Symbol.hasInstance](obj). Он должен вернуть либо true, либо false, и это конец. Это как раз и есть возможность ручной настройки instanceof:

```
// проверка instanceof будет полагать, что всё со свойством canEat - животное Animal
class Animal {
  static [Symbol.hasInstance](obj) {
    if (obj.canEat) return true;
  }
}

let obj = { canEat: true };
alert(obj instanceof Animal); // true: вызван Animal[Symbol.hasInstance](obj)
```

2. Большая часть классов не имеет метода Symbol.hasInstance. В этом случае снизу вверх проверяется, равен ли Class.prototype одному из прототипов в прототипной цепочке obj:

```
obj.__proto__ === Class.prototype?
obj.__proto__.__proto__ === Class.prototype?
obj.__proto__.__proto__.__proto__ === Class.prototype?
...
// если какой-то из ответов true - возвратить true
// если дошли до конца цепочки - false
```

В примере ниже, совпадение будет на втором шаге:

```
class Animal {}
class Rabbit extends Animal {}

let rabbit = new Rabbit();
alert(rabbit instanceof Animal); // true

// rabbit.__proto__ === Rabbit.prototype
// rabbit.__proto__.__proto__ === Animal.prototype (совпадение!)
```

Последствия изменения прототипа

Это может приводить к интересным последствиям при изменении свойства prototype после создания объекта. Как, например, тут:

```
function Rabbit() {}
```

```
// создаём экземпляр
let rabbit = new Rabbit();

// заменяем прототип
Rabbit.prototype = {};

// ...больше не rabbit!
alert( rabbit instanceof Rabbit ); // false
```

В приложении к уроку задача, которая тоже помогает понять эту тему:
Почему instanceof в примере ниже возвращает true? Мы же видим, что «a» не создан с помощью B().

```
function A() {}
function B() {}

A.prototype = B.prototype = {};

let a = new A();

console.log( a instanceof B ); // true
```

instanceof не учитывает саму функцию при проверке, а только prototype, который проверяется на совпадения в прототипной цепочке.
И в данном примере a.__proto__ == B.prototype, так что instanceof возвращает true.
Таким образом, по логике instanceof, именно prototype в действительности определяет тип, а не функция-конструктор.

isPrototypeOf

Есть метод objA.isPrototypeOf(objB), который возвращает true, если объект objA есть где-то в прототипной цепочке объекта objB.
Так что obj instanceof Class можно перефразировать как Class.prototype.isPrototypeOf(obj).
Забавно, но сам конструктор Class не участвует в процессе проверки! Важна только цепочка прототипов Class.prototype.

.toString возвращает тип

Обычные объекты преобразуется к строке как [object Object].
Так работает реализация метода toString. Но у toString имеются скрытые возможности, которые делают метод гораздо более мощным. Мы можем использовать его как расширенную версию typeof и как альтернативу instanceof.

Согласно спецификации, встроенный метод toString может быть позаимствован у объекта и вызван в контексте любого другого значения. И результат зависит от типа этого значения.

- Для числа это будет [object Number]
- Для булева типа это будет [object Boolean]
- Для null: [object Null]
- Для undefined: [object Undefined]
- Для массивов: [object Array]
- …и т.д. (поведение настраивается).

продемонстрируем:

```
// скопируем метод toString в переменную для удобства
let objectToString = Object.prototype.toString;

// какой это тип?
let arr = [];

console.log( objectToString.call(arr) ); // [object Array]
```

В примере мы использовали call, как описано в главе Декораторы и переадресация вызова, call/apply, чтобы выполнить функцию objectToString в контексте this=arr.
Внутри, алгоритм метода toString анализирует контекст вызова this и возвращает соответствующий результат.

Больше примеров:

```
let s = Object.prototype.toString;

console.log( s.call(123) );          // [object Number]
console.log( s.call(null) );         // [object Null]
console.log( s.call(console.log) ); // [object Function]
```

Symbol.toStringTag

Поведение метода объектов toString можно настраивать, используя специальное свойство объекта Symbol.toStringTag. О нём подробнее написано в «Типы данных».
Например:

```
let user = {
  [Symbol.toStringTag]: "User"
};

console.log( {}.toString.call(user) ); // [object User]
```

Такое свойство есть у большей части объектов, специфичных для определённых окружений. Вот несколько примеров для браузера:

```
// toStringTag для браузерного объекта и класса
console.log( window[Symbol.toStringTag]);                    // window
console.log( XMLHttpRequest.prototype[Symbol.toStringTag] ); // XMLHttpRequest

console.log( {}.toString.call(window) );                     // [object Window]
console.log( {}.toString.call(new XMLHttpRequest()) );       // [object XMLHttpRequest]
```

Результат преобразования к строке – это значение свойства obj.Symbol.toStringTag (если он имеется), которое находится в [object ...].
В итоге мы получили «typeof на стероидах», который не только работает с примитивными типами данных, но также и со встроенными объектами, и даже может быть настроен.

Можно использовать {}.toString.call вместо instanceof для встроенных объектов, когда мы хотим получить тип в виде строки, а не просто сделать проверку.

Как мы можем видеть, технически {}.toString «более продвинут», чем typeof.
А оператор instanceof – отличный выбор, когда мы работаем с иерархией классов и хотим делать проверки с учётом наследования.

Утиная типизация

Это раздел из старой страницы.
Альтернативный подход к типу – «утиная типизация», которая основана на одной известной пословице: *«If it looks like a duck, swims like a duck and quacks like a duck, then it probably is a duck (who cares what it really is)»*.
В переводе: *«Если это выглядит как утка, плавает как утка и крякает как утка, то, вероятно, это утка (какая разница, что это на самом деле)»*.

Утиная типизация (англ. Duck typing) в ООП-языках – определение факта реализации определённого интерфейса объектом без явного указания или наследования этого интерфейса, а просто по реализации полного набора его методов.

Смысл утиной типизации – в проверке отдельных необходимых методов и свойств.
Например, мы можем проверить, что объект – массив, не вызывая Array.isArray, а просто уточнив наличие важного для нас метода, например splice

```
var something = [1, 2, 3];

if (something.splice) {
  console.log( 'Это утка! То есть, массив!' );
}
```

в if мы не вызываем метод something.splice(), а пробуем получить само свойство something.splice. Для массивов оно всегда есть и является функцией, т.е. даст в логическом контексте true.

Аналогично, проверить на дату можно, определив наличие метода getTime:

```
var x = new Date();

if (x.getTime) {
  alert( 'Дата!' );
  alert( x.getTime() ); // работаем с датой
}
```

С виду такая проверка хрупка, её можно «сломать», передав похожий объект с тем же методом.
Но как раз в этом и есть смысл утиной типизации: если объект похож на дату, у него есть методы даты, то будем работать с ним как с датой (какая разница, что это на самом деле).
То есть мы намеренно позволяем передать в код нечто менее конкретное, чем определённый тип, чтобы сделать его более универсальным.

Проверка интерфейса

Если говорить словами «классического программирования», то «duck typing» – это проверка реализации объектом требуемого интерфейса. Если реализует – ок, используем его. Если нет – значит это что-то другое.

Примеси

В JavaScript можно наследовать только от одного объекта. Объект имеет единственный [[Prototype]]. И класс может расширить только один другой класс.
Иногда это может ограничивать нас. Например, у нас есть класс StreetSweeper и класс Bicycle, а мы хотим создать их смесь. Для таких случаев существуют «примеси».

Примесь – это класс, методы которого предназначены для использования в других классах, но без наследования от примеси. Примесь только содержит в себе методы, которые реализуют определённое поведение. Метод, хранящиеся в примеси, используются другими объектами, чтобы добавить функциональность другим классам.

Простейший способ реализовать примесь в JavaScript – это создать объект с полезными методами, которые затем могут быть добавлены в прототип любого класса. Это не наследование, а просто копирование методов.

В примере ниже создаётся самостоятельный объект, в котором лежат свойства, использующие this. Эти свойства копируются в прототип класса через Object.assign(куда?, откуда?). Так как методы из примеси залиты в прототип класса, экземпляры класса могут их использовать.
Таким образом, класс User может наследовать от другого класса, но при этом также включать в себя примеси, «подмешивающие» другие методы:

```
// объект-примесь
let sayHiMixin = {

  sayHi() {
    console.log(`Привет, ${this.name}`);
  },

  sayBye() {
    console.log(`Пока, ${this.name}`);
  }
};


// создание класса
class User {
  constructor(name) {
    this.name = name;
  }
}

// копирование в прототип класса
Object.assign(User.prototype, sayHiMixin);

new User('Вася').sayHi();
new User('Вася').sayBye();
```

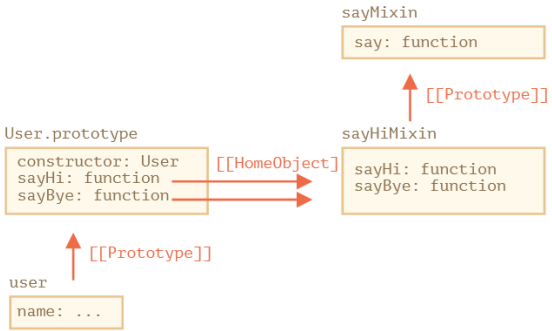
Тут используется метод Object.assign()
Создаёт новый объект путём копирования значений всех собственных перечислимых свойств из одного или более исходных объектов в целевой объект.

```
Object.assign(target, ...sources)

var object = { key1: 'value1', key2: 'value2' };
var copy = Object.assign({}, object);
```

Сами примеси могут наследовать друг друга. В примере ниже sayHiMixin наследует от sayMixin:

```
let sayMixin = {
  say(phrase) {
    console.log(phrase);
  }
};


let sayHiMixin = {
  __proto__: sayMixin,
  // для задания прототипа можно также использовать Object.create

  sayHi() {
    // вызвать метод родителя
    super.say(`Привет, ${this.name}`);
  },

  sayBye() {
    super.say(`Пока, ${this.name}`);
  }
};

class User {
  constructor(name) {
    this.name = name;
  }
}

Object.assign(User.prototype, sayHiMixin);

new User('Вася').sayHi()
// Привет, Вася
```

При вызове родительского метода super.say() из sayHiMixinэтот метод ищется в прототипе самой примеси, а не класса:



Это связано с тем, что методы sayHi и sayBye были изначально созданы в объекте sayHiMixin. Несмотря на то, что они скопированы в прототип класса User, их внутреннее свойство [[HomeObject]] ссылается на sayHiMixin, как показано на картинке выше.
Так как super ищет родительские методы в [[HomeObject]].[[Prototype]], это означает sayHiMixin.[[Prototype]], а не User.[[Prototype]].

для задания прототипа можно также использовать:
Object.create()
Создаёт пустой объект со свойством [[Prototype]], указанным как proto, и необязательными дескрипторами свойств descriptors.

```
Object.create(proto[, propertiesObject])
```

```
proto
Объект, который станет прототипом вновь созданного объекта
propertiesObject
Необязательный. дескрипторы свойств.
```

??? EventMixin

Вообще ничего не понял.

Многие браузерные объекты могут генерировать события. События – отличный способ передачи информации всем, кто в ней заинтересован.
Давайте создадим примесь, которая позволит легко добавлять функциональность по работе с событиями любым классам/объектам.

Методы в примеси:
.trigger(name, [data])
Для генерации события. Аргумент name – это имя события, за которым могут следовать другие аргументы с данными для события.

.on(name, handler)
Назначает обработчик для события с заданным именем. Обработчик будет вызван, когда произойдёт событие с указанным именем name, и получит данные из .trigger.

.off(name, handler)
Удаляет обработчик указанного события.

После того, как все методы примеси будут добавлены, объект `user` сможет сгенерировать событие "login" после входа пользователя в личный кабинет. Другой объект – calendar, сможет использовать это событие, чтобы показывать зашедшему пользователю актуальный для него календарь.
Или menu может генерировать событие "select", когда элемент меню выбран, а другие объекты могут назначать обработчики, чтобы реагировать на это событие, и т.п.

Код примеси:

```
```

```
let eventMixin = {
  /**
   * Подписаться на событие, использование:
   * menu.on('select', function(item) { ... }
   */
  on(eventName, handler) {
    if (!this._eventHandlers) this._eventHandlers = {};
    if (!this._eventHandlers[eventName]) {
      this._eventHandlers[eventName] = [];
    }
    this._eventHandlers[eventName].push(handler);
  },

  /**
   * Отменить подписку, использование:
   * menu.off('select', handler)
   */
  off(eventName, handler) {
    let handlers = this._eventHandlers && this._eventHandlers[eventName];
    if (!handlers) return;
    for (let i = 0; i < handlers.length; i++) {
      if (handlers[i] === handler) {
        handlers.splice(i--, 1);
      }
    }
```

```
  },

  /**
   * Сгенерировать событие с указанным именем и данными
   * this.trigger('select', data1, data2);
   */
  trigger(eventName, ...args) {
    if (!this._eventHandlers || !this._eventHandlers[eventName]) {
      return; // обработчиков для этого события нет
    }

    // вызовем обработчики
    this._eventHandlers[eventName].forEach(handler => handler.apply(this, args));
  }
};
```

Итак, у нас есть 3 метода:
.**on**(eventName, handler)
назначает функцию handler, чтобы обработать событие с заданным именем. Обработчики хранятся в свойстве _eventHandlers, представляющим собой объект, в котором имя события является ключом, а массив обработчиков – значением.

.**off**(eventName, handler)
убирает функцию из списка обработчиков.

.**trigger**(eventName, ...args)
генерирует событие: все назначенные обработчики из _eventHandlers[eventName] вызываются, и ...args передаются им в качестве аргументов.

Использование:

```
// Создадим класс
class Menu {
  choose(value) {
    this.trigger("select", value);
  }
}
// Добавим примесь с методами для событий
Object.assign(Menu.prototype, eventMixin);

let menu = new Menu();

// Добавить обработчик, который будет вызван при событии "select":
menu.on("select", value => alert(`Выбранное значение: ${value}`));

// Генерирует событие => обработчик выше запускается и выводит:
menu.choose("123"); // Выбранное значение: 123
```

Теперь если у нас есть код, заинтересованный в событии "select", то он может слушать его с помощью menu.on(...).
А eventMixin позволяет легко добавить такое поведение в любой класс без вмешательства в цепочку наследования.

Из комментов:
Разобрал пример с событиями в песочнице с подробными комментариями... Разбирался по ходу кода, поэтому сначала комментарии краткие, по действиям, а потом идут обобщения. Ну и пару примеров от себя, для

закрепления.
Когда читал уже авторские примечания, уже после того, как полностью разобрался, и то, сразу понять что же делает эта хреновина и для чего создаётся сразу не совсем очевидно. То есть, на мой взгляд, тут надо либо СНАЧАЛА разбираться полностью в коде и уже потом говорить о возможных вариантах применения этому (а автор сразу говорит о применении). Либо УЖЕ иметь в голове задачи, о которых говорит автор, и следовать ходу его рассуждений чётко понимая какие проблемы подобный код призван решать. Без этого, по примечаниям, воссоздать что же именно делает код - по-моему, ну очень не интуитивно... Кстати, общая логика, тоже не такая уж сложная, когда понимаешь, что именно призваны реализовать те или иные функции...

Итого

Примесь – общий термин в объектно-ориентированном программировании: класс, который содержит в себе методы для других классов.

Некоторые другие языки допускают множественное наследование. JavaScript не поддерживает множественное наследование, но с помощью примесей мы можем реализовать нечто похожее, скопировав методы в прототип.

Мы можем использовать примеси для расширения функциональности классов, например, для обработки событий, как мы сделали это выше.

С примесями могут возникнуть конфликты, если они перезаписывают существующие методы класса. Стоит помнить об этом и быть внимательнее при выборе имён для методов примеси, чтобы их избежать.

JS – Обработка ошибок

Ошибки кратко

Обычно в случае ошибки скрипт «падает», т.е. программа останавливается, вылетает с выводом ошибки в консоль. Чтобы этого не происходило, можно эти ошибки перехватывать, обрабатывать и продолжать работу.

Есть одно требования: ошибки должны возникать в синтаксически правильном коде. Если забыли поставить закрывающую скобку – это синтаксически неправильный код, такая ошибка не обработается, потому что JS просто не знает, как такой код выполнить.
Если делается ссылка на несуществующую переменную – такую ошибку можно обработать.

Исключения – это ошибки, которые возникаю в синтаксически корректном коде. Их также называют «ошибками во время выполнения». Есть ещё близкое понятие throw (возбуждение исключения), о нём написано ниже.

Блоки инструкций

Мануал здесь.
Конструкция try...catch состоит из 3-х блоков инструкций:
1. **try** {…}
В нём исполняется основной код. Здесь «должны» происходить ошибки.

2. **catch**(err) {…}
Этот блок будет вызван только в том случае, если в try произошла ошибка. Переменная «err» - это объект ошибки, который автоматом передаётся сюда как аргумент. Можно использовать любое имя для объекта. Подробнее об объекте ошибки будет ниже.
Если в блоке try нет ошибок, то блок catch игнорируется. Если ошибка есть, то выполнение try прерывается, и поток управления переходит в начало catch(err). Таким образом, при ошибке в блоке try {…} мы получаем возможность обработать ошибку внутри catch.

3. **finally** {…}
Выполняется всегда, вне зависимости от того, была ошибка или нет.
Если ошибки не было, то он выполнится после try, если была – выполнится после блока catch.
Секцию finally часто используют, когда мы начали что-то делать и хотим завершить это вне зависимости от наличия или отсутствия ошибки. Например, для завершения измерений.

См. также «finally, особенности»

На практике это выглядит так:

```
try {
  // код
} catch (err) {
  // обработка ошибки
} finally {
  // выполняется всегда
}
```

Всё, что объявлено в try, находится в локальной зоне видимости. Если тут объявить переменные, их не будет видно снаружи.

Блоки catch или finally опциональны. Можно использовать только конструкции try..catch и try..finally.

Конструкция try..catch работает синхронно. Это означает, что исключение, которое произойдёт в асинхронном коде, запланированном «на будущее», конструкция try..catch не поймает.
Например, в setTimeout асинхронная функция выполнится в стеке позже, когда движок уже покинул конструкцию try..catch. Чтобы поймать исключение внутри запланированной в таймере функции, try..catch должен находиться внутри самой этой функции.
Т.е. неправильно помещать таймер в конструкцию try-catch, надо наоборот конструкцию засовывать в таймер:

```
setTimeout(() => {
  try {
    noSuchVariable;
  }
  catch {
    console.log('Поймана!');
}
}, 100);

// Поймана!


Вот так делать нельзя
try {
  setTimeout(function() {
    noSuchVariable; // скрипт упадёт тут
  }, 1000);

} catch (e) {
  console.log( "catch" ); // не сработает
}

// ReferenceError: noSuchVariable is not defined
```

throw, исключения

Мануал здесь, хорошая статья на professorweb здесь.
Можно создавать собственные исключения. Например, если с т.з. JS код корректный и всё хорошо, но тебя не устраивает результат выполнения, можно создать собственное исключение и перехватить его.

Исключение - это сигнал, указывающий на возникновение какой-либо исключительной ситуации или ошибки.

throw (возбуждение исключения)
это способ просигнализировать о такой ошибке или исключительной ситуации.

catch (перехватить исключение)

значит обработать его, т.е. предпринять действия, необходимые или подходящие для восстановления после исключения.

В JavaScript обработка ошибок работает через механизм исключений. Одни функции их возбуждают, другие обрабатывают через try..catch.
Инструкция throw позволяет генерировать исключения, определяемые пользователем. Выполнение кода после throw не будет продолжено и управление будет передано в ближайший блок catch в стеке вызовов.
Если catch блоков среди вызванных функций нет, выполнение программы будет остановлено.
Сгенерированное исключение можно обработать в блоке catch, а можно не обрабатывать и передать дальше, т.е. «пробросить».

Работает предельно просто: указываешь оператор «throw» и что надо выбросить.
Сгенерировать и выбросить (не уверен, что это правильный термин) можно что угодно: число, строку и т.д., но чаще всего выбрасывается объект ошибки.
Синтаксис:

```
throw <объект ошибки>

throw "Error2"; // генерирует исключение, значением которого является строка
throw 42;       // ... число 42
throw true;     // ... логическое значение true
```

В действии:

```
const a = 4;

try {
  if (typeof a !== 'string') {
    throw 'неправильный тип данных';
  }

  } catch(err) {
    console.log(err);
}

// неправильный тип данных
```

Далее блок catch выполняет всю работу по обработке.
Как было сказано ранее, оператору throw чаще всего передаётся объект ошибки. Объекты могут быть встроенными или пользовательскими:

```
throw new Error('неправильный тип данных');
```

Подробнее об объектах ошибки в разделе «Объект ошибки»

Проброс исключения

Как было сказано ранее, исключение можно обработать в блоке catch, а можно «пробросить» - не обрабатывать, а вернуть наружу. Проброс исключения – это создание ещё одного исключения в блоке catch, которое никто уже ловить не будет.

В блок catch попадают все ошибки, которые появляются в блоке try. Правильно, когда блок catch работает только с теми ошибками, которые ему «известны». Мы пишем блок кода, который обрабатывает определённый тип ошибок. Получается не универсальный обработчик, который обрабатывает вообще все любые ошибки, а специальный, который по-разному работает с каждым известным типом ошибок.

Техника «проброс исключения» выглядит так:
1. Блок catch получает все ошибки.

2. В блоке catch(err) {...} мы анализируем объект ошибки err.
3. Если мы не знаем как её обработать, тогда делаем throw err уже в блоке catch.

Эта тема очень тесно связана с объектом ошибок, но я тут покажу простой пример. Catch в примере ниже умеет обрабатывать только ошибки типа ReferenceError, т.е. «переменная не определена». В случае, если в catch попадает другой тип ошибки, он этот же объект ошибки пробрасывает дальше:

```
try {
  constIsNotDefined;

} catch(err) {

  if (err instanceof ReferenceError) {
    console.log('Переменная не определена')
    // какой-то код для обработки ошибки

  } else {
    throw err;
    // пробросить эту ошибку наружу
  }
}
```

Подробнее о catch

Объект ошибки имеет 3 основных свойства:
name – имя ошибки
message – текстовое сообщение о деталях ошибки.
stack – текущий стек вызова: строка, содержащая информацию о последовательности вложенных вызовов, которые привели к ошибке. Используется в целях отладки. Это нестандартное свойство, но поддерживается большинством окружений.

При возникновении ошибки, создаётся объект ошибки и передаётся как аргумент в блок catch()
В примере ниже распечатаны основные свойства одной из ошибок:

```
try {
  noSuchVariable
} catch(err) {
  console.log(err.name);     // ReferenceError
  console.log(err.message);  // noSuchVariable is not defined
  console.log(err.stack)     // большое описание, где и что произошло
}
```

Можно распечатать сам объект ошибки. У меня получается то же самое, как если распечатать стек.

С помощью обработки ошибок можно выполнять широкий круг действий.
Например, если пришёл некорректный JSON, можно вообще не связываться с ошибкой, а выполнить несколько сопуствующих действий:
- написать пользователю сообщение через alert, почему произошла ошибка, потому что консоль он не увидит;
- отправить новый сетевой запрос;
- предложить альтернативный способ ввода;
- отослать информацию об ошибке на сервер для логирования;
… всё лучше, чем просто «падение».

В блок catch можно не передавать объект ошибки, если тебе не надо его обрабатывать. Это новая возможность и не всеми браузерами поддерживается. Выглядит вот так, без скобок: catch { }

Глобальный catch
Если происходит ошибка вне блоков try-catch, её можно поймать и обработать. В JS нет такой функции, но обычно окружения предоставляют соответствующие инструменты.

В Node.js для этого есть process.on("uncaughtException").
В браузере мы можем присвоить функцию специальному свойству window.onerror, которая будет вызвана в случае необработанной ошибки.

```
window.onerror = function(message, url, line, col, error) {
  // ...
};
```

message
Сообщение об ошибке.
url
URL скрипта, в котором произошла ошибка.
line, col
Номера строки и столбца, в которых произошла ошибка.
error
Объект ошибки.

Роль глобального обработчика window.onerror обычно заключается не в восстановлении выполнения скрипта – это скорее всего невозможно в случае программной ошибки, а в отправке сообщения об ошибке разработчикам.

Существуют также веб-сервисы, которые предоставляют логирование ошибок для таких случаев, такие как https://errorception.com или http://www.muscula.com.
Они работают так:
1. Мы регистрируемся в сервисе и получаем небольшой JS-скрипт (или URL скрипта) от них для вставки на страницы.
2. Этот JS-скрипт ставит свою функцию window.onerror.
3. Когда возникает ошибка, она выполняется и отправляет сетевой запрос с информацией о ней в сервис.
4. Мы можем войти в веб-интерфейс сервиса и увидеть ошибки.

Подробнее о finaly

У finaly есть несколько особенностей.

finally и return
В примере ниже из try происходит return, но finally получает управление до того, как контроль возвращается во внешний код.

```
function func() {

  try {
    return 1;

  } catch (e) {
    /* ... */
  } finally {
    console.log( 'finally' );
  }
}

console.log( func() );
// finally
// 1
```

try finally

Конструкция try..finally без секции catch также полезна. Мы применяем её, когда не хотим здесь обрабатывать ошибки (пусть выпадут), но хотим быть уверены, что начатые процессы завершились.
В приведённом коде ошибка всегда выпадает наружу, потому что тут нет блока catch. Но finally отрабатывает до того, как поток управления выйдет из функции:

```
function func() {
  // начать делать что-то, что требует завершения (например, измерения)
  try {
    // ...
  } finally {
    // завершить это, даже если все упадёт
  }
}
```

Объект ошибки

Встроенный Error

Документация тут.
Во время выполения кода ошибки приводят к созданию и выбрасыванию новых объектов Error.
Конструктор **Error** создаёт объекты ошибок, а также используется в качестве базового для создания пользовательских исключений с использованием наследования.

Синтаксис создания стандартного объекта ошибки:

```
Error(message)
// можно использовать без new
new Error(message)

message
Описание ошибки, обычная строка
```

Вообще, синтаксис создания ошибки шире. После message, можно также указать fileName, lineNumber, но на практике используют только message, а остальные значения подставляются автоматом. См. подробнее в документации.

Свойства объекта ошибки

.message
Описание и детали ошибки. Это то, что было указано в message при создании.
.name
Название ошибки. Как правило, это название функции-конструктора.
.stack
Текущий стек вызова: строка, содержащая информацию о последовательности вложенных вызовов, которые привели к ошибке. Используется в целях отладки. Это нестандартное свойство, но поддерживается большинством окружений.

.constructor – определяет функцию, создающую прототип экземпляра ошибки.
Error.prototype – позволяет добавлять свойства в экземпляры объекта Error.

Условно (!) можно сказать, что класс Error внутри устроен так. Это нужно понимать, чтобы иметь представление, как создавать пользовательские ошибки:

```
// "Псевдокод" встроенного класса Error
class Error {
  constructor(message) {
    this.message = message;
    this.name = "Error"; // (разные имена для разных встроенных классов ошибок)
    this.stack = <стек вызовов>; // нестандартное свойство, но обычно поддерживается
```

```
  }
}
```

При возникновении ошибки, создаётся объект ошибки и передаётся как аргумент в блок catch()
В примере ниже распечатаны основные свойства одной из ошибок:

```
try {
  noSuchVariable
} catch(err) {
  console.log(err.name);     // ReferenceError
  console.log(err.message);  // noSuchVariable is not defined
  console.log(err.stack)     // большое описание, где и что произошло
}
```

Можно распечатать сам объект ошибки. У меня получается то же самое, как если распечатать стек.

Для встроенных ошибок свойство name – это в точности имя конструктора. А свойство message берётся из аргумента. Например:

```
let error = new Error("Ого, ошибка! o_O");

alert(error.name); // Error
alert(error.message); //  Ого, ошибка! o_O
```

Всё это упаковывается в throw:

```
if (!user.name) {
    throw new SyntaxError("Данные неполны: нет имени"); // (*)
  }
```

Встроенные конструкторы ошибок

Кроме общего конструктора Error, в JavaScript существует ещё семь других основных конструкторов ошибок. По обработке исключений смотрите раздел Выражения обработки исключений.

В JavaScript есть множество встроенных конструкторов для стандартных ошибок: Error, SyntaxError, ReferenceError, TypeError и другие. Можно использовать и их для создания объектов ошибки.
Их синтаксис:

```
let error = new Error(message);
// или
let error = new SyntaxError(message);
let error = new ReferenceError(message);
// ...
```

ReferenceError
Создаёт экземпляр, представляющий ошибку, возникающую при разыменовывании недопустимой ссылки.

SyntaxError
Создаёт экземпляр, представляющий синтаксическую ошибку, возникающую при разборе исходного кода в функции eval().

TypeError
Создаёт экземпляр, представляющий ошибку, возникающую при недопустимом типе для переменной или параметра.

EvalError
Создаёт экземпляр, представляющий ошибку, возникающую в глобальной функции eval().

InternalError
Создаёт экземпляр, представляющий ошибку, возникающую при выбрасывании внутренней ошибки в движке JavaScript. К примеру, ошибки «слишком глубокая рекурсия» («too much recursion»).

RangeError
Создаёт экземпляр, представляющий ошибку, возникающую при выходе числовой переменной или параметра за пределы допустимого диапазона.

URIError
Создаёт экземпляр, представляющий ошибку, возникающую при передаче в функции encodeURI() или decodeURI() недопустимых параметров.

Пользовательские ошибки

Можно создавать собственные объекты ошибок на разные случаи, не предусмотренные стандартными конструкторами. Самый очевидный и правильный способ – это расширить стандартный класс Error.

```
class MyError extends Error {
  constructor(message) {
    super(message);
    this.name = 'my Error';
  }
}


try {
  throw new MyError('Сообщение об ошибке');

} catch(err) {
  console.log(err.name);     // my Error
  console.log(err.message);  // Сообщение об ошибке
}
```

По правилам наследования, в конструкторе наследника обязательно должна быть родительская функция super. Родительский конструктор устанавливает свойство message.
This.name – тоже в конструкторе. Это нужно для того, чтобы переопределить свойство .name родительского класса. Если этого не сделать, то его значением будет просто стандартное ‘Error’, как и у родителя.

Как это используется на практике.
Допустим, есть функция, которая принимает строку JSON и конвертит его её объект. Нужно создать дочерний класс объекта ошибки, который будет проверять, все ли свойства создаваемого объекта получены через строку JSON.

```
// создание пользовательского класса ошибки
// через расширение встроенного класса Error
class MyValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = 'my Validation Error';
  }
}

// функция конвертит строку JSON в объект
// и проверяет наличие всех необходимых свойств
function readUser(json) {
  const user = JSON.parse(json);
```

```
  if (!user.age) {
    throw new MyValidationError('Нет поля .age');
  }
  if (!user.name) {
    throw new MyValidationError('Нет поля .name');
  }
  return user;
}

// запуск блока try...catch
try {
  let user = readUser('{ "age": 25 }');

} catch(err) {

  if (err instanceof MyValidationError) {
    console.log(`Некорректные данные: ${err.message}`);

  } else if (err instanceof SyntaxError) {
    console.log(`JSON ошибка синтаксиса: ${err.message}`);

  } else {
    throw err; // в других случаях - пробросить исключение
  }
}

// Некорректные данные: Нет поля .name
```

instanceof используется для проверки конкретного типа ошибки.
В некоторых примерах используется проверка на факт наличия какого-то конкретного свойства, типа !obj.name, но версия с instanceof лучше, потому что в будущем можно расширить ValidationError, сделав его подтипы, такие как PropertyRequiredError, и проверка instanceof продолжит работать для новых наследованных классов.

Также важно, что если catch встречает неизвестную ошибку, то он пробрасывает её. Блок catch знает, только как обрабатывать ошибки валидации и синтаксические ошибки, а другие виды ошибок (из-за опечаток в коде и другие непонятные) он должен выпустить наружу.

Продвинутое наследование

Сейчас класс MyValidationError очень общий, его можно детализировать дальше и выделить из него класс ошибок, который будет отслеживать наличие всех необходимых свойств, переданных в JSON: PropertyRequiredError.
Он будет нести дополнительную информацию о свойстве, которое отсутствует.

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = 'ValidationError';
  }
}

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super(`Нет свойства: ${property}`);  // это св-во .message!
    this.name = 'PropertyRequiredError';
    this.property = property;
  }
}

const readUser = (json) => {
  const user = JSON.parse(json);
```

```
  if (!user.age) {
    throw new PropertyRequiredError('age');

  } else if (!user.name) {
    throw new PropertyRequiredError('name');
  }

  return user;
};

try {
  const user = readUser('{ "age": 25 }');

} catch(err) {
  if (err instanceof ValidationError) {  // указан родительский класс
    console.log(`Неверные данные: ${err.message}`);
    console.log(err.name);
    console.log(err.property);

  } else if (err instanceof SyntaxError) {
    console.log(`Ошибка синтаксиса: ${err.message}`);

  } else {
    throw err;
  }
}

// Неверные данные: Нет свойства: name
// PropertyRequiredError
// name
```

В этом примере есть непонятное место.
В блоке catch реализуется проверка на материнский класс, хотя экземпляр ошибки принадлежит к дочернему. По умолчанию, все дочерние классы пройдут проверку instanceof на материнский класс. Идея состоит в том, чтобы ловить сразу всю группу ошибок, это делается через материнский класс. При этом сами ошибки – инстансы дочернего класса, и свойства .name .message записаны в них как предписывает дочерний класс. Получается, что все ошибки можно отловить с помощью материнского класса, а потом вывести их индивидуальные свойства.

свойство this.name в конструкторе PropertyRequiredError снова присвоено вручную. Правда, немного утомительно – присваивать this.name = <class name> в каждом классе пользовательской ошибки. Можно этого избежать, если сделать наш собственный «базовый» класс ошибки, который будет ставить this.name = this.constructor.name. И затем наследовать все ошибки уже от него.

Назовём его MyError.
Вот упрощённый код с MyError и другими пользовательскими классами ошибок.
Теперь пользовательские ошибки стали намного короче, особенно ValidationError, так как мы избавились от строки "this.name = ..." в конструкторе.

```
class MyError extends Error {
  constructor(message) {
    super(message);
    this.name = this.constructor.name;
  }
}

class ValidationError extends MyError {}

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super(`Нет свойства: ${property}`);
```

```
    this.property = property;
  }
}

console.log( new PropertyRequiredError('field').name );
// PropertyRequiredError
```

Обёртывание исключений

Какая-то хрень.
В последнем примере для каждого вида ошибки в конструкции readUser создавался свой блок if чтобы выбросить исключение. Так делать оч долго, если пользовательских ошибок много.

Исключения можно оборачивать.
ReadError будет распознавать ошибки пользовательских классов.

```
class ReadError extends Error {
  constructor(message, cause) {
    super(message);
    this.cause = cause;
    this.name = 'ReadError';
  }
}

class ValidationError extends Error { /*...*/ }
class PropertyRequiredError extends ValidationError { /* ... */ }

function validateUser(user) {
  if (!user.age) {
    throw new PropertyRequiredError("age");
  }

  if (!user.name) {
    throw new PropertyRequiredError("name");
  }
}

function readUser(json) {
  let user;

  try {
    user = JSON.parse(json);
  } catch (err) {
    if (err instanceof SyntaxError) {
      throw new ReadError("Синтаксическая ошибка", err);
    } else {
      throw err;
    }
  }

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

}

try {
  readUser('{bad json}');
```

```
} catch (e) {
  if (e instanceof ReadError) {
    // Исходная ошибка: SyntaxError:Unexpected token b in JSON at position 1
  } else {
    throw e;
  }
}
```

В коде readUser распознаёт синтаксические ошибки и ошибки валидации и выдаёт вместо них ошибки ReadError. Неизвестные ошибки, как обычно, пробрасываются.

Внешний код проверяет только instanceof ReadError. Не нужно перечислять все возможные типы ошибок
Этот подход называется «обёртывание исключений», потому что мы берём «исключения низкого уровня» и «оборачиваем» их в ReadError, который является более абстрактным и более удобным для использования в вызывающем коде. Такой подход широко используется в объектно-ориентированном программировании.

Обёртывание исключений является распространённой техникой: функция ловит низкоуровневые исключения и создаёт одно «высокоуровневое» исключение вместо разных низкоуровневых. Иногда низкоуровневые исключения становятся свойствами этого объекта, как err.cause в примерах выше, но это не обязательно.
///

JS - Промисы, async/await

Статья на английском про асинхронность в книге «Вы не знаете JS» здесь.
Цикл курсов про асинхронность на хабре здесь.
Видео про Event Loop здесь.
Основные понятия асинхронного программирования, мануал – здесь.
Статья на Проглиб: здесь.

Есть специальные библиотеки для работы в асинхронном стиле:
FS (file system) – встроенный модуль для ноды: оригинал, перевод.
async – содержит десятки функций для большого числа задач, связанных с упорядочиванием асинхронных операций.
Документация по Path: оригинал, перевод.

Введение: колбэки

Пример синхронного кода для начала темы:

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}

loadScript('test.js');
console.log('текст после вызова функции');
```

Функция loadScript создаёт тег скрипт и загружает его на страницу.
На практике, дальнейший код в текущем файле не будет ждать, пока элемент будет добавлен на страницу и продолжит выполняться дальше.
Так, если запустить пример выше, то сначала выполнится печать «текст после вызова функции», а потом выполнится скрипт test.js. Если бы в этом же документе надо было использовать данные загружаемого скрипта, то произошла бы ошибка: скрипт к этому времени ещё не загружен.

Чтобы решить эту проблему, можно сделать асинхронную функцию, которая выполнится не сразу же, как только до неё дойдёт интерпретатор, а потом, при достижении определённых условий.
В качестве такого условия, можно прописать событие onload: как только скрипт загрузится на страницу, можно запускать какую-то другую функцию, использующую его данные.

В асинхронную функцию передаётся callback – функция обратного вызова, которая запускается при достижении нужных условий. Сама же функция callback принимает 2 аргумента: ошибку и собственно исполняющий код.
Если в результате выполнения произошла ошибка, то она передаётся первым аргументом, callback обрабатывает её и завершает работу.
Если ошибки не было, то её значение равняется null и callback работает в штатном режиме.

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Не удалось загрузить скрипт ${src}`));

  document.head.append(script);
}
```

Hexlet
Асинхронные функции никогда не возвращают результат асинхронной операции. Единственный способ получить результат — описать логику в колбеке.
Использование return всегда делает выход из АФ и возвращает undefined.
При этом инструкция return не вернёт результат, а выполнит выход из функции. Return можно использовать как guard expression, прекращая дальнейшие вычисления.

Как только происходит ошибка, мы *вызываем основной колбек и отдаём туда ошибку*. Если ошибка не возникла, то мы всё равно вызываем исходный колбек и передаём туда `null`. Вызывать его обязательно, иначе внешний код не дождётся окончания операции. Следующие вызовы, после вызова с ошибкой, больше не выполняются:

```
import fs from 'fs';

const unionFiles = (inputPath1, inputPath2, outputPath, cb) => {
  fs.readFile(inputPath1, 'utf-8', (error1, data1) => {
    if (error1) {
      cb(error1);
      return;
    }
    fs.readFile(inputPath2, 'utf-8', (error2, data2) => {
      if (error2) {
        cb(error2);
        return;
      }
      fs.writeFile(outputPath, `${data1}${data2}`, (error3) => {
        if (error3) {
          cb(error3);
          return;
        }
        cb(null); // не забываем последний успешный вызов
      });
    });
  });
}
```

Последний вызов можно сократить. Если в самом конце не было ошибки, то вызов cb(error3) отработает так же, как и cb(null), а значит, весь код последнего колбека можно свести к вызову cb(error3):

```
fs.writeFile(outputPath, `${data1}${data2}`, cb);
// что равносильно fs.writeFile(outputPath, `${data1}${data2}`, error3 => cb(error3));
```

Промисы

Объект promise

Promise - это объект, возвращаемый функцией, которая ещё не завершила свою работу.
Когда вызванная функция асинхронно завершает работу, этот объект переходит в соответствующее состояние и вызывает обработчик для дальнейшей работы с результатом асинхронной операции.

Объект Promise может находиться в трёх состояниях (свойство state, нет прямого доступа):

```
pending, ожидание      начальное состояние, не исполнен и не отклонён.
fulfilled, исполнено   операция завершена успешно.
rejected, отклонено    операция завершена с ошибкой.
```

Результат выполнения записывается в свойство result (нет прямого доступа) и может быть равен:

```
undefined  с самого начала
value      при вызове resolve(value)
error      при вызове reject(error)
```

Синтаксис создания промиса:

```
let promise = new Promise(executor);
let promise = new Promise(function(resolve, reject) { ... });


promise.then(
  function(result) { /* обработает успешное выполнение */ },
  function(error) { /* обработает ошибку */ }
);
```

Самому прописывать аргументы не надо: resolve и reject – функции, встроенные в JS.
В функции executor описывается выполнение асинхронной работы, а потом передаётся наружу через функции resolve или reject. Если что-то возвращать из функции через return, то результат будет проигнорирован.

Promise.resolve(value)
Возвращает промис, исполненный с результатом value. Вызывается, когда операция завершилась успешно и вернула результат своего исполнения в виде значения.
Promise.reject(reason)
Возвращает промис, отклонённый из-за reason. Вызывается, когда операция не удалась и возвращает значение, указывающее на причину неудачи, чаще всего объект ошибки. Если в промисе произошла ошибка, она вернётся через .reject

Промисы для любых функций
Чтобы снабдить свою функцию функциональностью промисов, нужно просто вернуть в ней объект Promise:

```
function myAsyncFunction(url) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest();
    xhr.open("GET", url);
    xhr.onload = () => resolve(xhr.responseText);
```

```
    xhr.onerror = () => reject(xhr.statusText);
    xhr.send();
  });
}
```

Пример из учебника (тут ошибка по ходу, .append должен быть за пределами функции)

```
function loadScript(src) {

  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Ошибка загрузки скрипта ${src}`));

    document.head.append(script);
  });
}
```

Обработчики решения

Результат, возвращённый из промиса, можно продолжить обрабатывать дальше через **обработчики решения**. Это методы промиса .then, .catch и .finally.

.then(onFulfilled, onRejected)

У этого обработчика два аргумента, они должны быть функциями-обработчиками результата выполнения промиса. Какая из них запустится, зависит от статуса выполненного промиса (value или err).
В первую функцию-обработчик передаётся результат успешного выполнения промиса, во вторую – ошибка выполнения промиса.
.then возвращает новый промис.
Если промис не был обработан (т.е. если аргументы .then – это не функции), то вернётся оригинальное значение.

```
promise.then(
  function(result) { /* обработает успешное выполнение */ },
  function(error) { /* обработает ошибку */ }
);
```

.catch(onRejected)

Добавляет функцию для обработки ошибки и возвращает новый промис.
.catch(f) является аналогом .then(null, f).

.finally()

Переданный в .finaly обработчик выполнится в любом случае, успешно ли завершился промис или с ошибкой. Хорошо подходит, например, для остановки индикатора загрузки, потому что его нужно остановить вне зависимости от результата.

Отличия от then(f,f):
- Обработчик, вызываемый из finally, не имеет аргументов. В finally мы не знаем, как был завершён промис. Это правильно, потому что обычно наша задача – выполнить «общие» завершающие процедуры.
- Обработчик finally «пропускает» результат или ошибку дальше, к последующим обработчикам.

Цепочка промисов

В обработчиках можно создавать промисы и возвращать их как обычный результат.

Явное создание промисов

Вообще, обработчики сами по себе возвращают результат в виде промиса, но в этом примере промис создаётся и возвращается явно вручную:

```
new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
})

.then(function(result) {
  alert(result); // 1
  return new Promise((resolve, reject) => { // (*)
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) { // (**)
  alert(result); // 2
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) {
  alert(result); // 4
});
```

Хороший комментарий: «Явно мы сами возвращаем промис, когда хотим не просто вернуть ответ, а еще и выполнить какую-то работу, которая занимает время. Тогда then будет ждать, когда наш созданный явный промис выполнится и отдаст ей результат».
Таким образом, выводятся 1, 2, 4, но сейчас между вызовами alert существует пауза в 1 секунду.

Цепочка .then

Можно выстраивать обработчики друг за другом. Так, например, скрипты будут загружены по очереди и в последнем скрипте можно использовать замыкание из первого:

```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;
    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Ошибка загрузки скрипта ${src}`));
    document.head.append(script);
  });
}

loadScript("/article/promise-chaining/one.js")
  .then(script => loadScript("/article/promise-chaining/two.js"))
  .then(script => loadScript("/article/promise-chaining/three.js"))
  .then(script => {
    // скрипты загружены, мы можем использовать объявленные в них функции
    one();
    two();
    three();
  });
```

«Пирамида» .then
Технически мы бы могли добавлять .then напрямую к каждому вызову loadScript, как в примере ниже.
Этот код делает то же самое: последовательно загружает 3 скрипта. Но он «растёт вправо», так что возникает такая же проблема, как и с колбэками.
Иногда всё же приемлемо добавлять .then напрямую, чтобы вложенная в него функция имела доступ к внешней области видимости. В примере выше самая глубоко вложенная функция обратного вызова имеет доступ ко всем переменным script1, script2, script3. Но это скорее исключение, чем правило (какое-то противоречие, потому что пример выше тоже вызывает функции из вложенных скриптов).

```
loadScript("/article/promise-chaining/one.js").then(script1 => {
  loadScript("/article/promise-chaining/two.js").then(script2 => {
    loadScript("/article/promise-chaining/three.js").then(script3 => {
      // эта функция имеет доступ к переменным script1, script2 и script3
      one();
      two();
      three();
    });
  });
});
```

Использование внешних переменных для «протаскивания» данных
Если соединять несколько then друг с другом, то не получится протянуть какие-то аргументы от первого вызова к последнему:

```
import { promises as fs } from 'fs';

const unionFiles = (inputPath1, inputPath2, outputPath) => {
  const result = fs.readFile(inputPath1, 'utf-8')
    .then((data1) => fs.readFile(inputPath2, 'utf-8'))
    .then((data2) => fs.writeFile(outputPath, `${data1?}${data2}`));
  return result;
};
```

Data 1 сюда уже не попадёт, она находится в локальной области видимости функции 1.
Выход из этой ситуации – после then создать переменные и протащить их от 1 вызова до конца:

```
import { promises as fs } from 'fs';

const unionFiles = (inputPath1, inputPath2, outputPath) => {
  let data1;

  return fs.readFile(inputPath1, 'utf-8')
    .then((content) => {
      data1 = content; <==
    })
    .then(() => fs.readFile(inputPath2, 'utf-8'))
    .then((data2) => fs.writeFile(outputPath, `${data1}${data2}`));
};
```

В этом месте содержимое промиса уже точно записывается в константу и не пропадает.

thenable
Обработчик может возвращать не промис, а любой объект, содержащий метод .then, такие объекты называют «thenable», и этот объект будет обработан как промис.
Смысл в том, что сторонние библиотеки могут создавать свои собственные совместимые с промисами объекты. Они могут иметь свои наборы методов и при этом быть совместимыми со встроенными промисами, так как реализуют метод .then.

Это позволяет добавлять в цепочки промисов пользовательские объекты, не заставляя их наследовать от Promise.
Вот пример такого объекта:

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve); // function() { native code }
    // будет успешно выполнено с аргументом this.num*2 через 1 секунду
    setTimeout(() => resolve(this.num * 2), 1000); // (**)
  }
}

new Promise(resolve => resolve(1))
  .then(result => {
    return new Thenable(result); // (*)
  })
  .then(alert); // показывает 2 через 1000мс
```

Промисы и ошибки

Управление в ближайший .catch
Если промис завершается с ошибкой, то управление переходит в ближайший обработчик ошибок. .catch не обязательно должен быть сразу после ошибки, он может быть после нескольких .then. Самый лёгкий путь перехватить все ошибки – это добавить .catch в конец цепочки.

```
fetch('url')
  .then()
  .then()
  .then()
  .catch(error => alert(error.message));
```

Невидимый try..catch
Вокруг функции промиса и обработчиков находится "невидимый try..catch". Если происходит исключение, то оно перехватывается, и промис считается отклонённым с этой ошибкой. Иными словами, любая ошибка в промисе автоматом делает его отклонённым.
Два кода ниже работают одинаково:

```
new Promise((resolve, reject) => {
  throw new Error("Ошибка!");
}).catch(alert);

new Promise((resolve, reject) => {
  reject(new Error("Ошибка!"));
}).catch(alert);
```

Можно нормально завершить цепочку промисов, даже если в ней была ошибка.
Для этого нужно после обработчика .catch продолжить по цепочке другие обработчики, даже если из .catch вернуть переданный в неё объект ошибки. В коде ниже считается, что ошибка обработана и всё ок, можно продолжать выполнение:

```
new Promise((resolve, reject) => {
```

```
  throw new Error("Ошибка!");
})
.catch(error => console.log("ошибка обработана"))
.then(() => console.log("then работает нормально"));

// ошибка обработана index.js:4:25
// then работает нормально
```

А в этом примере блок .catch тоже перехватывает ошибку, но не может обработать её, поэтому ошибка пробрасывается далее:

```
new Promise((resolve, reject) => {
  throw new Error("Ошибка!");
})

.catch(function(error) {
  console.log("Не могу обработать ошибку");
  throw error;
})

.then(function() {
  // не сработает
})

.catch(error => {
  // ничего не возвращаем => выполнение продолжается в нормальном режиме
});
```

Необработанные ошибки
Если забыть добавить .catch и не обработать ошибку в промисе, то скрипт упадёт, как если бы это была обычная необработанная ошибка: JavaScript-движок отслеживает такие ситуации и генерирует в этом случае глобальную ошибку.

В браузере мы можем поймать такие ошибки, используя событие unhandledrejection.
Если происходит ошибка, и отсутствует её обработчик, то генерируется это событие и соответствующий объект event содержит информацию об ошибке.

```
window.addEventListener('unhandledrejection', function(event) {
  // объект события имеет два специальных свойства:
  alert(event.promise); // [object Promise] - промис, который сгенерировал ошибку
  alert(event.reason); // Error: Ошибка! - объект ошибки, которая не была обработана
});
```

Promise API

В классе Promise есть 5 статических методов.

Promise.all

Ожидает исполнения всех промисов или отклонения любого из них.
Возвращает промис, который исполнится после исполнения всех промисов в iterable.
В случае, если любой из промисов будет отклонён, Promise.all будет также отклонён.

```
let promise = Promise.all([promise1, promise2, ...]);
```

Принимает массив промисов (любой перебираемый объект). Новый промис завершится, когда завершится весь переданный список промисов, и его результатом будет массив их результатов.
Если передан объект, который не является промисом, то он будет преобразован с помощью метода Promise.resolve.

При этом результат будет собран в таком же порядке, в каком промисы были переданы. Даже если передать несколько неодинаковых таймеров, результат будет с соблюдением очерёдности передачи:

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(console.log);

// [ 1, 2, 3 ]
```

.map и Promise.all
Часто применяемый трюк – пропустить массив данных через map-функцию, которая для каждого элемента создаст задачу-промис, и затем обернёт получившийся массив в Promise.all.

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://api.github.com/users/jeresig'
];

// Преобразуем каждый URL в промис, возвращённый fetch
let requests = urls.map(url => fetch(url));

// Promise.all будет ожидать выполнения всех промисов
Promise.all(requests)
  .then(responses => responses.forEach(
    response => console.log(`${response.url}: ${response.status}`)
));
```

Если попробовать распечатать этот массив после map, то он будет выглядеть так:

```
const promises = filepaths.map((filepath) => fs.readFile(filepath, 'utf-8'));
console.log(promises);
[
  Promise { <pending> },
  Promise { <pending> },
  Promise { <pending> },
  Promise { <pending> },
]
```

Ошибки и Promise.all
Если хотя бы один промис завершился с ошибкой, весь результат Promise.all будет помечен как ошибочный. Чтобы этого избежать, можно передавать в Promise.all промисы с повешенными на них обработчиками .catch, из которых уже возвращаются данные с пометкой об успешности:

```
const promises = filepaths.map((filepath) => fs.readFile(filepath, 'utf-8')
  .then((v) => ({ result: 'success', value: v }))
  .catch((e) => ({ result: 'error', error: e })));

const promise = Promise.all(promises);
```

Promise.allSettled

Ожидает завершения всех полученных промисов (как исполнения так и отклонения).
Возвращает промис, который исполняется когда все полученные промисы завершены, содержащий массив результатов исполнения полученных промисов.
Массив будет содержать объекты в формате { status: 'fulfilled / rejected', value: 'value' }.

```
let promise = Promise.allSettled([promise1, promise2, ...]);

выполнен
{ status: 'fulfilled', value: 'value' }
отклонён
{ status: 'rejected', reason: 'value' }
```

Promise.race

Возвращает результат или ошибку только того промиса, который выполнится быстрее всех. После этого остальные промисы игнорируются.

Promise.resolve(value)
Promise.reject(reason)

Редко используются в современном коде из-за синтаксиса async/await.
Возвращает исполненный/отклонённый промис с результатом value / reason.

Promise.resolve(value) создаёт успешно выполненный промис с результатом value. Этот метод используют для совместимости: когда ожидается, что функция возвратит именно промис.

```
let promise = new Promise(resolve => resolve(value));
```

Promise.reject(error) создаёт промис, завершённый с ошибкой error. На практике почти не используется.

```
let promise = new Promise((resolve, reject) => reject(error));
```

Промисификация

Промисификация – это преобразование функции на коллбэках в промисы.
Многие библиотеки основаны на колбэках, есть смысл их «промисифицировать», потому что промисы удобнее.

Существуют модули с гибкой промисификацией, например, es6-promisify или встроенная функция util.promisify в Node.js.

? Помните, промис может иметь только один результат, но колбэк технически может вызываться сколько угодно раз. Поэтому промисификация используется для функций, которые вызывают колбэк только один раз. Последующие вызовы колбэка будут проигнорированы.

```
// было
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Ошибка загрузки скрипта ${src}`));

  document.head.append(script);
}
```

```
// использование: loadScript('path/script.js', (err, script) => {...})


// стало
let loadScriptPromise = function(src) {
  return new Promise((resolve, reject) => {
    loadScript(src, (err, script) => {
      if (err) reject(err)
      else resolve(script);
    });
  })
}
// использование: loadScriptPromise('path/script.js').then(...)
```

Функция-обёртка для промисификации
Можно сделать универсальную обёртку для промисификации. Эта обёртка такая же, как функция выше, но если исходная f ожидает колбэк с большим количеством аргументов callback(err, res1, res2, ...), то при вызове promisify(f, true) результатом промиса будет массив результатов [res1, res2, ...]:
(не понимаю, как это работает)

```
// promisify(f, true), чтобы получить массив результатов
function promisify(f, manyArgs = false) {
  return function (...args) {

    return new Promise((resolve, reject) => {
      function callback(err, ...results) { // наш специальный колбэк для f
        if (err) {
          return reject(err);
        } else {
          // делаем resolve для всех results колбэка, если задано manyArgs
          resolve(manyArgs ? results : results[0]);
        }
      }

      args.push(callback);

      f.call(this, ...args);
    });
  };
};

// использование:
f = promisify(f, true);
f(...).then(arrayOfResults => ..., err => ...)
```

Микрозадачи

Обработчики .then/catch/finally всегда асинхронны, поэтому даже если в них передаётся готовое решение, они вызываются только после выполнения текущего синхронного кода (который находится под ними). Пример:

```
let promise = Promise.resolve(console.log('1'));  // выполнится сразу

promise.then(() => console.log('2'));             // всегда асинхронный

console.log('3');                                 // выполнится вторым

// 1, 3, 2
```

Почему .then срабатывает позже?
Асинхронные задачи требуют правильного управления, для этого стандарт предусматривает внутреннюю очередь PromiseJobs, более известную как «очередь микрозадач» (microtask queue).
Когда промис выполнен, его обработчики .then/catch/finally попадают в конец этой очереди. Если есть цепочка с несколькими .then/catch/finally, то каждый из них ставится в очередь, а потом выполняется, когда добавленные ранее в очередь обработчики выполнены.

Как сказано в спецификации:
- задачи, попавшие в очередь первыми, выполняются тоже первыми (FIFO).
- выполнение задачи происходит только в том случае, если ничего больше не запущено (?).

Теперь мы можем описать, как именно JavaScript понимает, что ошибка не обработана: необработанная ошибка возникает в случае, если ошибка промиса не обрабатывается в конце очереди микрозадач. Если мы забудем добавить `.catch`, то, когда очередь микрозадач опустеет, движок сгенерирует событие «unhandledrejection». Событие unhandledrejection возникает, когда очередь микрозадач завершена: движок проверяет все промисы и, если какой-либо из них в состоянии «rejected», то генерируется это событие.

Async/Await

Связка async/await нужна для того, чтобы работать с АФ как с обычными функциями. Это «синтаксический сахар» для получения результата промиса, более наглядный, чем promise.then.

Хотя при работе с async/await можно обходиться без promise.then/catch, иногда всё-таки приходится использовать эти методы (на верхнем уровне вложенности, например). Также await отлично работает в сочетании с Promise.all, если необходимо выполнить несколько задач параллельно.

async
Ключевое слово async ставится перед какой-нибудь функцией и такая функция всегда будет возвращать промис. Например, эта функция возвратит промис с результатом 1:

```
async function f() {
  return 1;
}

console.log(f());
// Promise { 1 }

f().then(console.log);
// 1
```

await
Ключевое слово await используется только внутри async-функций. Await заставит интерпретатор JavaScript ждать до тех пор, пока промис справа от `await` не выполнится. После чего он вернёт его результат и выполнение кода продолжится.

Подряд идущие `await` в рамках одной функции всегда выполняются строго друг за другом. Проще всего это понимать если представлять код как цепочку промисов, где каждая следующая операция выполняется внутри `then`.
Получается, что они выполняются последовательно, а не параллельно. Чтобы выполнение было параллельным, надо использовать функцию promise.all. Используя только *async/await* невозможно одновременно запускать несколько промисов.

```
let value = await promise;
```

В этом примере промис успешно выполнится через 1 секунду:

```
async function f() {
```

```
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("готово!"), 1000)
  });

  let result = await promise; // будет ждать, пока промис не выполнится (*)
  console.log(result); // "готово!"
}

f();
```

В данном примере выполнение функции остановится на строке (*) до тех пор, пока промис не выполнится. Это произойдёт через секунду после запуска функции. После чего в переменную result будет записан результат выполнения промиса.

Стрелочные функции
Тоже используются:

```
самостоятельная функция
const foo = async () => {…}

метод объекта
foo = async () => {…}
```

IIFE
Можно обернуть код в анонимную async-функцию:

```
(async () => {
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();
  ...
})();
```

Если у объекта можно вызвать метод then, этого достаточно, чтобы использовать его с await.

Обратите внимание, хотя await и заставляет JavaScript дожидаться выполнения промиса, это не отнимает ресурсов процессора. Пока промис не выполнится, JS-движок может заниматься другими задачами: выполнять прочие скрипты, обрабатывать события и т.п.

await нельзя использовать на верхнем уровне вложенности
из-за того, что await работает только внутри async–функций, так сделать не получится:

```
let response = await fetch('/article/promise-chaining/user.json');
let user = await response.json();

// SyntaxError
```

Но можно сделать анонимную функцию-обёртку, и всё заработает:

```
(async () => {
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();
  ...
})();
```

Использование с promise.all
Можно сделать так с использованием promise.all, и файлы начнут читаться одновременно, а не последовательно:

```
const unionFiles = async (inputPath1, inputPath2, outputPath) => {
  // Эти вызовы начинают чтение почти одновременно и не ждут друг друга
  const promise1 = fs.readFile(inputPath1, 'utf-8');
  const promise2 = fs.readFile(inputPath2, 'utf-8');

  // Теперь дожидаемся когда они оба завершатся
  // Данные можно сразу разложить
  const [data1, data2] = await Promise.all([promise1, promise2]);
  await fs.writeFile(outputPath, `${data1}${data2}`);
};
```

await работает с «thenable»–объектами
Как и promise.then, await позволяет работать с промис–совместимыми объектами. Идея в том, что если у объекта можно вызвать метод then, этого достаточно, чтобы использовать его с await.

Асинхронные методы классов
Для объявления асинхронного метода достаточно написать async перед именем:

```
class Waiter {
  async wait() {
    return await Promise.resolve(1);
  }
}

new Waiter()
  .wait()
  .then(alert); // 1
```

Обработка ошибок
Когда промис завершается успешно, await promise возвращает результат. Когда завершается с ошибкой – будет выброшено исключение, как если бы на этом месте находилось выражение throw. Если забыть добавить .catch, то будет сгенерирована ошибка «Uncaught promise error» и информация об этом будет выведена в консоль.

```
// такой код:
async function f() {
  await Promise.reject(new Error("Упс!"));
}

// Делает тоже самое, что и такой:
async function f() {
  throw new Error("Упс!");
}
```

Можно использовать с конструкцией try-catch и ошибки будут отловлены:

```
import { promises as fs } from 'fs';

const unionFiles = async (inputPath1, inputPath2, outputPath) => {
  try {
    const data1 = await fs.readFile(inputPath1, 'utf-8');
    const data2 = await fs.readFile(inputPath2, 'utf-8');
    await fs.writeFile(outputPath, `${data1}${data2}`);

  } catch (e) {
```

```
    console.log(e);
    throw e; // снова бросаем, потому что вызывающий код должен иметь возможность отловить ошибку
  }
};
```

Можно использовать метод промисов .catch уже после синтаксиса async:

```
async function f() {
  let response = await fetch('http://no-such-url');
}

// f() вернёт промис в состоянии rejected
f().catch(alert); // TypeError: failed to fetch
```

Hexlet

Традиционная задача про объединение файлов:

```
// Код на колбеках
import fs from 'fs';

fs.readFile('./first', 'utf-8', (error1, data1) => {
  if (error1) {
    console.log('boom!');
    return;
  }
  fs.readFile('./second', 'utf-8', (error2, data2) => {
    if (error2) {
      console.log('boom!');
      return;
    }
    fs.writeFile('./new-file', `${data1}${data2}`, (error3) => {
      if (error3) {
        console.log('boom!');
      }
    });
  });
});


// Код на промисах
import { promises as fsPromises } from 'fs';

let data1;
fsPromises.readFile('./first', 'utf-8')
  .then((d1) => {
    data1 = d1;
    return fsPromises.readFile('./second', 'utf-8');
  })
  .then((data2) => fsPromises.writeFile('./new-file', `${data1}${data2}`))
  .catch(() => console.log('boom!'));


// То же самое с Async/Await
const unionFiles = async (inputPath1, inputPath2, outputPath) => {
  // Как и в примере выше, эти запросы выполняются строго друг за другом,
  // хотя при этом не блокируется программа, это значит, что другой код тоже может выполняться во время этих запросов)
  // В реальной жизни чтение файлов лучше выполнять параллельно через Promise.all

  const data1 = await fsPromises.readFile(inputPath1, 'utf-8');
  const data2 = await fsPromises.readFile(inputPath2, 'utf-8');
  await fsPromises.writeFile(outputPath, `${data1}${data2}`);
```

```
};
```

Промисы и fetch

Сложная тема, смотреть её вместе с fetch темой.

Комментарий:
«Чтобы понять примеры, внимание обратите на след. цитаты. Из этого и предыдущего уроков:
Если промис в состоянии ожидания, обработчики в .then/catch/finally будут ждать его. Однако, **если промис уже завершён, то обработчики выполнятся сразу**.
Обработчик handler, переданный в .then(handler), может вернуть промис. В этом случае дальнейшие обработчики ожидают, пока этот промис выполнится, и затем получают его результат.

Важно понимать, что асинхронный код обрабатывает только промис. .then/catch/finall - обычные функции.
в примерах же, все функции которые делают асинхронные запросы, уже либо обернуты в промис (лоадСкрипт) либо возвращают промис как результат реализации (фетч). Поэтому в цепочке .then они выполняются друг за другом, но если сделать к примеру пару .then с синхронным кодом, потом отправить в .then запрос на сервер, без промиса, и снова продолжить .then с синхронным - паравоз будет нестись не ожидая ответа».

Промисы часто используются, чтобы делать запросы по сети. Базовый синтаксис:

```
let promise = fetch(url);
```

Запрос работает так:
1. код запрашивает url по сети и возвращает промис.
2. Промис ждёт, пока сервер вернёт пришлёт заголовки ответа и возвращает объект response. У этого объекта много своих методов, например: response.text(), response.json()
3. Чтобы прочитать полный ответ, надо вызвать метод response.text(). Он тоже возвращает промис, который выполняется, когда данные полностью загружены с удалённого сервера, и возвращает эти данные.

```
fetch('https://learn.javascript.ru/article/promise-chaining/user.json')
  .then( (response) => response.text() )
  .then( (text) => console.log(text) ); // текст


fetch('https://learn.javascript.ru/article/promise-chaining/user.json')
  .then((response) => response.json())
  .then((user) => console.log(user) );  // объект
```

Плученные данные можно отправить на Гитхаб и скачать аватар пользователя:

```
fetch('https://learn.javascript.ru/article/promise-chaining/user.json')
  .then((response) => response.json())
  // запрос на гитхаб
  .then(obj => fetch(`https://api.github.com/users/${obj.name}`)
  .then(response => response.json())
  // на основе полученных данных сделать изображение
  .then(githubUser => new Promise(function(resolve, reject) {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);
```

```
    // через 3 секунды удалить
    setTimeout(() => {
      img.remove();
      resolve(githubUser); // (**)
    }, 3000);
  }))

  .then(githubUser => alert(`Закончили показ ${githubUser.name}`))
);
```

обработчик .then в строке // 1 будет возвращать new Promise, который перейдёт в состояние «выполнен» только после того, как в строке // 2 таймер вызовет resolve(githubUser). Соответственно, следующий по цепочке .then будет ждать этого.
Как правило, все асинхронные действия должны возвращать промис.

///

fs.promises API

Документация по FS: перевод.
Для того, чтобы работать с функциями было ещё проще, в нодовской библиотеке File System есть соответствующий раздел:
«The fs.promises API provides an alternative set of asynchronous file system methods that return Promise objects rather than using callbacks. The API is accessible via require('fs').promises»
Очень полезная и удобная штука. К ней можно обращаться так:

```
import { promises as smth } from 'fs';

или https://nodejs.org/api/fs.htmls
let fs = require('fs');
let fsPromises = fs.promises;

или
fs.promises.method(arg, option)
```

У этого метода есть свои такие же методы, как и у обычного fs, только в них не надо передавать коллбек. Просто аргументы и свойста, а затем обработчик .then

```
Схема
fs.promises.method(arg, option).then( callback_ok, callback_error) )

Пример работы
const copy = (src, dest) =>
  promises.readFile(src, 'utf-8').then((content) => fs.writeFile(dest, content));
```

fs.readFile
fs.writeFile

fs.readdir - чтение содержимого директории
fs.readdirSync(dirpath);

fs.stat - информация о файле
fs.statSync(path.join(dirpath, fname))])

Задача
Реализуйте и экспортируйте асинхронную функцию `reverse`, которая изменяет порядок расположения строк в файле на обратный:

```
// one
// two
reverse(filepath);

// two
// one
```

```
import { promises as fsPromises } from 'fs';

export const reverse = (file) => fsPromises.readFile(file, 'utf-8')
  .then((data) => String(data))
  .then((str) => str.split('\n').reverse().join('\n')) // здесь готовый для записи текст
  .then((str) => fsPromises.writeFile(file, str));

препод
export const reverse = (filepath) => fs.readFile(filepath, 'utf-8')
  .then((data) => {
    const preparedData = data.split('\n').reverse().join('\n');
    return fs.writeFile(filepath, preparedData);
  });
```

Сетевые запросы

Для сетевых запросов используется термин AJAX (**A**synchronous **J**avaScript **A**nd **X**ML). Это старое понятие, здесь есть указание на XML, это один из способов сделать запрос.
Сетевые запросы можно сделать с помощью:
- Fetch
- XML

Fetch

Fetch API предоставляет интерфейс для получения ресурсов (в том числе по сети). Fetch обеспечивает обобщённое определение объектов Request и Response.

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

url      Строка с прямым указанием адреса или Request объект (объект ответа).
options  Объект с опциями, которые нужно применить к запросу.


Некоторые опции
method:   метод запроса (GET, POST и т.д.
headers:  заголовки, содержащиеся в объекте.
body:     тело запроса, может быть: Blob, BufferSource, FormData, URLSearchParams, или
USVString объектами. GET или HEAD запрос не может иметь тела.


Обёртка для запросов
const getResourse = async (url) => {
  const response = await fetch(url);

  if (!response.ok) {
    throw new Error(`Could not fetch ${url}, received ${res.status}`);
  }

  const body = await response.json();
  return body;
};


Типичная реализация
let response = await fetch(url, options); // завершается с заголовками ответа
let result = await response.json();       // читать тело ответа в формате JSON
```

```
fetch(url, options)
  .then(response => response.json())
  .then(result => /* обрабатываем результат */)
```

Принцип работы и методы

1. fetch(url) возвращает промис.
Работать с ним надо с помощью методов .then .catch или async-await. Когда промис успешно выполнен, он возвращает объект response.
Промис завершается с ошибкой, если fetch не смог выполнить HTTP-запрос, например при ошибке сети или если нет такого сайта. HTTP-статусы 404 и 500 не являются ошибкой.

2. Объект response.
Response – это самостоятельный API с набором методов.
Сначала объект получает только заголовки и статус запроса (тело ответа приходит потом).
Можно применить следующие методы:

```
Response          API в MDN
Response.ok       true/false, выполнился ли запрос успешно (коды ответа 200–299).
Response.status   код ответа.
Response.headers  объект Headers, который содержит заголовки ответа (у него свои методы).


Headers API
Headers()
Creates a new Headers object.
Headers.get()
Returns a ByteString sequence of all the values of a header within a Headers object with a
given name.
Headers.entries()
Returns an iterator allowing to go through all key/value pairs contained in this object.
Headers.has()
Returns a boolean stating whether a Headers object contains a certain header.
Headers.keys()
Returns an iterator allowing you to go through all keys of the key/value pairs contained in
this object.

response.headers.get('Content-Type')

for (let [key, value] of response.headers) {
  alert(`${key} = ${value}`);
}
```

Чтобы работать с телом ответа, нужно применить к объекту response методы из интерфейса Body.
В результате будет возвращён ещё один промис, с содержимым которого уже можно работать.
Можно выбрать только один метод чтения ответа.

```
Body           API в MDN предоставляет методы, относящиеся к телу запроса/ответа.
Body.json()    Декодирует ответ в формате JSON.
Body.text()    Читает ответ и возвращает как обычный текст.
Body.blob()    Возвращает объект как Blob (бинарные данные с типом).

Body.arrayBuffer()
Возвращает ответ как ArrayBuffer (низкоуровневое представление данных).
Body.formData()
Takes a Response stream and reads it to completion. It returns a promise that resolves with a
FormData object.
Body.body
A simple getter used to expose a ReadableStream (en-US) of the body contents. Это объект
ReadableStream, с помощью которого можно считывать тело запроса по частям.
```

Опции запроса

Для любых запросов, кроме GET, нужно использовать опции запроса.
Про опции можно прочитать в документации про fetch здесь.

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

Некоторые опции
method    метод запроса (GET, POST и т.д.)
headers   заголовки, содержащиеся в объекте.
body      тело запроса: Json, FormData, Blob/BufferSource и другие.
          GET или HEAD запрос не может иметь тела.

mode      режим, например, cors, no-cors или same-origin.
```

Headers
Для установки заголовка запроса нужно использовать опцию headers. Её значение – это объект с определёнными ключами и значениями.
Посмотреть возможные HTTP-заголовки этого объекта можно в Документации здесь.

```
const options = {
  headers: {
    Authentication: 'secret'
    'Content-Type': 'application/json;charset=utf-8'
  },
};

const response = fetch(url, options);


Content-Type
Строка для определения типа ресурса. Список таких типов здесь.
```

Есть список запрещённых HTTP-заголовков, которые нельзя установить. Они обеспечивают корректную работу протокола HTTP, поэтому контролируются только браузером.

Body
данные для отправки (тело запроса) в виде текста, FormData, BufferSource, Blob или UrlSearchParams

Другие опции
Объект с опциями, содержащий пользовательские настройки запроса, может содержать:

```
credentials:
Полномочия: omit, same-origin или include. Для автоматической отправки куки для текущего
домена, эта опция должна быть указана. Начиная с Chrome 50, это свойство также принимает
экземпляр класса FederatedCredential (en-US) или  PasswordCredential (en-US).

cache:
Режим кеширования запроса default, no-store, reload, no-cache, force-cache или only-if-cached.

redirect:
Режим редиректа: follow (автоматически переадресовывать), error (прерывать перенаправление
ошибкой) или manual (управлять перенаправлениями вручную). В Chrome по дефолту
стоит follow (ранее, в Chrome 47, стояло manual).

referrer:
USVString, определяющая no-referrer, client или a URL. Дефолтное значение - client.
```

referrerPolicy:
Определяет значение HTTP заголовка реферера. Может быть: no-referrer, no-referrer-when-downgrade, origin, origin-when-cross-origin, unsafe-url.

integrity:
Содержит значение целостности субресурсов (subresource integrity) запроса (например, sha256-BpfBw7ivV8q2jLiT13fxDYAe2tJllusRSZ273h2nFSE=).

keepalive:
Эта опция может быть использована, чтобы разрешить запросу "пережить" страницу. Получение ресурсов с флагом keepalive - это альтернатива Navigator.sendBeacon() API.

signal:
Экземпляр объекта AbortSignal; позволяет коммуницировать с fetch запросом и, если нужно, отменять его с помощью AbortController.

POST-запросы

```
const options = {
  method: post,
  body: json    // это указание на переменную
  headers: { 'Content-Type': 'application/json;charset=utf-8' },
};

const promise = fetch(url, options);
```

body
Данные, которые надо отправить

headers
Сообщить браузеру, как правильно интерпритировать данные. В примере указано, что это – json.

Content-Type
Строка для определения типа ресурса. Список таких типов здесь.
По умолчанию будет text/plain;charset=UTF-8.

Отправка изображения
Пропустил это, потому что не шарю в Blob.

Примеры

```
Запрос и работа с заголовками
fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits')
  .then(response => console.log( response.headers ));

Запрос тела ответа и его декодирование в формате json
fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits')
  .then(response => response.json())
  .then(commits => console.log(commits[0].author.login)
);


Обработать сразу несколько адресов + обработать ошибки
Если возникает какая-то ошибка – возвратить null
async function getUsers(names) {
```

```
  // перегнать все имена в промисы-запросы
  let requests = names.map(el => fetch(`https://api.github.com/users/${el}`)
    .then(
      // correctResponse
      response => {
        if (response.status !== 200) return null;
        return response.json();
      },
      // errorResponse
      error => null)
    );

  // дождаться выполнения всех запросов и их обработки
  let result = await Promise.all(requests);
  return result;
}
console.log(getUsers(['iliakan', 'remy', 'no.such.users']));


Чтение и перебор заголовков
(async () => {
  let response = await fetch('https://api.github.com/repos/javascript-
tutorial/en.javascript.info/commits');
  // чтение одного заголовка
  console.log(response.headers.get('Content-Type'));
  // перебрать все заголовки
  for (let [key, value] of response.headers) {
    console.log(`${key} = ${value}`);
  }
})();
```

FormData

FormData – это API, которое автоматом создаёт объект из полей формы.
Объекты FormData позволяют конструировать наборы пар ключ-значение, представляющие поля формы и их значения, которые в дальнейшем можно отправить с помощью метода send().

Объект будет соответствующим образом закодирован и отправлен с заголовком Content-Type: form/multipart. Будет такой же формат на выходе, как если бы отправлялась обыкновенная форма с encoding установленным в "multipart/form-data".

```
Конструктор
let formData = new FormData(form);


const options = {
  method: 'POST',
  body: new FormData(formElem)
}
```

Методы FormData

```
FormData в MDN

FormData.append(name, value)
formData.append(name, blob, fileName)
Создать или обновить пару ключ-значение.

FormData.delete(name)
```

```
Удаляет поле.

FormData.get(name)
Возвращает значение поля с именем name.

FormData.getAll(name)
Возвращает массив всех значений ассоциированных с переданным ключом.

FormData.has(name)
true / false

FormData.set()
formData.set(name, blob, fileName)
Как append, но сначала удаляет все уже имеющиеся поля с ключом name.

Можно перебирать в цикле:
for(let [name, value] of formData) {…}


FormData.keys()
Возвращает iterator , который позволяет пройтись по всем ключам для каждой пары "ключ-значение"
, содержащимся внутри объекта FormData
FormData.entries()
Возвращает iterator который позволяет пройтись по всем парам "ключ-значение", содержащимся
внутри объекта FormData
FormData.values()
Возвращает iterator , который позволяет пройтись по всем значениям , содержащимся в
объекте FormData
```

Технически форма может иметь много полей с одним и тем же именем name, поэтому несколько вызовов append добавят несколько полей с одинаковыми именами.

Примеры

Отправка простой формы

В этом примере серверный код не представлен. Он принимает POST-запрос с данными формы и отвечает сообщением.

```
<form id="form-elem" />

formElem.onsubmit = async (e) => {
  e.preventDefault();

  const options = {
    method: 'POST',
    body: new FormData(formElem)
  }

  let response = await fetch(url, options);
  let result = await response.json();

  console.log(result.message);
};
```

Отправка формы с файлом

Файл удобнее всего отправлять вместе с формой. В этом примере ничего не меняется.

```
<form id="form-elem">
  <input type="file" name="picture" accept="image/*">
</form>

formElem.onsubmit = async (e) => {
  e.preventDefault();
```

```
  const options = {
    method: 'POST',
    body: new FormData(formElem)
  }

  let response = await fetch(url, options);
  let result = await response.json();

  console.log(result.message);
};
```

Отправка формы с Blob-данными
Кликай на ссылку.

Обратите внимание на то, как добавляется изображение Blob:

```
formData.append("image", imageBlob, "image.png");

Это как если бы в форме был input type="file" со значениями:
name = "image"
наименование файла – image.png
imageBlob – сам файл.
```

Сервер прочитает и данные и файл, точно так же, как если бы это была обычная отправка формы.

Fetch: ход загрузки

Статья тут, не читал.

Fetch: прерывание запроса

Не читал

Fetch: запросы на другие сайты

Статья тут.

SWAPI

Получение данных

Чтобы получить данные с сервера, надо выполнить два вызова:

```
const getResourse = async(url) => {
  const res = await fetch(url);
  const body = await res.json();
  return body;
}

getResourse('https://swapi.dev/api/people/1')
  .then((body) => {
    console.log(body);
  });
```

Обработка ошибок

Есть 2 типа ошибок: сетевые ошибки и ошибки, которые сообщает сервер.

Сетевые происходят, когда мы вызываем сервер, но сервер не ответил. Возможно, проблема с интернетом или сервер недоступен (сломался, лёг).
В этом случае промис в первой строке (в примере выше) будет rejected.

```
const getResourse = async(url) => {
  const res = await fetch(url);
  const body = await res.json();
  return body;
}

getResourse('https://swapi.dev/api/people/1404')
  .then((body) => {
    console.log(body);
  })
  .catch((err) => {
    console.error('Could not fetch', err);
  });
```

Ошибки, которые сообщает сервер.
Если персонаж с указанным id не был найден, сервер должен вернуть 404.
В этом случае блок catch из примера выше не сработает. Логика fetch очень простая: мы вызвали сервер, получили ответ 404, но при этом сервер вернул валидный ответ. С т.з. fetch его функция выполнена отлично: мы послали запрос и мы получили ответ.

А вот интерпретация ответа – это вопрос клиентского кода. Очевидно, что надо сделать так, чтобы ошибочные коды сервера давали такой же результат, как и ошибка сети. Чтобы написать такой код, надо проверить статус ответа.

```
const getResourse = async(url) => {
  const res = await fetch(url);
  if (!res.ok) {
    throw new Error(`Could not fetch ${url}, received ${res.status}`);
  }
  const body = await res.json();
  return body;
}

getResourse('https://swapi.dev/api/people/1404')
  .then((body) => {
    console.log(body);
  })
  .catch((err) => {
    console.error('Could not fetch', err);
  });
```

Клиент для API

Классы-сервисы нужны для того, чтобы инкапсулировать весь сетевой код и изолировать его от остальных частей приложения в отдельный класс-сервис. Это хорошая практика при разработке больших приложений, потому что весь код работы с данными находится в одном месте и в будущем можно реализовать что-нибудь типа кеширования, сменить источник данных или реализовать дополнительную логику фильтрации.

Для остальной части приложения этот класс-сервис – это просто асинхронный источник данных. Компоненты не должны знать, откуда именно берутся данные.

_apiBase

Это та часть сетевого адреса, которая присутствует во всех запросах. Чтобы её каждый раз не дублировать, достаточно сохранить её в приватном поле и приклеивать к передаваемому url.

async getAllPeople()
Метод автоматом вычленяет массив персонажей и возвращает его. Если не поставить async, то пришлось бы использовать промисы вне класса при использовании этого метода.
Строчка res.results – это свойство возвращаемого сервером объекта, в котором хранится массив объектов-персонажей.

В примере ниже будет распечатываться список имён персонажей:

```
export default class SwapiService {

  _apiBase = 'https://swapi.dev/api';

  async getResourse(url) {
    const res = await fetch(`${this._apiBase}${url}`);
    if (!res.ok) {
      throw new Error(`Could not fetch ${url}, received ${res.status}`);
    }
    const body = await res.json();
    return body;
  }


  async getAllPeople() {
    const res = await this.getResourse(`/people/`);
    return res.results;
  }
  async getPerson(id) {
    const res = await this.getResourse(`/people/${id}/`);
    return res;
  }


  async getAllPlanets() {
    const res = await this.getResourse('/planets/');
    return res;
  }
  async getPlanet(id) {
    const res = await this.getResourse(`/planets/${id}`);
    return res;
  }


  async getAllStarships() {
    const res = await this.getResourse(`/starships/`);
    return res;
  }
  async getStarship(id) {
    const res = await this.getResourse(`/starships/${id}`);
    return res;
  }

};


const swapi = new SwapiService();

swapi.getAllPeople().then((people) => {
    people.forEach((p) => console.log(p.name));
  });

swapi.getPerson(3).then((p) => {
```

```
  console.log(p.name);
});
```

Трансформация данных

API не всегда передаёт данные в нужном формате. Например, вместо camelCase может использоваться нижнее подчёркивание, или приложению нужно использовать только несколько свойств, а не все свойства, которые передаются с сервера.

В этом случае, в клиенте можно трансформировать данные, полученные с сервера, чтобы передавать их компонентам приложения в нужном формате. Для этого надо дописать ещё несколько приватных функций.

Такие трансформации нельзя делать в одной функции, которая получает данные от APi, потому что надо отделять модель данных от API и модель данных приложения. Такая практика применяется для крупных проектов со сложными моделями данных, которые могут изменяться.

Swapi не передаёт id, поэтому его надо вытащить из свойства url через регулярное выражение.

```
  _extractId(item) {
    const idRegExp = /\/([0-9]*)\/$/;
    return item.url.match(idRegExp)[1];
  }

  _transformPlanet = (planet) => {
    return {
      id: this._extractId(planet),
      name: planet.name,
      population: planet.population,
      rotationPeriod: planet.rotation_Period,
      diameter: planet.diameter
    }
  }
```

Итоговый вариант класса:

```
export default class SwapiService {

  _apiBase = 'https://swapi.dev/api';

  async getResourse(url) {
    const res = await fetch(`${this._apiBase}${url}`);
    if (!res.ok) {
      throw new Error(`Could not fetch ${url}, received ${res.status}`);
    }
    const body = await res.json();
    return body;
  }


  async getAllPeople() {
    const res = await this.getResourse(`/people/`);
    return res.results.map(this._transformPerson);
  }
  async getPerson(id) {
    const person = await this.getResourse(`/people/${id}/`);
    return this._transformPerson(person);
  }


  async getAllPlanets() {
```

```
    const res = await this.getResourse('/planets/');
    return res.results.map(this._transformPlanet);
  }
  async getPlanet(id) {
    const planet = await this.getResourse(`/planets/${id}`);
    return this._transformPlanet(planet);
  }


  async getAllStarships() {
    const res = await this.getResourse(`/starships/`);
    return res.results.map(this._transformStarship);
  }
  async getStarship(id) {
    const starship = await this.getResourse(`/starships/${id}`);
    return this._transformStarship(starship);
  }


  _extractId(item) {
    const idRegExp = /\/([0-9]*)\/$/;
    return item.url.match(idRegExp)[1];
  }

  _transformPlanet = (planet) => {
    return {
      id: this._extractId(planet),
      name: planet.name,
      population: planet.population,
      rotationPeriod: planet.rotation_Period,
      diameter: planet.diameter
    }
  }

  _transformStarship= (starship) => {
    return {
      id: this._extractId(starship),
      name: starship.name,
      model: starship.model,
      manufacturer: starship.manufacturer,
      costInCredits: starship.costInCredits,
      length: starship.length,
      crew: starship.crew,
      passengers: starship.passengers,
      cargoCapacity: starship.cargoCapacity
    }
  }

  _transformPerson = (person) => {
    return {
      id: this._extractId(person),
      name: person.name,
      gender: person.gender,
      birthYear: person.birthYear,
      eyeColor: person.eyeColor
    }
  }

};
```

Поскольку сервис возвращает готовый объект с нужными полями, то в state можно передавать информацию так:

```
state = {
  planet: {}
```

```
};

componentDidMount() {
  this.updatePlanet();
}

updatePlanet() {
  const id = Math.floor(Math.random()*21) + 1;
  this.swapiService.getPlanet(id)
    .then(this.onPlanetLoaded);
}

onPlanetLoaded = (planet) => {
  this.setState({planet})
}
```

Спиннер

Скачать спиннер можно тут:

```
loading.io
Готовые индикаторы загрузки. Мой любимый тут.
```

Простой вариант
Для того, чтобы спиннер работал только когда компонент ждёт данные с сервера:

В state компонента, который подгружает данные, добавить ещё одно поле: loading. Как только компонент инициализируется, будет показан спиннер, потому что он только обращается к данным.

Первый способ:
в функции Render прописать условие показа спиннера: если loadind – true, то возвращать элемент спиннер.
Проблема – спиннер будет заменять собой весь элемент.

Второй способ:
чтобы спиннер отображался внутри основного элемента, нужно все дочерние элементы, которые ожидают данные с сервера, убрать в отдельный приватный фрагмент.
В JSX если значение какого-либо элемента – null, то он не отображается. Вместо if-else можно использовать тернарный оператор для спиннера.

В функции, которая срабатывает при получении данных сервера, поменять true на false одновременно с разбросом этих данных.

```
state = {
  planet: {},
  loading: true
};

onPlanetLoaded = (planet) => {
  this.setState({
    planet,
    loading: false
  });
}


render() {
  const { planet, loading } = this.state;

  const spinner = loading ? <Spinner /> : null;
  const content = !loading ? <PlanetView planet={planet} /> : null;
```

```
  return (
    <div className="random-planet jumbotron rounded">
      {spinner}
      {content}
    </div>
  );
}


const PlanetView = ({ serverData }) => {
  return (
    <React.Fragment>
      <img>
      <div>
    </React.Fragment>
  );
}
```

Обработка ошибок

Чтобы из-за неудачного запроса приложение не падало целиком, надо добавить блок .catch в компонент.
Дополнительно надо создать компонент, который будет заниматься отображением ошибки.

Компонент-ошибка:

```
import icon from './death-star.png';

const ErrorIndicator = () => {
  return (
    <div className='error-indicator'>
      <img src={icon} alt='error icon'/>
      <span className='boom'>BOOM!</span>
      <span>
        something has gone terribly wrong
      </span>
      <span>
        (but we already sent droids to fix it)
      </span>
    </div>
  );
};
```

В state компонента добавляем поле error, которое по умолчанию false.
Добавляем функцию onError, котрая занимается изменением state и помещается в catch.
hasData – данные будут отображаться, когда нет ни ошибки, ни загрузки.

```
  onPlanetLoaded = (planet) => {
    this.setState({
      planet,
      loading: false,
      error: false
    });
  }

  onError = (err) => {
    this.setState({
      error: true,
      loading: false
    });
  }

  updatePlanet() {
```

```
    const id = Math.floor(Math.random()*21) + 1;
    this.swapiService.getPlanet(id)
      .then(this.onPlanetLoaded)
      .catch(this.onError);
  }

  render() {
    const { planet, loading, error } = this.state;

    const hasData = !(loading || error);

    const errorMessage = error ? <ErrorIndicator /> : null;
    const spinner = loading ? <Spinner /> : null;
    const content = hasData ? <PlanetView planet={planet} /> : null;

    return (
      <div className="random-planet jumbotron rounded">
        {errorMessage}
        {spinner}
        {content}
      </div>
    );
  }
}
```

Все методы жизненного цикла

```
Mounting
Когда срабатывает
Компонент создаётся и первый раз отображается на странице.
Если этот метод вызван, значит элементы уже гарантированно находятся на странице.
Что делать (не использовать для этого всего конструктор):
- проводить инициализацию компонента,
- делать сетевые запросы,
- получать данные в компонент,
- работать с DOM.

Что вызывается:
constructor() => render() => componentDidMount()


Updates
Когда срабатывает
Компонент получает обновления, т.е. это либо:
- пришли новые свойства
- изменился state (вызван setState)
При этом функция срабатывает после того, как state обновлён.
Что делать:
Запрашивать новые данные для обновлённых свойств.
Что вызывается:
render() => componentDidUpdate(prevProps, prevState)
  if (this.props.personId !== prevProps.personId) {
    this.updatePerson();
  }
}

Unmounting
Когда срабатывает
Во время удаления компонента со страницы . В момент вызова DOM-элемент всё ещё будет находиться
на странице.
Что делать:
- очищать те ресурсы, с которыми работал компонент
```

```
- останавливать запущенные таймеры
- останавливать запросы к серверу
Что вызывается:
componentWillUnmount()


Error
В компоненте произошла ошибка, которая не была поймана раньше.
Работает с ошибками жизненного цикла и метода rendering.
Это не замена try-catch.
Что делать:
Отключить ветку, в которой произошла ошибка.
Что вызывается:
componentDidCatch(error, info)
error – типичная ошибка JS
info – детали ошибки, характерные для React
```

componentDidMount()

Получение данных для itemList
В State хранится список предметов, который будет запрошен через сеть. По умолчанию этот список – null.
В render этот список передаётся из state.
В Render есть условие: если список – null (т.е. ещё не получен по сети), то отобразить spinner.

renderItems(arr)
Метод,который получает массив персонажей (от сетевого запроса) и через map делает из них li-элементы.

onItemSelected(id)
Функция определена в App передаётся в ItemList через props.

```
export default class ItemList extends Component {

  swapiService = new SwapiService();

  state = {
    peopleList: null
  };

  componentDidMount() {
    this.swapiService.getAllPeople()
      .then((peopleList) => {
        this.setState({
          peopleList
        });
    });
  }

  renderItems(arr) {
    // return arr.map((person) => {
    return arr.map(({id, name}) => {
      return (
        <li
          className="list-group-item"
          key={id}
          onClick={() => this.props.onItemSelected(id)}
        >
          {name}
        </li>
      );
    });
  }
```

```
  render() {
    const { peopleList } = this.state;

    if (!peopleList) {
      return <Spinner />
    }

    const items = this.renderItems(peopleList);

    return (
      <ul className="item-list list-group">
        {items}
      </ul>
    );
  }
}
```

componentDidUpdate()

Компонент PersonDetails должен обновляться каждый раз при выборе персонажа из списка слева.

В State записывается текущий выбранный персонаж. По умолчанию – null. Текущий персонаж передаётся из App. В App он хранится в State.

updatePerson()
Функция, которая занимается сетевым запросом и отрисовкой данных. Также перезаписывает state: поле person, которое обновляется после получения данных с сервера, и поле loading.

componentDidUpdate
Если этот метод будет менять что-то, что в конечном счёте повлечёт обновление props компонента (в моём случае props компонента обновляется, потому что props получает информацию из App State), то обязательно внутри должно быть условие на проверку предыдущего props и нового. Иначе обновления будут запускаться по кругу бесконечно.

```
export default class PersonDetails extends Component {

  swapiService = new SwapiService();

  state = {
    person: null,
    loading: false
  }

  componentDidMount() {
    this.updatePerson();
  }

  componentDidUpdate(prevProps) {
    if (this.props.personId !== prevProps.personId) {
      this.updatePerson();
    }
  }

  updatePerson() {
    const { personId } = this.props;
    if (!personId) {
      return;
    }

    this.setState({
      loading: true
```

```
    });

    this.swapiService.getPerson(personId)
      .then((person) => {
        this.setState({ person, loading: false });
      });
  }


  render() {
    const { person, loading } = this.state;

    const selectMessage = !person ? <span>Select a person from a list</span> : null;
    const spinner = loading ? <Spinner /> : null;
    const personData = (!loading && person) ? PersonView(person) : null;

    return (
      <div className="person-details card">
        {selectMessage}
        {spinner}
        {personData}
      </div>
    );
  }
};


const PersonView = (props) => {
  const { id, name, gender, birthYear, eyeColor } = props;

  return (
    <React.Fragment>
      <img className="person-image"
          src={`https://starwars-visualguide.com/assets/img/characters/${id}.jpg`}
          alt="character"/>

        <div className="card-body">
          <h4>{name}</h4>

          <ul className="list-group list-group-flush">
            <li className="list-group-item">
              <span className="term">Gender</span>
              <span>{gender}</span>
            </li>
            <li className="list-group-item">
              <span className="term">Birth Year</span>
              <span>{birthYear}</span>
            </li>
            <li className="list-group-item">
              <span className="term">Eye Color</span>
              <span>{eyeColor}</span>
            </li>
          </ul>

        </div>
    </React.Fragment>
  )
}
```

componentWillUnmount()

Вызывается перед тем, как компонент удалится (компонент всё ещё находится в dom-дереве). Используетя для того, чтобы очистить ресурсы, с которыми работал компонент.

Например, если был запущен таймер и его надо отменить. Можно отменять текущие запросы к серверу или отписываться от веб-сокетов.
Если компонент работает со сторонней библиотекой и этой библиотеке тоже надо сделать очистку, то этот метод – самое лучшее место, чтобы написать такой код.

```
componentDidMount() {
  this.updatePlanet();
  this.interval = setInterval(() => this.updatePlanet(), 5000);
}

componentWillUnmount() {
  clearInterval(this.interval);
}
```

componentDidCatch()

Отлавливает ошибки из функций, которые отвечают за корректный рендеринг компонента.
Если в одном из компонентов на странице выбросить ошибку, то всё приложение ляжет: будет показан пустой экран. В консоле будет виден код ошибки, но пользователь ничего не поймёт.
React даёт возможность обрабатывать ошибки и ограничивать их область действия.

В случае, если ошибка произойдёт, то отключится только вышестоящий по иерархии компонент, в котором был метод componentDidCatch, а всё остальное будет работать.

У метода есть 2 аргумента: (error, info)

В state компонента добавлятся новый флаг hasError: false, который говорит, была ли ошибка или нет. А когда componentDidCatch – true.

```
  state = {
    showRandomPlanet: true,
    selectedPerson: null,
    hasError: false
  };

  componentDidCatch() {
    console.log('componentDidCatch()');
    this.setState({ hasError: true });
  }

  render() {
    if (this.state.hasError) {
      return <ErrorIndicator />
    }
```

Выводы

1. React ничего не знает о работе с сервером, это задача других библиотек.

2. Сетевой код следует изолировать от кода компонентов

3. Если необходимо, трансформировать данные надо до того, как их получит компонент

4. Приложение должно уметь обрабатывать состояния «загрузка» и «ошибка»

5. Разделять ответственность компонентов: логику и рендеринг

</>

React

React
JavaScript-библиотека с открытым исходным кодом для разработки пользовательских интерфейсов. И это именно библиотека для создания UI (в смысле, инструменты для создания, а не готовые компоненты), а не фреймворк.

JSX
JavaScript XML (JSX) — это расширение синтаксиса JavaScript, которое позволяет использовать HTML-подобный синтаксис для описания структуры интерфейса. В курсе сказано, что он позволяет комбинировать JS с разметкой. Таким образом, код и UI находятся рядом и это помогает описывать логику.

Создание проекта

```
Установка приложения из репозитория

npm install -g create-react-app
Установить приложение для использования React

npx create-react-app todo
Создать react-проект
Будет создан каталог с react-проектом, в него скачаются все необходимые пакеты.


Команды после создания проекта
npm start      Запустить компиляцию, открыть проект в браузере, запустится локальный сервер
npm run build  Bundles the app into static files for production.
npm test       Starts the test runner.


Ограничение на установку скриптов
Если команды выше не работают, надо отключить ограничение на установку скриптов
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser  Отключить
Set-ExecutionPolicy Restricted -Scope CurrentUser    Вернуть назад


Подключение библиотек
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render( element, container, [callback] )

element      элемент для отрисовки
container    где отрисовывать элемент
```

React
Библиотека, которая говорит трансплаеру, как перегнать синтаксис JSX в обычный JS-код.

ReactDOM
Библиотека, которая может преображать virtual dom в реальный дом. Рендерит элементы на странице, чтобы их отобразил браузер.
Этот код написан на JSX, его перегоняет babel. Чтобы переписать этот код на чистом JS, надо сделать так:

```
const el = React.createElement('h1', null, 'Hello World!!');
```

Структура проекта
Компоненты выносят в отдельные файлы и сохраняют в каталоге components. Один компонент – один файл.
HTML, CSS и JS компоненты называются одинаково.
Используется kebab-case для наименования файлов.
Не забыть импортировать react и сделать export default.

```
Каталоги
Public    хранит статичные ресурсы
Src       действующий код

index.js
import React from 'react';
import ReactDOM from 'react-dom';
import TodoList from './components/todo-list';
```

Элементы, компоненты, фрагменты

Элемент – это самый маленький кирпич.
По сути, это присваивание в переменную элемента, который описан через JSX.

```
Создать один элемент
const el = <h1>Hello, World</h1>;

Создать сразу несколько вложенных элементов
const el = (
  <div>
    <h1>My Todo List</h1>
    <input placeholder="search"/>
    <ul>
      <li>Learn React</li>
      <li>Build App</li>
    </ul>
  </div>
);
```

Компоненты – это независимые блоки пользовательского интерфейса, которые могут иметь своё поведение.
Создать реакт-компонент – это создать функцию, которая возвращает элемент. Они нужны для того, чтобы легко переиспользовать компонент в других частях приложения и прикручивать им логику.

Внутри такой функции может быть только один элемент-корень, но внутри корня можно вложить сколько угодно других элементов.
Наименование функции-компонента пишется с большой буквы. Это требование реакта, чтобы отличать обычные теги от react-компонентов.
Вставку null / undefined, true / false реакт проигнорирует.

Контролируемые элементы (в компонентах).
Элемент является контролируемым, если он управляется из state. По сути, это связь между state и данными элемента. Для того, чтобы сделать элемент контролируемым, нужно сделать так, чтобы значение элемента устанавливалось из state компонента. Например, если текстовое поле в форме очищается из state после события onSubmit.

```
Создание компонента
const TodoList = () => {
  return (
    <ul>
      <li>Learn React</li>
      <li>Build App</li>
    </ul>
  );
};


Использование компонента
используется так, будто это обычный элемент
const el = (
  <div>
    <TodoList />
  </div>
);
```

Fragment
Во фрагментах элементы могут иметь несколько корневых элементов (в отличие от компонентов).
Поэтому фрагмент – это специальный элемент, который служит обёрткой, когда надо поместить несколько элементов в один контейнер.

```
Можно импортировать вместе с React
import React, { Fragment } from 'react';

const PlanetView = () => {
  return (
    <Fragment>
      <img className="planet-image" alt="" src={`https`} />
      <div> … </div>
    </Fragment>
  );
};


Можно использовать без именованного импорта
const PlanetView = () => {
  return (
    <React.Fragment>
      <img className="planet-image" alt="" src={`https`} />
      <div> … </div>
    </React.Fragment>
  );
};
```

Использование JSX

В JSX можно использовать как обычные html-теги, так и имена react-компонентов.

Кроме обычной html-разметки, можно вставлять js-выражения или переменые.
Такой код пишется в фигурных скобках.

Можно вставлять только str, num, bool, null, undefined и react-элементы.
Игнорируется вставка null, undefined, true, false. Это можно использовать с тернарным оператором.

Вставлять объекты нельзя.
Вставлять массивы с элементами можно.

Все переданные значения становятся обычным текстом. Это позволяет делать безопасные вставки.

```
<TodoList />   Компонент добавляется в JSX
{ loginBox }   Переменная JS, которая хранит реакт-элемент или текст, вставляется в JSX
{ newDate() }  Не сработает, потому что объект. Надо привести к строке.


Использование JS выражений
const TodoList = () => {
  const items = ['Learn React', 'Build App'];
  return (
    <ul>
      <li>{ items[0] }</li>
      <li>{ (new Date()).toString() }</li>
    </ul>
  );
};


null сочетается с тернарным оператором
const App = () => {
  const loginBox = <span>Log in please</span>
  return (
    <div>
      { isLoggedIn ? null : loginBox }
    </div>
  );
};


Вставка массива
const elements = [li, li, li…];
<ul> { elements } </ul>


JSX безопасно вставляет HTML как текст
const value = '<script> alert('!') </script>';
return (
  <div> { value } </div>
);
```

Атрибуты и свойства компонентов

Атрибуты
Наименование атрибутов – строго в camelCase.
Стандартные атрибуты прописываются как обычно.
Кроме того, атрибуты можно передавать как переменные.
Если свойству или атрибуту не передать значение явно, то его значение будет true.

```
2 атрибута, которые называются в JSX по-другому
className   вместо class
htmlFor     вместо for
```

```
Значения из переменных
const searchText = 'Type here to search';
<input placeholder={ searchText }/>;


Значение атрибута по умолчанию – true
return <input
  disabled
  disabled = { true }
/>;
```

Свойства
В реакт-компоненты можно передавать свойства. Они записываются в компонент так же, как атрибуты, но значением свойства может быть всё, что угодно. После передачи, свойства доступны в функции-конструкторе через пропсы. В функцию-конструктор props передаются первым параметром в виде объекта.

Наименование свойств – в camelCase.
Если свойству или атрибуту не передать значение явно, то его значение будет true.

Набор свойств можно сначала сохранить как объект в переменную, а потом распаковать его через spread.

```
Передача свойств
<Component key1={ value1 } key2={ value 2} />


Использование в функции-конструкторе
const Component = (props) => {
  props.key1;
  props.key2;
}

const component = (props) => {
  const {key1, key2} = props;
}

const Component = ( {key1, key2} ) =>


Передача всех свойств через spread
Переменная хранит набор свойств
const item = { label: 'Make Awesome App', important: true }

Обычная передача
<TodoListItem label={item.label} important={item.important} />

Через …rest
<TodoListItem { ...item } />
```

CSS и стили

Инлайн-стили описываются в объекте, сохраняются в переменную.
Переменная вставляется в атрибут style.

```
const searchStyle = { fontSize: '20px' };

return <input style={ searchStyle } placeholder={ searchText } />;
```

Webpack позволяет импортировать **css-файлы** со стилями прямо в JS файл.

```
import './todo-list.css';

Bootstrap
CDN подключается здесь.
```

Жизненный цикл

lifecycle hooks – специальные функции, которые Реакт автоматом вызывает на каждом этапе жизненного цикла.

Этапы жизненного цикла:

```
Mounting
Когда срабатывает
Компонент создаётся и первый раз отображается на странице.
Если этот метод вызван, значит элементы уже гарантированно находятся на странице.
Что делать (не использовать для этого всего конструктор):
- проводить инициализацию компонента,
- делать сетевые запросы,
- получать данные в компонент,
- работать с DOM.

Что вызывается:
constructor() => render() => componentDidMount()


Updates
Когда срабатывает
Компонент получает обновления, т.е. это либо:
- пришли новые свойства
- изменился state (вызван setState)
При этом функция срабатывает после того, как state обновлён.
Что делать:
Запрашивать новые данные для обновлённых свойств.
Что вызывается:
render() => componentDidUpdate(prevProps, prevState)
  if (this.props.personId !== prevProps.personId) {
    this.updatePerson();
  }
}

Unmounting
Когда срабатывает
Во время удаления компонента со страницы . В момент вызова DOM-элемент всё ещё будет находиться
на странице.
Что делать:
- очищать те ресурсы, с которыми работал компонент
- останавливать запущенные таймеры
- останавливать запросы к серверу
Что вызывается:
componentWillUnmount()


Error
В компоненте произошла ошибка, которая не была поймана раньше.
Работает с ошибками жизненного цикла и метода rendering.
Это не замена try-catch.
Что делать:
Отключить ветку, в которой произошла ошибка.
Что вызывается:
componentDidCatch(error, info)
error – типичная ошибка JS
info – детали ошибки, характерные для React
```

Некоторые методы жизненного цикла компонента в порядке срабатывания:

```
componentDidMount()
componentDidUpdate()
componentWillUnmount()
componentDidCatch()
```

Граница ошибок

У проекта может быть несколько уровней компонентов, и если ошибка поймана, то вся цепочка вложенных компонентов может быть отключена. Если поймать ошибку на самом верхнем уровне Root, то выключится вся страница. Если на любом из нижестоящих – то только нижестоящий, а остальные продолжат работу.

Принцип создания таких отсекаемых компонентов:

```
import ErrorIndicator from 'error-indicator';
export default class My_component extends Component {

  state = {
    hasError:  false;
  }
  componentDidCatch() {
    this.setState({ hasError: true });
  }

  render() {
    if (this.state.hasError) {
      return <ErrorIndicator />;
    }

    return (
      …
    );
  }
}
```

Это называется границей ошибок, error bounding. Поскольку они ловят ошибки ниже себя по иерархии и ограничевают область действия этих ошибок.

```
<Root>             <-- componentDidCatch()
  <Header>
    <Menu>
  </header>

  <LeftWidget>     <-- componentDidCatch()
    <Title>
    <Content>
  </LeftWidget>

  <RightWidget>    <-- componentDidCatch()
    <Title>
    <Content>
  </RightWidget>
</Root>
```

Это похоже на обычный try-catch. По аналогии, если в componentDidCatch() будет выброшено ислючение, оно будет передано в следующий componentDidCatch().

Два аспекта:
- componentDidCatch() работает только с ошибками в методах жизненного цикла и в рендеринге.
А ошибки, например, в eventListener в каком-нибудь onClick не будут тут обрабатываться, или в асинхронных коллбеках, даже если они были инициированны в методах жизненного цикла.

- это не замена стандартным проверкам. Его роль – ловить действительно непредвиденные ошибки и в идеале он вообще не должен работать. Они позволяют одному компоненту закрашиться, но при этом всё остальное приложение будет жить.

Классы

Главное отличие классов от функций – наличие свойства state, в котором можно хранить состояние.
Состояние – это разного рода переменные, которые влияют на внешний вид элемента или его потомков.
Если компоненту нужно работать со своим внутренним состоянием, (например, с счётчиком кликов или менять цвет элементов в зависимости от чего-то), то нужно использовать класс, а не функцию-конструктор.

```
Импорты
import React from 'react';
export default class TodoListItem extends React.Component {

import React, {Component} from 'react';
export default class TodoListItem extends Component {


render()
Этот метод возвращает элемент. Он делает то же самое, что функция-конструктор и прописывается
так же, кроме аргументов. Доступ к аргументам .render получает так:

render() {
  const {label, important = false} = this.props;
  return (…);
```

State, внутреннее состояние

state

В реакте внутреннее состояние компонентов хранится в специальном поле state. Это основная причина для использования классов.
Он обязательно должен быть объектом, в котором можно сохранить любую информацию.

Если данные нужно использовать в нескольких компонентах, их надо хранить в родительском компоненте.

```
старомодный синтаксис
constructor() {
  super();
  this.state = { … };
}

новый синтаксис
эквивалентно инициализации в конструкторе (поля классов)
state = { … };
```

setState()

В реакте есть важное правило: state нельзя изменять после инициализации напрямую. Изменения в state вносятся через setState().
В функцию setState передаются только те свойства объекта, которые надо изменить. SetState() говорит реакту: состояние этого компонента изменилось, перерендери его, заново вызови функцию render().

В функции render() прописана вся логика, часть которой опирается на state.
Например, в state можно указать, выполнен ли элемент списка задач. Render прочитает state и, в зависимости от данных в state, выберет класс done или удалит элемент.

setState может работать асинхронно.
Чтобы учитывать асинхронность, надо возвращать анонимную функцию, которая принимает в качестве первого параметра текущий state. Это означает, что текущий код надо выполнить только тогда, когда текущий state будет в финальном состоянии и его можно будет использовать для того, чтобы вычислить новый state (к вопросу об асинхронности). Этот подходи используется, когда новое значение state опирается на старое значение.
Например, если надо поменять true или false на обратное значение, увеличить счётчик на единицу.
Если новое состояние никак не зависит от старого состояния, то не надо использовать функцию, можно просто передавать объект с новым состоянием.
Первым параметром в функцию setState автоматом передаётся state, его можно деструктуризировать сразу.

В примере ниже в функцию передаётся объект с изменениями, которые надо внести в state:

```
onMarkImportant = () => {
  this.setState( (state) => {
    return { important: !state.important };
  });
};

onMarkImportant = () => {
  this.setState( ({important}) => {
    return { important: !important };
  });
};

упрощённо
onMarkImportant = () => {
  this.setState( { important: true } );
}
```

Обработка событий

Главная задача – правильно передать this и не наебаться.

```
Обработчики-атрибуты
Всё как в обычных html: onClick = { функция }



Обработчик в экземпляре класса
constructor() {
  super();
  this.onLabelClick = () => {
    console.log(`Done: ${this.props.label}`)
  }
}

render() {
  const {label, important = false} = this.props;
  …
```

```
  <span
    onClick={ this.onLabelClick } >  без вызова() функции!
  </span>
```

```
Ещё варик
Забыл, как называется новый синтаксис. Вписывается без конструктора, просто так в теле

export default class TodoListItem extends Component {
  onLabelClick = () => {
    console.log(`Done: ${this.props.label}`)
  }
```

React фишули

Пометить li как важный или выполненный в todo list

```
const { important, done } = this.props;

let classNames = 'todo-list-item';

if (done) classNames += ' done';
if (important) classNames += ' important';

return <span className={classNames} />
```

React API

Children

props.children
Второй способ передать в компоненты свойства – это записать их в «тело» тега между открывающим и закрывающим тегом.
Доступ к таким свойствам осуществляется через: this. props.children

Для работы с children есть специальный API React.Children.
Документация тут.

Каждого children можно скрывать, оборачивать, делать что угодно. Потому что компонент не обязательно должен использовать child в том виде, в котором они переданы.
Изменять children нельзя. Для изменений, их надо скопировать и менять копии. Для этого существует React.cloneElement().

```
this.props.children
Обращение ко всему, что было передано в теле элемента
```

```
React.Children.map(children, function(el, index))
React.Children.forEach(children, function[(thisArg)])
React.Children.count(children)
React.Children.only(children)
React.Children.toArray(children)


const itemList = (
  <ItemList
    onItemSelected={this.onPersonSelected}
    getData={this.swapiService.getAllPeople}>

      {(i) => (
        `${i.name}, ${i.birthYear}`
      )};

  </ItemList>
);
```

Context api

Специальное хранилище данных, которое позволяет не пробрасывать эти данные через пропсы от вышестоящих в иерархии компонентов к нижестоящим, которые эти данные применяют.

Контекст нужен для того, чтобы решить проблему «глобальных» данных.
Вместо того, чтобы передавать props через все слои приложения, данные можно передавать через контекст.
С помощью контекста можно сделать так, чтобы компоненты не создавали объекты сервиса, а получали этот один объект.

Создаются 2 компонента:
Provider – в котором в свойстве value указываются данные, которые надо протащить.
Consumer – используется для получения данных. Должен возвращать функцию.

```
Создание файла-компонента для контекста
React.createContext();
Значение по умолчанию, если consumer не сможет найти никакой контекст.
Возвращает пару: Provider и Consumer.

const {
  Provider: SwapiServiceProvider,
  Consumer: SwapiServiceConsumer
} = React.createContext();


Кратко
// в модуле
const { Provider, Consumer } = React.createContext();

// в родительском файле
<Provider value={someValue}>
  // провайдер оборачивает часть приложения
</Provider>

// в файле-потомке любой вложенности
<Consumer>
  { (someValue) => <MyComponentData={someValue} /> }
</Consomer>



Пример
```

```
Provider ипортируется в вышестроящий компонент (app.js) и оборачивает нижестоящие компоненты,
получает value={}, который надо в них протащить.

<SwapiServiceProvider value = {this.swapiService}>
  <el>
    <el>
    <details>
  <el>
</SwapiServiceProvider>


Consumer используется в компоненте, который использует этот value.
Его  тоже надо импортировать.
Он возвращает функцию, которая первым параметром получит value.
Можно сразу деструктиризировать (в примере для наглядности без деструктуризации).

const PersonDetails = ({itemId}) => {
  return (
    <SwapiServiceConsumer>
      { эта скобка обязательно должна стоять здесь, а не выше
        (swapiService) => {
          return (
            <ItemDetails
              itemId={itemId}
              getData={swapiService.getPerson}
              getImageUrl={swapiService.getPersonImage}>
            </ItemDetails>
          );
        }
      }
    </SwapiServiceConsumer>
  );
};
```

Hooks

Справочник API хуков здесь.

Хук – это специальная функция, которая даёт возможность функциональным компонентам использовать почти все возможности компонентов-классов:
- использование state;
- методы жизненного цикла;
- передача контекста;
- какие-то другие возможности.

Правила хуков:
- Можно вызывать только из react-компонентов и собственных хуков.
- Нельзя вызывать внутри циклов, условных операторов или вложенных функций.
- Не работают в компонентах-классах.
- Не покрывают все методы жизненного цикла: при помощи хуков нельзя создать componentDidCatch.

Документация здесь.

React содержит несколько встроенных хуков, таких как useState. Вы также можете создавать собственные хуки.

```
Основные хуки
import React, { useState, useContext, createContext } from 'react';


useState
используется для обновления состояния
const [state, setState] = useState(initialState);

старое значение нельзя мутиовать,
но можно использовать его для возврата из функции нового значения
setState((oldState) => oldState + 1);

объекты нужно деструктуризировать
setState((oldState) => {…oldState, name: value})


useContext
const MyContext = createContext();
const value = useContext(MyContext);

Используется для получения значения из контекста.
Контекст – это пропс value ближайшего <MyContext.Provider>.


useEffect
сработает только при изменении value
useEffect( () => console.log('Вызвана'), [value]);



Дополнительные хуки
useReducer
useCallback
useMemo
useRef
useImperativeHandle
useLayoutEffect
useDebugValue
```

useState

Используется для обновления состояния.

```
import React, { useState, useContext, createContext } from 'react';

useState
const [state, setState] = useState(initialState);

state
Переменная хранит какое-то текущее значение
setState()
Это функция, в которую надо передать новое значение для изменения state.
Старое значение нельзя мутиовать.
```

Как и в случае использования «классического» state, нельзя мутировать старое значение. Если надо получить новое значение на основе старого (вычесть из старого или прибавить к нему), то надо использовать функцию и, после обработки логики, вернуть из неё новое значение.
Первый аргумент функции – старый state.

```
setState((oldState) => oldState + 1);
```

```
Если в state хранится объект с данными, объекты можно деструктуризировать и дописать в конце
новое значение, которое надо обновить:
setState((oldState) => {…oldState, name: value})
```

Также, чтобы обновить поле в «классическом» state, нужно было вернуть из setState только одно обновлённое свойство: setState({ поле: значение })
Хук работает иначе.
Если в state хранится объект с данными и надо обновить только одно его поле, то объект можно деструктуризировать и дописать в конце новое значение, которое надо обновить:

```
setState((oldState) => {…oldState, name: value})
```

useContext

Единственное отличие хука от обычного контекста – это способ получения данных:
1. Вызвать функцию `createContext()` с сохранить вызов в переменную. Эта функция возвращает объект.
2. В любом месте создать элемент-провайдер `<Переменная.Provider value='123'>`.
В элемент-провайдер обернуть элемент, который будет потребителем контекста.

Всё, как обычно.
3. Единственное отличие – потребитель получает контекст так: `value = useContext(Переменная)`.

```
import React, { useContext, createContext } from 'react';
import ReactDOM from 'react-dom';


const MyContext = createContext();

const App = () => {
  return (
    <MyContext.Provider value="Hello World">
      <Child />
    </MyContext.Provider>
  );
};

const Child = () => {
  // какой именно контекст надо использовать
  const value = useContext(MyContext);
  return <p>{value}</p>;
};

ReactDOM.render(<App />, document.getElementById('root'));
```

useEffect

Документация тут.
Заменяет компоненты жизненного цикла:
- componentDidMount
- componentDidUpdate
- componentWillUnmount

В зависимости от того, какие переданы аргументы в функцию, хук будет срабатывать на разных этапах жизненного цикла компонента, в котором он находится:

```
componentDidUpdate и componentDidMount
```

```
useEffect( () => console.log('Вызвана'));


componentDidMount
useEffect( () => console.log('Вызвана'), []);

сработает только при изменении value
useEffect( () => console.log('Вызвана'), [value]);


componentWillUnmount
useEffect(() => () => console.log('unmount'), []);


componentDidMount и componentWillUnmount
useEffect( () => {
  console.log('Создана');
  return () => console.log('Удалена');
}, []);
```

Принимает 2 аргумента:
1. функцию, которая будет срабатывать при определённых условиях;
2. необязательно – массив массив данных. Будет срабатывать только если данные из массива обновились. Пустой массив – useEffect сработает только при componentDidMount.

Очистка предыдущих эффектов.
Если в первом аргументе сделать return и вернуть какую-то функцию, то она будет срабатывать при удалении или обновлении компонента.
Чтобы функция срабатывала только при удалении компонента, надо передать пустой массив.

useCallback

Документация здесь.
Возвращает мемоизированный колбэк (кеширует функцию).
Мемоизация - сохранение результатов выполнения функций для предотвращения повторных вычислений.

Хук вернёт мемоизированную версию колбэка, который изменяется только, если изменяются значения одной из зависимостей. Это полезно для предотвращения ненужных рендеров.

Если данные в массиве не изменяются, то useCallback вернёт ту же ссылку на ту же функцию.

```
const foo = useCallback(fn, [deps])

аналог:
const foo = useMemo(() => fn, [deps])
```

useMemo

Документация здесь.
Возвращает мемоизированное значение (кеширует значение).

Первый аргумент – функция, которая создаёт и возвращает какое-то значение.
Второй – те данные, от которых это значение зависит. Если передать пустой массив, то это означает, что значение не зависит ни от каких данных и будет вычисляться только 1 раз.

Если в коде есть какое-то значение, которое надо передавать в useEffect, чтобы сравнить старое состояние и новое, то этот хук понадобится.

```
const memorized = useMemo( () => a, [a]);
```

Хуки и загрузка данных

```
const PlanetInfo = ({id}) => {

  const [name, setName] =  useState(null);

  useEffect( () => {
    let cancelled = false;

    fetch(`https://swapi.dev/api/planets/${id}/`)
      .then(res => res.json())
      .then(data => !cancelled && setName(data.name));
    return () => cancelled = true;
    }, [id]);

  return (
    <div>{id} - {name}</div>
  );
};
```

Создание хуков

Хук – функция, имя которой начинается с use. Внутри используются стандартные хуки.

```
const useCustomHook = (id) => {

  const [name, setName] = useState(null);

  useEffect( () => {
    let cancelled = false;
    fetch(`https://swapi.dev/api/planets/${id}/`)
      .then(res => res.json())
      .then(data => !cancelled && setName(data.name));
    return () => cancelled = true;
    }, [id]);

  return name;
};

const PlanetInfo = ({id}) => {
  const name = useCustomHook(id);
  return <div>{id} - {name}</div>;
};
```

Паттерны React

Большинству приложений необходимы вспомогательные компоненты:
- ErrorBoundry
- контекст или несколько контекстов
- HOC для работы с контекстом (withXyzService)

Эти и другие вспомогательные компоненты лучше создать до начала работы над основным функционалом приложения.

Из подкурса react-redux, каркасс react-redux приложения:

```
ReactDOM.render(

  доступ к Redux Store
  <Provider store={store}>

    обработка ошибок в компонентах ниже
    <ErrorBoundry>

      передаёт сервис через Context API
      <BookstoreServiceProvider value={bookstoreService}>

        из пакета react-router
        <Router>
          <App />

        </Router>
      </BookstoreServiceProvider>
    </ErrorBoundry>
  </Provider>,
```

props.children
Второй способ передать в компоненты свойства – это записать их в «тело» тега между открывающим и закрывающим тегом.
Доступ к таким свойствам: this. props.children

```
this.props.children
Обращение ко всему, что было передано в теле элемента


const itemList = (
  <ItemList
    onItemSelected={this.onPersonSelected}
    getData={this.swapiService.getAllPeople}
  >
    {(i) => (
      `${i.name}, ${i.birthYear}`
    )};

  </ItemList>
);
```

ErrorBoundry
Крутая штука – отлавливатель ошибок на любых компонентах. См. ErrorBoundry в проекте SWAPI.

```
class ErrorBoundry extends Component {
  state = {
    hasError: false
  };

  componentDidCatch() {
    this.setState({
      hasError: true
    });
  }

  render() {
    if (this.state.hasError) {
```

```
      return <ErrorIndicator />
    }
    return this.props.children;
  }
};


Любой элемент, на котором надо отловить ошибки, оборачивается в это добро:
<ErrorBoundry>
  <PersonDetails/>
</ErrorBoundry>
```

HOC – компоненты высшего порядка
Это функции-обёртки.
Этот компонент берёт на себя обязанности, о котрых не нужно заботиться внутреннему компоненту. Например, получать данные по сети, отображать спиннер или компонент с ошибкой.

Можно сказать, что обёртка занимается менеджментом данных. В неё можно передавать любой компонент, для которого она выполняет рутинные операции, а компонент будет только отображать данные. В неё выносится вся логика, котора была в оригинальном компоненте: state, componentDidMount, спиннер.

Раньше оригинальный компонент получал данные из state. Поскольку state теперь находится в компоненте-обёртке, теперь он будет получать данные из props.
Сопутствующие данные и функции (типа что именно запрашивать по сети), необходимые для оборачиваемого компонента, можно передать в hoc аргументами (пропсами).

```
const hoc = (MyComponent, [options]) => {
  return class extends Component {
    state {…}
    componentDidMount() {…}
    getData() {…}

    render() {
      const { data } = this.state;
      if (!data) return <Spinner />
      return <MyComponent {...this.props} />
    }
  }
}

const MyWrappedComponent = hoc(InnerComponent);
```

Композиция функций
Композиция – применение одной функции к результату другой.
Результат работы одной функции передаётся в другую функцию.

Компоненты высшего порядка – это обычные функции, которые возвращают компоненты, используя метод композиции. Так можно применять несколько эффектов HOC.

```
const comp = (x) => f( g(x) );
```

HOC-context
Обязанность получать данные из контекста можно вынести в компонент высшего порядка.

```
const withValueFromContext = (Wrapped) => {
```

```
  return (

    <Consumer>
      { (value) => <Wrapped value={value} /> }
    </Consumer>

  );
};
```

Зачем нужны паттерны?

Задача паттернов – избегать копирования кода и правильно переиспользовать части поведения компонентов.

Одинаковые компоненты могут иметь разное содержание: список персонажей, кораблей, планет.
Компоненты будут отличаться данными с сервера, но процесс получения и обработки данных (паттерн создания) у них одинаковый:
- запрос данных с сервера при создании;
- отрисовка этих данных;
- обработка ошибок с error indicator.

Чтобы избежать копипасты и не плодить одинаковые компоненты, можно сделать один универсальный компонент, который будет просто получать разные данные и одинаково работать с ними.

Использование функций

Функции погут инкапсулировать получение данных по сети и передавать их через пропсы. Эта часть про то, как можно скрыть получение данных.

itemList может стать универсальным компонентом, который работает с разными данными, не только со списком персонажей. Важно! Универсальным станет только ItemList, т.е. только списки слева, но не PersonDetails.

Если представить, что itemList спользуется для запроса списка планет или кораблей, то вот их сходства и различия:

swapiService
У всех присутствует
RenderItems
не изменяется.

componentDidMount
В зависимости от того, какую сущность надо отобразить, выполяется разный запрос (метод) swapiService.
Единственное, что меняется у компонентов – функция для запроса данных:
- getAllPeople()
- getAllStarships()
- getAllPlanets()

Обновляют состояние setState они одинаково. Логичный шаг – вынести получение данных наружу из компонента.

Что делать:
1. SwapiService удаляется из компонента itemList. Функцию запроса данных можно вынести наружу. Вместо того, чтобы создавать экземпляр класса внутри компонента, этот компонент получит функцию getData из свойств.

getData тоже будет возвращать промис. Компонент будет получать данные из этой функции, устанавливать их в качестве своего state.
Все peopleList заменяются на общее itemList, потому что компонент будет заниматься не только персонажами, а всем, чем угодно.

Было:

item-list.js

```
  state = {
    peopleList: null
  };

  swapiService = new SwapiService();

  componentDidMount() {
    this.swapiService.getAllPeople()
      .then((peopleList) => {
        this.setState({
          peopleList
        });
    });
  }
```

Стало:

```
  state = {
    itemList: null
  };

  componentDidMount() {
    const { getData } = this.props;

    getData()
      .then((itemList) => {
        this.setState({
          itemList
        });
    });
  }
```

2. сделать методы SwapiService стрелочными функциями, чтобы this не терялся:

swapi-service.js

```
getResourse = async (url) => {

getAllPeople = async () => {
getPerson = async(id) => {

getAllPlanets = async () => {
getPlanet = async (id) => {

getAllStarships = async () => {
getStarship = async (id) => {

_extractId = (item) => {

_transformPlanet = (planet) => {
_transformStarship = (starship) => {
_transformPerson = (person) => {
```

3. Изменить способ вызова универсального компонента.
Важно! Универсальным стал только ItemList, т.е. списки слева, но не PersonDetails.
Такие же изменения надо внести в компонент people-page, потому что там тот же самый код.

Было:

app.js

```
      <div className="row mb2">
        <div className="col-md-6">
          <ItemList onItemSelected={this.onPersonSelected} />
        </div>
        <div className="col-md-6">
          <PersonDetails personId={this.state.selectedPerson} />
        </div>
      </div>
```

Стало:

```
      <div className="row mb2">
        <div className="col-md-6">
          <ItemList
            onItemSelected={this.onPersonSelected}
            getData={this.swapiService.getAllPlanets}
          />
        </div>
        <div className="col-md-6">
          <PersonDetails personId={this.state.selectedPerson} />
        </div>
      </div>

      <div className="row mb2">
        <div className="col-md-6">
          <ItemList
            onItemSelected={this.onPersonSelected}
            getData={this.swapiService.getAllStarShips}
          />
        </div>
        <div className="col-md-6">
          <PersonDetails personId={this.state.selectedPerson} />
        </div>
      </div>
```

Render-функции

Паттерн, в котором в компонент в качестве пропса передаётся функция, которая занимается рендерингом всего компонента или его части. Такая функция возвращает строку, react-элемент или разметку jsx.
Это чем-то похоже на функцию .filter, которая решает, что оставить (в данном случае, что отобразить).

Допустим, надо сделать так, чтобы в списке персонажей рядом с именем отображались также пол и возраст, а в списке планет – диаметр, для кораблей – модель. Т.е. должна быть какая-то дополнительная информация.

До сих пор все элементы списков отображались одинаково. Внутри компонента item-list есть функция renderItems. Она печатает только свойство name и это свойство есть у всех сущностей.

item-list.js

```
  renderItems(arr) {
    return arr.map(({id, name}) => {
      return (
        <li
          className="list-group-item"
          key={id}
          onClick={() => this.props.onItemSelected(id)}
        >
          {name}
        </li>
      );
```

```
    });
  }
```

Теперь требования изменились и в зависимости от типа сущностей надо отображать разную информацию. Один из способов решить эту задачу – использвать render-функцию и передавать её через пропсы.

Функция renderItem() (внутри компонента item-list) получает на вход item и возвращает то, что надо будет отрисовать.
Такие же измнеенения надо внести в компонент people-page, потому что там тот же самый код.

app.js

```
      <div className="row mb2">
        <div className="col-md-6">
          <ItemList
            onItemSelected={this.onPersonSelected}
            getData={this.swapiService.getAllStarships}
            renderItem={(item) => item.name}
          />
        </div>
        <div className="col-md-6">
          <PersonDetails personId={this.state.selectedPerson} />
        </div>
      </div>
```

Теперь надо перейти в компонент item-list и использовать эту рендер-функцию: заменить прошлый блок {name} на результат вызова рендер-функции. Name больше не используется, его не надо деструктуризировать, а вместо person в массиве содержится любой элемент, поэтому он заменятся на item.

item-list.js

```
  renderItems(arr) {
    // было return arr.map((person) => {
    return arr.map((item) => {
      const { id } = item;
      const label = this.props.renderItem(item);

      return (
        <li
          className="list-group-item"
          key={id}
          onClick={() => this.props.onItemSelected(id)}
        >
          {label}
        </li>
      );
    });
  }
```

Теперь для каждого компонента ItemList можно определять, что он будет выводить с помощью рендер-функции. Так можно сделать в app.js и в people-page.js:

```
<ItemList
  onItemSelected={this.onPersonSelected}
  getData={this.swapiService.getAllPeople}
  renderItem={(item) => `${item.name} (${item.gender}, ${item.birthYear})`}/>

сразу деструктуризировать:
  renderItem={({name, gender, birthYear}) => `${name} (${gender}, ${birthYear})`}/>
```

Пока что возвращается только текст для отображения в li. Можно пойти дальше и вернуть JSX-разметку. В примере ниже будет создаваться кнопка:

```
renderItem={(item) => (<span> {item.name} <button>!</button> </span>)}
```

Свойства-элементы (контейнеры)

В качестве свойств в компонент можно передавать другие react-элементы.

Сейчас на странице отображаются 2 элемента рядом: список персонажей и их детализация. Может понадобиться в дальнейшем создавать группы компонентов в таком же формате: список кораблей и их детализация и т.д.
Чем больше копипасты – тем сложнее поддерживать код, потому что придётся делать несколько одинаковых правок сразу в нескольких местах. Для этого разметку (не имеющиеся сейчас компоненты, а именно разметку, в которую они обёрнуты), можно вынести в отдельный компонент. Это именно создание компонента из разметки: кода html и стилей без сложного функционала и логики.

Можно сделать компонент-контейнер, который будет принимать через пропсы другие компоненты и правильно их размещать. Этот контейнер должен уметь работать с любыми двумя компонентами, которые нужно разнести по двум сторонам, а не предзаданными. Соответственно, у него в пропсах будут 2 свойства: left и right.

После этого, если понадобится переиспользовать разметку, не нужно будет копировать html, а достаточно переимпользовать этот компонент.

Кроме того, можно прикрутить логику, чтобы такие элементы могли выбирать, что рендерить в зависимости от условия (загрузка, ошибка и т.д.).

Было:

```
people-page.js

  render() {
    if (this.state.hasError) {
      return <ErrorIndicator />;
    }

    return (
      <div className="row mb2">
        <div className="col-md-6">
          <ItemList
            onItemSelected={this.onPersonSelected}
            getData={this.swapiService.getAllPeople}
            renderItem={({name, gender, birthYear}) => `${name} (${gender}, ${birthYear})`}/>
        </div>
        <div className="col-md-6">
          <PersonDetails personId={this.state.selectedPerson} />
        </div>
      </div>
    );
  }
```

В файле people-page.js для удобства в отдельные константы будут вынесены itemList и personDetails, потому что сейчас эти компоненты разрастаются, в их пропсы передаётся много кода.
Вся окружающая их разметка будет вынесена в отдельный компонент Row.

Стало:

```
row.js
const Row = ({ left, right }) => {
  return (
    <div className="row mb2">
```

```
      <div className="col-md-6">
        {left}
      </div>
      <div className="col-md-6">
        {right}
      </div>
    </div>
  );
};
```

people-page.js

```
render() {
    if (this.state.hasError) {
      return <ErrorIndicator />;
    }

    const itemList = (
      <ItemList
        onItemSelected={this.onPersonSelected}
        getData={this.swapiService.getAllPeople}
        renderItem={({name, gender, birthYear}) => `${name} (${gender}, ${birthYear})`}
    />);

    const personDetails = (
      <PersonDetails personId={this.state.selectedPerson} />
    );

    return (
      <Row left={itemList} right={personDetails} />
    );
  }
```

Для приложения можно создать также другие элементы-контейнеры, чтобы переиспользовать блоки html и css. Например, вместо random-planet можно сделать элемент-контейнер, который будет заниматься не только планетами.

Children + ErrorBoundry

Документация по API Children тут.

Есть 2 способа передавать свойства компонентам (равнозначны):
- передавать пары ключ-значения, похожие на атрибуты в html;
- записать что-то в тело компонента.

```
<Component>
  Hello, {[1, 2, 4]}
</Component>


class Component extends… () {
  const data = this.props.children;
}

const Component = (props) => {
  const data = this.props.children;
}
```

Сейчас в файле people-page.js компонент ItemList получает рендер функцию так:

people-page.js

```
<ItemList
  onItemSelected={this.onPersonSelected}
  getData={this.swapiService.getAllPeople}
  renderItem={({name, gender, birthYear}) => `${name} (${gender}, ${birthYear})`}
/>
```

Поскольку рендер-функция отвечает за то, что будет выведено на экран, её можно передать в теле компонента. Деструктуризации больше нет, чтобы всё было более читаемо.

people-page.js

```
const itemList = (
  <ItemList
    onItemSelected={this.onPersonSelected}
    getData={this.swapiService.getAllPeople}
  >
    {(i) => (
      `${i.name}, (${i.birthYear})`
    )}

  </ItemList>
);
```

Теперь надо сделать так, чтобы сам компонент ItemList мог получить доступ к содержимому тела. Доступ к такой информации можно получить через this.props.children.
.children работает только в том случае, если из компонента App удалить разметку, которая «не компонент» и создаёт другие пары: корабли и планеты. См. ответ здесь.

Было:

```
renderItems(arr) {
  return arr.map((item) => {
    const { id } = item;
    const label = this.props.renderItem(item);
```

Стало:

```
renderItems(arr) {
  return arr.map((item) => {
    const { id } = item;
    const label = this.props.children(item);
```

ErrorBoundry
С помощью children можно передавать дерево компонентов. Предлагается выделить в отдельный компонент ErrorBoundry. Он будет заниматься выскакивающей ошибкой. Компонент может получить один или несколько элементов в качестве children и он отрендерит их точно в таком виде, в котором получил.

Из компонента people-page переезжают в отдельный компонент ErrorBoundry:
- метод жизненного цикла componentDidCatch
- hasError из стейта.

В компоненте ErrorBoundry поступаем так же, как обычно при обработке ошибок. Этот компонент будет решать, что ему возвращать: компонент-ошибку или props.children.
В такой компонент можно обернуть любой другой компонент и ErrorBoundry автоматом будет отлавливать все ошибки, которые происходят в его children.

В примере ниже в ErrorBoundry можно обернуть компонент Row целиком и/или PersonDetails и удалить из этих компонентов их собственные обработчики ошибок. Компонент-обёртка ErrorBoundry будет отлавливать всё.

```
error-boundry.js
class ErrorBoundry extends Component {
  state = {
    hasError: false
  };

  componentDidCatch() {
    this.setState({
      hasError: true
    });
  };

  render() {
    if (this.state.hasError) {
      return <ErrorIndicator />
    }

    return this.props.children;
  };
};


people-page.js
  render() {

    const itemList = (
      <ItemList
        onItemSelected={this.onPersonSelected}
        getData={this.swapiService.getAllPeople}
      >
        {(i) => (
          `${i.name}, (${i.birthYear})`
        )}

      </ItemList>
    );

    const personDetails = (
      <ErrorBoundry>
        <PersonDetails personId={this.state.selectedPerson} />
      </ErrorBoundry>
    );

    return (
      <ErrorBoundry>
        <Row left={itemList} right={personDetails} />
      </ErrorBoundry>
    );
  }
```

Практика - рефакторинг компонента

Файл person-details.js и все person-переменные переименовываются в item-details.js и item соответственно.

Поскольку item-details должен работать с любыми данными, наружу в пропсы выносятся некоторые функции. В State добавляется поле image, чтобы хранить «универсальное» изображение, которое туда записывается после загрузки данных:

```
app.js
    const { getPerson, getStarship } = this.swapiService;

    const personDetails = ( <ItemDetails
```

```
      itemId={11}
      getData={getPerson}
      getImageUrl={}/> );

      const starshipDetails = ( <ItemDetails
      itemId={5}
      getData={getStarship}
      getImageUrl={}/> );


item-details.js
  state = {
    item: null,
    loading: false,
    image: null
  }

  getData(itemId)
    .then((item) => {
      this.setState({ item, loading: false, image: getImageUrl(item) });
    });
```

В Swapi-Service добавляются функции для получения изображений персонажей, кораблей и планет. Эти функции будут передаваться в item-details как пропсы для получения того или иного изображения:

```
swapi-service.js
  _imageBase = 'https://starwars-visualguide.com/assets/img';

  getPersonImage = ({id}) => {
    return `${this._imageBase}/characters/${id}.jpg`
  };

  getStarshipImage = ({id}) => {
    return `${this._imageBase}/starships/${id}.jpg`
  };

  getPlanetImage = ({id}) => {
    return `${this._imageBase}/planets/${id}.jpg`
  };
```

Работа с props.children

Сейчас компонент item-details хорошо работает только с персонажами, потому что у кораблей нет полей «возраст», «цвет глаз» и т.д., а Gener, Birth Year и Eye Color захардкожены. Эти значения надо вынести наружу и сделать так, чтобы компонент можно было конфигурировать.

Решить задачу можно несколькими способами.
Например, передавать в пропсы объект с конфигурацией типа fields = {[{field: 'gender', label:' Gender'}]}. Такой код – не в духе React (Юра говорит), потому что не соблюдается принцип компонентности.

Надо знать наименование полей объекта и сам объект с данными. Это можно реализовать с функцией Record, которая будет создавать li-элементы. Её параметры: item – сам элемент, field – поле, которое надо достать из объекта и label – запись на ui.

Было (захардкожено и заточено под конкретные данные):

```
<ul className="list-group list-group-flush">

  <li className="list-group-item">
    <span className="term">Gender</span>
    <span>{gender}</span>
  </li>
```

```
  <li className="list-group-item">
    <span className="term">Birth Year</span>
    <span>{birthYear}</span>
  </li>

</ul>
```

Стало (пока добавить в этот файл функцию record):

```
item-details.js

const Record = ({item, field, label}) => {
  return (
    <li className="list-group-item">
    <span className="term">{label}</span>
    <span>{ item[field] }</span>
  </li>
  );
};

export { Record };


app.js
import ItemDetails, { Record } from '../item-details';

    const personDetails = ( <ItemDetails
      itemId={11}
      getData={getPerson}
      getImageUrl={getPersonImage}
      >
        <Record field='gender' label='Gender' />
        <Record field='eyeColor' label='Eye Color' />
      </ItemDetails> );
```

Это не всё «стало», продолжение ниже.

API Children

Чтобы Record заработал, ему надо знать item, из которого он будет доставать все эти данные. Т.е. сейчас можно достать детей (функции Record) и вывести на экран их содержимое через { this.props.children }, но нужно как-то обратиться к родительскому объекту, чтобы получить доступ к его полям. Так, например, item и item[field] пока не доступны.

Для работы с детьми есть специальный API Children, документация по нему тут.
Дело в том, что children может быть строкой, элементом, числом, чем угодно. Функция React.children.map сделает так, чтобы не надо было задумываться, какого именно типа child сейчас попался, он пройдётся по каждому child и обработает все специальные случаи типа null или undefined. Можно заменять и обрабатывать child-элементы как угодно, можно возвращать вместо них null и тем самым скрывать, можно оборачивать… не обязательно возвращать их в том виде, как они поступают.

Кратко: React.Children.map() позволяет проитерироваться по this.props.children и сделать что-нибудь с каждым child и вернуть что угодно. Есть одно правило: react-элементы нельзя изменять после того, как они были созданы.

В данном случае, в каждый child надо передать item.

```
item.details, class ItemDetails на том месте, где было захардкожено

<div className="card-body">
  <h4>{name}</h4>
  <ul className="list-group list-group-flush">
    {
      React.Children.map(this.props.children, (child, idx) => {
```

```
        return <li>{idx}</li>;
      })
    }
  </ul>
  <ErrorButton />
</div>
```

Это не весь рефактор, сейчас пока он не работает правильно, потому что нет доступа к item, см. следующий раздел.

Клонирование элементов

React.Children.map() позволяет проитерироваться по this.props.children и сделать что-нибудь с каждым child и вернуть что угодно. Есть одно правило: react-элементы нельзя изменять после того, как они были созданы, с ними надо работать так, будто они неизменяемые. Поэтому нельзя написать внутри map-функции child.item = item.

Вместо этого надо создать новый элемент, в котором есть новое свойство item. Для этого есть ещё один метод: React.cloneElement(). Документация тут.
Копирует и возвращает новый реакт-элемент.
Config должен содержать все новые props, key или ref. Возвращаемый элемент будет иметь все старые + новые props. Новые children заменят существующих children.

```
React.cloneElement(element, [config], [...children])

Почти эквивалентно этому
<element.type {...element.props} {...props}>{children}</element.type>
```

Вот итоговая реализация:
Был создан дополнительный компонент **Record** (запись).
Ему необходимы 3 поля: «field», «label» и «item». Первые 2 передаются через app, а item нельзя сразу передать, потому что эти данные получаются внутри компонента ItemDetails и снаружи их получить никак нельзя. Внешний код ничего не знает о том, как создаётся объект item.

Для того, чтобы всё-таки передать объект item в child-элементы ItemDetails, была осуществлена итерация по каждому child с помощью React.**Children**.map() и внутри каждый child был преобразован с помощью React.**cloneElement**(), с помощью которого элементу был передан готовый к этому времени item.

```
app.js
import ItemDetails, { Record } from '../item-details';
    const personDetails = (
      <ItemDetails
        itemId={11}
        getData={getPerson}
        getImageUrl={getPersonImage}
      >
        <Record field='name' label='Name' />
        <Record field='gender' label='Gender' />
        <Record field='eyeColor' label='Eye Color' />
      </ItemDetails> );



item-details.js
const Record = ({item, field, label}) => {
  return (
    <li className="list-group-item">
    <span className="term">{label}</span>
```

```
    <span>{ item[field] }</span>
  </li>
  );
};


export default class ItemDetails extends Component {
…state = {
    item: null,
    loading: false,
    image: null
  }

  updateItem() {
    …getData(itemId)
      .then((item) => {
        this.setState({ item, loading: false, image: getImageUrl(item) });
      });
  }

  render() {
    const { name, item, loading, image } = this.state;
    …return (
        …<ul className="list-group list-group-flush">
            {
              React.Children.map(this.props.children, (child) => {
                return React.cloneElement(child, {item});
              })
            }
          </ul>
```

Компоненты высшего порядка (HOC)

Работа с itemList (списко персонажей, кораблей и т.д.).
Как у любого другого сетевого компонента, его работа состоит из нескольких фаз:
1. В componentDidMount() делаем запрос, получаем данные, обновляем state.
2. В render() проверяем, если есть данные, то их отображаем. Если данных нет – отображаем спиннер.

Чтобы создать новый сетевой компонент, не обязательно копировать весь этот код (который отправляет запрос, проверяет наличие данных, обновляет state, отображает спиннер, ошибку и т.д.). На самом деле, меняются только 2 аспекта: как делается запрос и как отображается результат на экране.

Чтобы сделать этот код лучше, существует паттерн HOC.

Цель – вынести сетевой код, отображение данных и ошибку в отдельную конструкцию и переиспользовать с другими компонентами.

В JS функция может возвращать другую функцию и/или класс.
Вместо экспорта класса, можно экспортировать функцию (её вызов), которая возвращает класс.

```
// возврат функции
const f = (a) => {
  return (b) => {
    console.log(a + b);
  }
};
f(1)(2);


// возврат класса
const f = () => {
  return ItemList;
```

```
};

export default f();
```

В JS можно создавать безымянные функции и безымянные классы. Таким оразом будет возвращён новый класс, у которого нет имени, но есть содержимое. В примере ниже возвращается анонимный класс, который наследует Component и у него есть функция render():

```
const f = () => {
  return class extends Component {
    render() {
      return <p>Hi</p>
    }
  };
};

export default f();
```

Поскольку внутренний анонимный класс – это компонент, значит в него можно передать componentDidMount() или любую другую функцию жизненного цикла:

```
const f = () => {
  return class extends Component {

    componentDidMount() {
      console.log(this.props);
    }

    render() {
      return <p>Hi</p>
    }
  };
};

export default f();
```

Поскольку возврат функции (а функция возвращает анонимный класс) экспортируется под именем ItemList (переименование происходит в index.js, тут я её называю anonim), то она получает все пропсы (в т.ч. children), которые в неё передаются в App (выведутся в консоль), а именно:

```
<AnonimItemList
  onItemSelected={this.onPersonSelected}
  getData={this.swapiService.getAllPeople}
>

  { ({name}) => <span>{name}</span> }

</ AnonimItemList >
```

Следующий шаг – сделать полноценный компонент-класс <ItemList>, в этом же файле создать функцию высшего порядка, которая будет брать на себя все рассчёты и запросы, а возвратит компонент-класс <ItemList> и «напихает» ему уже полученных пропсов.

В примере ниже создана обёртка, которая не делает ничего. Она получает пропсы, как в примере выше, и возвращает класс <ItemList/> и передаёт ему свои пропсы. Всё будет работать, как и раньше:

```
class ItemList extends Component {
  …объявление класса
```

```
}

const f = () => {
  return class extends Component {

    render() {
      return <ItemList {...this.props} />;
    }
  };
};
export default f();


app.js
const list = <ItemList
  onItemSelected={this.onPersonSelected}
  getData={this.swapiService.getAllPeople}
>
  { ({name}) => <span>{name}</span> }
</ItemList>
```

HOC

Финальный шаг – вынести в обёртку всю логику из оригинального компонента:
- работа с сетью
- спиннер
- ошибка
- что отображать по клику

State, componentDidMount, логика спиннера и данные пеерезжают из компонента в обёртку. Переменная itemList переименовывается на data.

```
item-list.js

// компонент, который надо вернуть
class ItemList extends Component {
  renderItems(arr) {
    return arr.map((item) => {
      const { id } = item;
      const label = this.props.children(item);
      return (
        <li
          className="list-group-item"
          key={id}
          onClick={() => this.props.onItemSelected(id)}
        >
          {label}
        </li>
      );
    });
  }

  render() {
    const { data } = this.props;
    const items = this.renderItems(data);

    return (
      <ul className="item-list list-group">
        {items}
      </ul>
    );
  }
}


// возврат класса
```

```
const f = () => {
  return class extends Component {

    state = { data: null };

    componentDidMount() {
      const { getData } = this.props;

      getData()
        .then((data) => {
          this.setState({
            data
          });
      });
    }

    render() {
      const { data } = this.state;
      if (!data) return <Spinner />;

      return <ItemList {...this.props} data={data} />;
    };
  };
};

export default f();
```

app.js

```
import ItemList from '../item-list';

<ItemList
  onItemSelected={this.onPersonSelected}
  getData={this.swapiService.getAllPeople}
>
  { ({name}) => <span>{name}</span> }
</ItemList>
```

Единственная зависимость, которая осталась – компонент знает, что работает с ItemList и заточен под него.
В функцию f можно передать аргумент, который будет являться совершенно любым компонентом вместо ItemList.
F переименована в withData чтобы было понятно, что она делает.

```
const withData = (View) => {
  return class extends Component {

    state = { data: null };

    componentDidMount() {
      const { getData } = this.props;

      getData()
        .then((data) => {
          this.setState({
            data
          });
      });
    }

    render() {

      const { data } = this.state;
      if (!data) return <Spinner />;
```

```
      return <View {...this.props} data={data} />;
    };
  };
};

export default withData(ItemList);
```

Таким образом, ItemList разделён на 2 части. Одна отвечает только за отрисовку, а вторая – за логику работы с сетью. Функция нужна для того, чтобы можно было выбирать (передавать в её параметр), какой именно компонент будет заниматься отображением данных.

Следующий шаг: поскольку itemList не содержит state, из него можно сделать компонент-функцию.
Обёртка больше не получает GetData в пропсах, теперь это делается явно при экспорте.
Поскольку логика и отрисовка больше не связаны друг с другом, можно раскидать их по разным файлам.

Итоговая реализация (ErrorIndicator пока не прикручен):
item-list.js

```
import React from 'react';
import SwapiService from '../../services/swapi-service';
import { withData } from '../hoc-helpers/';
import './item-list.css';


const ItemList = (props) => {

  const { data, onItemSelected, children: renderLabel } = props;

  const items = data.map((item) => {
    const {id} = item;
    const label = renderLabel(item);
    return (
      <li
        className="list-group-item"
        key={id}
        onClick={() => onItemSelected(id)}
      >
        {label}
      </li>
    );
  });

  return (
    <ul className="item-list list-group">
      {items}
    </ul>
  );
};


const { getAllPeople } = new SwapiService();

export default withData(ItemList, getAllPeople);
```

with-data.js

```
import React, { Component } from 'react';
import Spinner from '../spinner';
import ErrorIndicator from '../error-indicator';

const withData = (View, getData) => {
```

```
  return class extends Component {

    state = { data: null };

    componentDidMount() {
      getData()
        .then((data) => {
          this.setState({ data });
      });
    }

    render() {
      const { data } = this.state;

      if (!data) return <Spinner />;

      return <View {...this.props} data={data} />;
    };
  };
};


export default withData;
```

app.js

```
<ItemList
  onItemSelected={this.onPersonSelected}
  getData={this.swapiService.getAllPeople}
>
  { ({name}) => <span>{name}</span> }
</ItemList>
```

Рефакторинг, вынос больших компонентов

Если компонент гибкий с т.з. настройки, то он начинет разрастаться в объёме и его сложно читать в общем потоке кода: очень много конфигурационных параметров. Хорошее решение – вынести такие компоненты с раздутой конфигурацией в отдельные файлы и затем импортировать их.

В папке sw-components хранятся новые компоненты: item-lists.js и details.js

Последние 2 строчки в ItemList удалить, экспортировать только сам ItemList

```
item-list.js
Было
const { getAllPeople } = new SwapiService();
export default withData(ItemList, getAllPeople);

Стало
export default ItemList;
```

item-lists.js

```
import React from 'react';
import ItemList from '../item-list';
import { withData } from '../hoc-helpers';
import SwapiService from '../../services/swapi-service';

const swapiService = new SwapiService();
const { getAllPeople, getAllStarships, getAllPlanets } = swapiService;
```

```
const PersonList = withData(ItemList, getAllPeople);
const PlanetList = withData(ItemList, getAllPlanets);
const StarshipList = withData(ItemList, getAllStarships);

export { PersonList, PlanetList, StarshipList };
```

details.js (не используется HOC)

```
import Rect from 'react';
import ItemDetails, { Record } from '../item-details';
import SwapiService from '../../services/swapi-service';

const swapiService = new SwapiService();
const {
  getPerson, getStarship, getPlanet,
  getPersonImage, getStarshipImage, getPlanetImage
} = swapiService;

const PersonDetails = ({ itemId }) => {
  return (
    <ItemDetails itemId={itemId} getData={getPerson} getImageUrl={getPersonImage}>
      <Record field='gender' label='Gender' />
      <Record field='eyeColor' label='Eye Color' />
    </ItemDetails>
  );
};

const PlanetDetails = ({ itemId }) => {
  return (
    <ItemDetails itemId={itemId} getData={getPlanet} getImageUrl={getPlanetImage}>
      <Record field='popelation' label='Population' />
      <Record field='diameter' label='Diameter' />
    </ItemDetails>
  );
};

const StarshipDetails = ({ itemId }) => {
  return (
    <ItemDetails itemId={itemId} getData={getStarship} getImageUrl={getStarshipImage}>
      <Record field='model' label='Model' />
      <Record field='length' label='Length' />
    </ItemDetails>
  );
};

export { PersonDetails, PlanetDetails, StarshipDetails };
```

Использование в App.js

```
    return (
    <div className='app'>
      <Header />

      <PersonDetails   itemId={11} />
      <PlanetDetails   itemId={5} />
      <StarshipDetails itemId={9} />

      <PersonList>
        { ({name}) => <span>{name}</span> }</PersonList>

      <StarshipList>
        { ({name}) => <span>{name}</span> }</StarshipList>

      <PlanetList>
```

```
        { ({name}) => <span>{name}</span> }</PlanetList>
    </div>
```

Композиция компонентов высшего порядка

Композиция – применение одной функции к результату другой: f(g(x)).
Композиция HOC – это использование одних HOC внутри других HOC.

Сейчас в компоненты-листы всё ещё надо явно передавать children, в котором хранится рендер-функция. Чтобы от этого избавиться, можно использовать ещё одну функцию высшего порядка.

Функция withChildFunction умеет брать любой компонент и вставлять ему что-то в качестве children.
Wrapped – это компонент, который она будет оборачивать.
fn – функция, которую надо передать в качестве props.children во Wrapped-компонент.

Рендер-функции тоже могут быть разными.
«renderName» и «renderModelAndName» немного отличаются и выводят на экран разный текст.

```
import React from 'react';
import ItemList from '../item-list';
import { withData } from '../hoc-helpers';
import SwapiService from '../../services/swapi-service';

const swapiService = new SwapiService();
const { getAllPeople, getAllStarships, getAllPlanets } = swapiService;


// обернёт компонент и вставит ему детей
const withChildFunction = (Wrapped, fn) => {
  return (props) => {
    return (
      <Wrapped {...props}>
        {fn}
      </Wrapped>
    );
  }
};

// рендер-функции для персонажей и кораблей пойдут в children
const renderName = ({ name }) => <span>{name}</span>;
const renderModelAndName = ({ model, name }) => <span>{name} ({model})</span>

// композиция функций высшего порядка
const PersonList = withData(
  withChildFunction(ItemList, renderName),
  getAllPeople);

const PlanetList = withData(
  withChildFunction(ItemList, renderName),
  getAllPlanets);

const StarshipList = withData(
  withChildFunction(ItemList, renderModelAndName),
  getAllStarships);

export { PersonList, PlanetList, StarshipList };
```

Использование в app.js

```
<PersonList />
```

```
<StarshipList />
<PlanetList />
```

Теперь детали конфигурации скрыты. Использовать такие компоненты в разы легче.

Контекст API

Документация тут.

Context API решает проблему проброса свойств (property drill) от компонентов верхнего уровня до компонентов нижнего.
Контекст позволяет создать специальное хранилище данных и эти данные будут доступны для всех дочерних компонентов без необходимости явно передавать эти свойства.
Гугли dependency injection.

На практике вызывается функция, которая возвращает объект, в котором хранятся создаются 2 компонента: Provider и Consumer.

Кратко:

```
import React from 'react';

1. создать контекст через функцию (можно сразу деструктуризировать)

const { Provider, Consumer } = React.createContext()


2. Обернуть в Provider компоненты-потребители

<Provider value={this.swapiService}>
  <div className='app'>
    <PersonList />
    <StarshipList />
    <PlanetList />
  </div>
</Provider>


3. Через Consumer вытащить переданное значение

const PersonDetails = () => {
  return (
    <Consumer>
      {
        (swapiService) => {
          return (
            <ItemDetails getData={swapiService.getPerson} />
          );
        }
      }
    </Consumer>
  );
};
```

createContext

```
import React from 'react';

const myContext = React.createContext(defaultValue)

const { Provider, Consumer } = React.createContext()
```

```
const { Provider : SwapiProvider, Consumer : SwapiConsumer } = React.createContext()
```

Функция возвращает объект контекста.
Этот объект можно сразу деструктуризировать на константы-компоненты Provider и Consumer.
Provider – компонент, который хранит передаваемое значение
Consumer – компонент, который будет потреблять это значение.

Аргумент default будет использован только в том случае, если у компонента нет соответствующего Provider в иерархии над ним. Если Consumer не сможет найти никакого Provider, он будет использовать значение default.

Provider
Передаёт значение из своего пропса компонентам-потребителям.

```
<MyContext.Provider value={someValue}>
```

Consumer
Считывает значение из Provider. Внутри содержит функцию, которая принимает на вход свойство-пропс из Provider, а возвращает любые компоненты, которые используют это переданное из provider свойство.

```
<ChildElement>
  <Consumer>
    {
      (lang) => {
        return (
          <Chat lang={lang} />
        )
      }
    }
  </Consumer>
</ChildElement>
```

Использование Context API

С помощью контекста элементы смогут получать сервис, а не создавать каждый для себя его инстансы.
Рефакториться будут details-элементы.

Создаётся новая директория:

```
components
  swapi-service-context
    index.js
    swapi-service-context.js
```

Весь код – это создать Provider и Consumer и экспортировать их.
При создании они сразу переименовываются в swapi-поставщик и swapi-потребитель.

```
swapi-service-context.js

import React from 'react';

const {
  Provider : SwapiServiceProvider,
  Consumer: SwapiServiceConsumer
} = React.createContext();
```

```
export { SwapiServiceProvider, SwapiServiceConsumer };
```

Теперь можно пойти в компонент самого высшего уровня и использовать контекст, чтобы у всех компонентов был доступ к тому значению, которое передаём:

```
app.js

import { SwapiServiceProvider } from '../swapi-service-context/swapi-service-context';

return (
  <ErrorBoundry>
    <SwapiServiceProvider value={this.swapiService}>

      <div className='app'>
        <Header />
        <PersonDetails   itemId={11} />
        <PlanetDetails   itemId={5} />
        <StarshipDetails itemId={9} />
        <PersonList />
        <StarshipList />
        <PlanetList />
      </div>

    </SwapiServiceProvider>
  </ErrorBoundry>
)};
```

Теперь в details можно получить это значение.
Импорт потребителя, обернуть в него элементы, которые ждут передаваемое значение.
Важный нюанс – потребитель передаёт значение как аргумент функции. Следовательно, он должен вернуть все оборачиваемые компоненты.

Аргумент swapiService – это именно то значение, которое было передано в Provider по иерархии выше.

```
.details.js.

const PersonDetails = ({ itemId }) => {
  return (
    <SwapiServiceConsumer>
      {
        (swapiService) => {
          return (
            <ItemDetails
              itemId={itemId}
              getData={swapiService.getPerson}
              getImageUrl={swapiService.getPersonImage}
            >
            <Record field='gender' label='Gender' />
            <Record field='eyeColor' label='Eye Color' />
          </ItemDetails>
          );
        }
      }
    </SwapiServiceConsumer>
  );
};
```

Можно сразу же деструктуризировать:

```
    <SwapiServiceConsumer>
      {
```

```
        // (swapiService) => {
        ({getPerson, getPersonImage}) => {
          return (
            <ItemDetails
              itemId={itemId}
              getData={getPerson}
              getImageUrl={getPersonImage}
            >
            <Record field='gender' label='Gender' />
            <Record field='eyeColor' label='Eye Color' />
          </ItemDetails>
          );
        }
      }
    </SwapiServiceConsumer>
```

То же самое проделать с кораблями и планетами и swapiService можно выпиливать из details.js.

Теперь для того, чтобы изменить данные, которые запрашиваются по сети, на локальные тестовые данные, достаточно сделать так:

```
app.js

import DummySwapiService from '../../services/dummy-swapi-service';
swapiService = new DummySwapiService();
```

Таким же макаром можно подменять данные, если сервис будет выдавать ошибку или долго грузиться.
Сейчас этот код выглядит громоздко, но его можно улучшить с HOC.

Использование HOC для работы с контекстом

Details.js в каталоге sw-components надо разбить на 3 отдельных файла: PersonDetails, PlanetDetails, StarshipDetails.

Задача получения контекста должна быть вынесена в компонент высшего порядка.
В каталоге hoc-components создаётся компонент WithSwapiService. Вместо всего кода, который повторялся, можно использовать эту HOC-функцию.

```
with-swapi-service.js

import React from 'react';
import { SwapiServiceConsumer } from '../swapi-service-context/';

const withSwapiService = (Wrapped) => {

  return (props) => {
    return (
      <SwapiServiceConsumer>
        {
          (swapiService) => {
            return (
              <Wrapped {...props} swapiService={swapiService} />
            )
          }
        }
      </SwapiServiceConsumer>
    );
  }
};

export default withSwapiService;
```

В файле person-details.js можно удалить всё, что работает с контекстом, а экспорт обернуть в withSwapiService. Также, надо вытащить переданный HOC-ом в пропсы swapiService.

person-details.js

```
import React from 'react';
import ItemDetails, { Record } from '../item-details';
import { withSwapiService } from '../hoc-helpers';


const PersonDetails = ({ itemId, swapiService }) => {
  const { getPerson, getPersonImage } = swapiService;

  return (
    <ItemDetails
      itemId={itemId}
      getData={getPerson}
      getImageUrl={getPersonImage}
    >
      <Record field='gender' label='Gender' />
      <Record field='eyeColor' label='Eye Color' />
  </ItemDetails>
  );

};

export default withSwapiService(PersonDetails);
```

Таким образом, HOC взял на себя функцию «выплёвывания» wrapped-компонента и передачи ему в пропсы swapiService.

Трансформация props в компонентах HOC

Вместо того, чтобы передавать весь swapiService в PersonDetails, можно туда передать исключительно те методы, которые нужны этому компоненту: getPerson и getPersonImage.
Ещё лучше – если передать getPerson под именем getData, а getPersonImage – как getImageUrl, потому что именно с такими именами работают details-компоненты.

Правила этой передачи проще всего описать функцией map.
Эта функция должна использоваться в withSwapiService в качестве второго параметра для вычленения нужных данных.

Person-details.js

```
import React from 'react';
import ItemDetails, { Record } from '../item-details';
import { withSwapiService } from '../hoc-helpers';


// было
const PersonDetails = ({ itemId, swapiService }) => {
  const { getPerson, getPersonImage } = swapiService;

  return (
    <ItemDetails
      itemId={itemId}
      getData={getPerson}
      getImageUrl={getPersonImage}
    >
      <Record field='gender' label='Gender' />
      <Record field='eyeColor' label='Eye Color' />
  </ItemDetails>
  );
```

```
};

const PersonDetails = ({ itemId, getData, getImageUrl }) => {

  return (
    <ItemDetails
      itemId={itemId}
      getData={getData}
      getImageUrl={getImageUrl}
    >
      <Record field='gender' label='Gender' />
      <Record field='eyeColor' label='Eye Color' />
  </ItemDetails>
  );

};


const mapMethodsToProps = (swapiService) => {
  return {
    getData: swapiService.getPerson,
    getImageUrl: swapiService.getPersonImage
  }
};

export default withSwapiService(PersonDetails, mapMethodsToProps);
```

Поскольку передаваемые пропсы теперь полностью соответствуют наименованию требуемых свойств, можно сократить их так:

```
Person-details.js

import React from 'react';
import ItemDetails, { Record } from '../item-details';
import { withSwapiService } from '../hoc-helpers';

const PersonDetails = (props) => {

  return (
    <ItemDetails ...props>
      <Record field='gender' label='Gender' />
      <Record field='eyeColor' label='Eye Color' />
  </ItemDetails>
  );

};


const mapMethodsToProps = (swapiService) => {
  return {
    getData: swapiService.getPerson,
    getImageUrl: swapiService.getPersonImage
  }
};

export default withSwapiService(PersonDetails, mapMethodsToProps);
```

Переделка with-swapi-service.js под функцию map:

```
with-swapi-service.js

import React from 'react';
import { SwapiServiceConsumer } from '../swapi-service-context/';
```

```
const withSwapiService = (Wrapped, mapMethodsToProps) => {

  return (props) => {
    return (
      <SwapiServiceConsumer>
        {
          (swapiService) => {
            const serviceProps = mapMethodsToProps(swapiService);

            return (
              <Wrapped {...props} {...serviceProps} />
            );
          }
        }
      </SwapiServiceConsumer>
    );
  }
};

export default withSwapiService;
```

Теперь надо обновить list-компоненты. Сейчас они используют with-data.
WithData вторым аргументом получал getData как внешний аргумент. Но withSwapiService уже умеет передавать нужные функции для запроса данных, поэтому getData больше не надо передавать в явном виде.
Вместо этого можно взять getData из props.

Было:

with-data.js

```
import React, { Component } from 'react';
import Spinner from '../spinner';

const withData = (View, getData) => {
  return class extends Component {

    state = { data: null };

    componentDidMount() {
      getData()
        .then((data) => {
          this.setState({
            data
          });
      });
    }

    render() {
      const { data } = this.state;

      if (!data) return <Spinner />;

      return <View {...this.props} data={data} />;
    };
  };
};


export default withData;
```

Стало:

with-data.js

```
import React, { Component } from 'react';
import Spinner from '../spinner';

const withData = (View) => {
  return class extends Component {

    state = { data: null };

    componentDidMount() {
      this.props.getData()
        .then((data) => {
          this.setState({
            data
          });
      });
    }

    render() {
      const { data } = this.state;

      if (!data) return <Spinner />;

      return <View {...this.props} data={data} />;
    };
  };
};


export default withData;
```

Обновление списков

Было

item-lists.js

```
import React from 'react';
import ItemList from '../item-list';
import SwapiService from '../../services/';

// удаляется полностью
// const swapiService = new SwapiService();
// const { getAllPeople, getAllStarships, getAllPlanets } = swapiService;


const withChildFunction = (Wrapped, fn) => {
  return (props) => {
    return (
      <Wrapped {...props}>
        {fn}
      </Wrapped>
    );
  }
};

const renderName = ({ name }) => <span>{name}</span>;
const renderModelAndName = ({ model, name }) => <span>{name} ({model})</span>

const PersonList = withData(
  withChildFunction(ItemList, renderName),
  // getAllPeople); в явном виде не надо

const PlanetList = withData(
  withChildFunction(ItemList, renderName),
  // getAllPlanets); в явном виде не надо
```

```
const StarshipList = withData(
  withChildFunction(ItemList, renderModelAndName),
  // getAllStarships); в явном виде не надо

export { PersonList, PlanetList, StarshipList };
```

Стало:

```
item-lists.js

import React from 'react';
import ItemList from '../item-list';
import { withData, withSwapiService } from '../hoc-helpers';


// обернёт компонент и вставит ему детей
const withChildFunction = (Wrapped, fn) => {
  return (props) => {
    return (
      <Wrapped {...props}>
        {fn}
      </Wrapped>
    );
  }
};

// рендер-функции для персонажей и кораблей пойдут в children
const renderName = ({ name }) => <span>{name}</span>;
const renderModelAndName = ({ model, name }) => <span>{name} ({model})</span>

const mapPersonMethodsToProps = (swapiService) => {
  return { getData: swapiService.getAllPeople }
};

const mapPlanetMethodsToProps = (swapiService) => {
  return { getData: swapiService.getAllPlanets}
};

const mapStarshipMethodsToProps = (swapiService) => {
  return { getData: swapiService.getAllStarships }
};

// композиция функций высшего порядка
const PersonList = withSwapiService(
    withData(
      withChildFunction(ItemList, renderName)),
    mapPersonMethodsToProps);

const PlanetList = withSwapiService(
    withData(
      withChildFunction(ItemList, renderName)),
    mapPlanetMethodsToProps);

const StarshipList = withSwapiService(
    withData(
      withChildFunction(ItemList, renderModelAndName)),
    mapStarshipMethodsToProps);


export { PersonList, PlanetList, StarshipList };
```

Обновление контекста

Контекст не обязательно должен быть статичным, его можно обновлять «на лету». Он работает так же, как любые другие компоненты: если value обновилось, то компоненты ниже по иерархии получат обновлённое значение.
Это может пригодиться, если приложение поддерживает смену языков, темы визуального оформления и т.д.

Важно, что компоненты должны уметь правильно реагировать на изменение контекста. Обычно компоненты реагируют на изменения элементов, которые рендерятся на странице (например, новое значение строки). Надо прописать соответствующие условия в их componentDidUpdate, чтобы компоненты сравнивали предыдущие и новые функции получения (getData) данных из props. Ниже про это будет написано.

Сейчас через контекст передаётся swapiService. Если залезть в App.js, то в нём можно переписать swapi на dummySwapi. Такое переключение можно реализовать через кнопку на странице, которая будет обновлять контекст.

1. Объявить onServiceChange в app.js и передать его через пропсы в header.
2. В header.js добавить кнопку, повесить на неё onServiceChange.

Как реализовать смену значения в сервисе?
Сейчас сервис – это просто поле класса и не находится в state. Если его обновить, то React не узнает, что приложение надо перерисовать. В этой связи, swapiService надо перенести в state и обновить код, который его использует.

onServiceChange
При переключении надо знать предыдущее значение state чтобы знать, на что переключиться. Поэтому в setState передаётся функция.

```
Header.js
const Header = ({ onServiceChange }) => {
  return (
    <div className="header d-flex">
      ...
      <button onClick={onServiceChange} > Change Service </button>


App.js
state = {
  showRandomPlanet: true,
  swapiService: new SwapiService()
};

onServiceChange = () => {
  this.setState(({ swapiService }) => {

    const Service = swapiService instanceof SwapiService ? DummySwapiService : SwapiService;
    return { swapiService: new Service() };
  });
}


<SwapiServiceProvider value={this.state.swapiService}>

<Header onServiceChange={this.onServiceChange} />
```

Если всё оставить в таком виде, то service действительно будет меняться, но компоненты не будут обновляться. Причина – компоненты обновляются только тогда, когда меняется их id. Поскольку нужно сохранить старый id, но вынудить компонент обновиться, надо дописать им в update это:

```
Item-details.js
```

```
componentDidUpdate(prevProps) {
  if (this.props.itemId !== prevProps.itemId ||
      this.props.getData !== prevProps.getData ||
      this.props.getImageUrl !== prevProps.getImageUrl) {
    this.updateItem();
  }
}
```

Код сверху будет обновлять компонент, если новая функция получения данных getData или новая функция получения изображения getImageUrl не соответствуют прежним.

Компонент высшего порядка в With-data.js сейчас работает только с методом componentDidMount. Надо дописать обновления. Весь код из didMount выносится в функцию update, которая вызывается в двух методах жизненного цикла:

```
With-data.js

update() {
  this.props.getData()
    .then((data) => {
      this.setState({ data })
    });
}


componentDidMount() {
  this.update();
}

componentDidUpdate(prevProps) {
  if (this.props.getData !== prevProps.getData) {
    this.update();
  }
}
```

Рефакторинг HOC-компонентов.

Упаковка <Row />, -list и -details компонентов в обобщённые -page компоненты.

Лишние импорты и поля класса в app.js удаляются.
Директория People-page удаляется, т.к. нингде не используется.
Создаётся новая директория pages.

Ранее в App использовались Row, StarshipDetails, StarshipsList и т.д., но теперь их заменяют соответствующие компоненты pages:

```
People-page.js

import React, { Component } from 'react';
import { PersonDetails, PersonList } from '../sw-components';
import Row from '../row';


export default class PeoplePage extends Component {

  state = {
    selectedItem: null
  };

  onItemSelected = (selectedItem) => {
```

```
    this.setState({ selectedItem });
  };

  render() {
    const { selectedItem } = this.state;

    return (
      <Row
        left={<PersonList onItemSelected={this.onItemSelected} />}
        right={<PersonDetails itemId={selectedItem} />}
      />
    );
  };

};


App.js
Было
<Row
  left={<StarshipList />}
  right={<StarshipDetails itemId={9} />}
/>

Стало
<PeoplePage />
<StarshipsPage />
<PlanetsPage />
```

+ правки в with-data.js

Функция compose()

В этом уроке автор что-то намудрил. Если следовать его указаниям, то приложение ломается. Чтобы всё работало, надо переписать много компонентов, которые он оставил за кадром.
Решения использовать функцию compose() особой погоды не делает, поэтому просто запишу принцип работы, а всоё приложение переписывать не буду.

Сейчас файл с HOC-компонентами item-lists.js выглядит запутанно и непонятно. Чтобы избавиться от цепочки вложенных функций, можно создать одну функцию compose(), которая будет принимать массив функций, которые должны поочерёдно передавать результат друг другу, и также компонент, который надо пропустить через эти функции.

Функцию compose можно было бы организовать так:
1. передать несколько других функций, композицию которых надо получить;
2. передать компонент, который через эти функции должен пройти.

```
compose(
  withSwapiService(mapMethodsToProps),
  withData,
  withChildFunction(renderModelAndName)
)(itemList);


В общих чертах выглядеть будет так
const compose = (...functions) => (component) => {...}


Эти функции будут в итоге работать одинаково
compose(a, b, c)(value) == a(b(c(value)));
```

Этот код читать легче (нет): видно, во что компонент оборачивается.

Можно пройтись по массиву функций справа налево, вызвать каждую, передать её результать в следующую функцию. Но это не лучшее решение (Дима так говорит в своём курсе).

В js есть функция reduceRight (именно о ней сразу и подумал).
Для этой функции создаётся отдельный файл в HOC-helpers

```
compose.js

const compose = (...functions) => (component) => {
  return functions.reduceRight(
    (prevResult, foo) => foo(prevResult), component);
};

export default compose;
```

Функция withChildFunction тоже переезжает из item-lists.js в директорию HOC-helpers и немного изменяется:

```
with-child-function.js

было
const withChildFunction = (Wrapped, fn) => {
  return (props) => {
    return (
      <Wrapped {...props}>
        {fn}
      </Wrapped>
    );
  }
};

стало:
const withChildFunction = (fn) => (Wrapped) => {
  return (props) => {
    return (
      <Wrapped {...props}>
        {fn}
      </Wrapped>
    )
  };
};

export default withChildFunction;
```

defaultProps

Работает так же, как значения по умолчанию параметров функции, только устанавливает значения по умолчанию для свойств (пропсов) компонентов.

defaultProps можно записать как свойство созданного компонента. Эта запись происходит вне тела определения. Так надо записывать defaultProps для компонентов-функций:

```
// как свойство компонента
ComponentName.defaultProps = {
  onItemSelected: () => {}
};
```

Для компонентов-классов defaultProps можно записывать как статическое свойство всего класса:

```
class MyComponent extends Component {

  static defaultProps {
    props: value
  }

}
```

Теперь в случае, если в пропсы компонентов не передано значение или это значение – undefined, будет использовано дефолтное значение. В этом свойстве-объекте можно перечислить сколько угодно свойств, которые копонент потребляет из пропсов. Важно, что значение null будет обрабатываться нормально и defaultProps использоваться не будет.

Например, **itemList** рассчитывает получить функцию-обработчик onItemSelected из people-page.js.
Может возникнуть ситуация, когда надо будет просто отобразить список чего-то и никак не реагировать на клики. Один из вариантов (не лучший) – прописать пустую функцию в деструктуризации. Если ничего не передать в props, то в обработчик присвоится пустая функция:

```
const { onItemSelected = () => {} } = props;
```

Намного проще будет установить значение по умолчанию. Теперь если функция onItemSelected не будет передана вышестоящим компонентом, то будет использоваться значение по умолчанию.

```
const ItemList = (props) => {
  // определение компонента
};

ItemList.defaultProps = {
  onItemSelected: () => {}
};
```

Ещё пример – компонент **random-planet**.js
Интарвал обновления сейчас прописывается в самом компоненте – в функции-таймере. Можно сделать так, чтобы время обновления передавалось через пропсы, заодно установить defaultProps:

```
random-planet.js

static defaultProps = {
  updateInterval: 10000
}

componentDidMount() {
  const { updateInterval } = this.props;
  this.updatePlanet();
  this.interval = setInterval(() => this.updatePlanet(), updateInterval);
}
```

propTypes

Поскольку JS – это язык с динамической типизацией, переменные могут менять тип хранимых данных во время жизненного цикла, а отлов ошибок, связанных с типами, откладывается до момента запуска приложения.

TS добавляет строгую типизацию, но это «громоздкая» надстройка, не очень удобно использовать.
В React есть способ указать типы для свойств компонентов. Свойства начнут проверяться перед тем, как компонент их получит и начнёт работать. Проверка propTypes срабатывает после defaultProps.

```
random-planet.js
```

```
static propTypes = {

  propName: (props, propName, componentName) => {
    const value = props[propName];

      if (typeof value === 'number' && !isNaN(value)) {
        return null;
      }

      return new TypeError(`${componentName}: ${propName} must be number`);
    }
  }
}

props           это весь объект свойств (пропсов) компонента.
propName        имя того свойства, для которого сейчас проводится валидация.
componentName   название компонента, для которого проводится валидация
```

Если все проверки прошли успешно, надо вернуть null.
Если нет – возвращаем (не выбрасываем) ошибку.

prop-types библиотека

Чтобы не писать код проверки вручную, как в примере выше, есть несколько библиотек, которые реализуют правило валидации.
Документация здесь.

Библиотека с большими возможностями есть у Air bnb, называется тоже prop-types и лежит на GitHub. В ней множество разных валидаций, более сложная логика.

Установка:

```
npm install prop-types
```

Использование:

```
import PropTypes from 'prop-types';
import propTypes from 'prop-types';  // с функциональными компонентами


static propTypes = {
  propName: PropTypes.number
}


// если свойство необходимо передать явно
static propTypes = {
  propName: PropTypes.number.isRequired
}
```

Поскольку defaultProps срабатывает раньше, то .isRequired нужно указывать по больше части для разработчика (самодокументирование и всё такое).

Кроме простых пропсов с примитивными типами, можно описывать функции или массивы.
Например, в компоненте ItemList:

```
item-list.js

ItemList.propTypes = {
```

```
  onItemSelected: propTypes.func,
  data:           propTypes.arrayOf(propTypes.object).isRequired,  // какого типа массив
  children:       propTypes.func.isRequired
};


row.js

Row.propTypes = {
  left: propTypes.element,  // только react-элемент
  right: propTypes.node     // более универсальный, всё что можно отрендерить
};
```

Можно описать объект с определённой структурой.
Если есть компонент, который должен получать в качестве пропса объект, у которого есть поля user и role:

```
MyComponent.propTypes = {

  user: PropTypes.shape({
    name: PropTypes.string,
    role: PropTypes.oneOf([ 'user', 'admin' ])  // одно из значений
  })

}
```

Router

Роутинг – это переключение между «виртуальными страницами» UI приложения.
Гит роутера здесь.

В обычных приложениях переход между страницами влечёт их полную перезагрузку. В SPA, при переходе по внутренним ссылкам, переход на новые страницы не происходит: страница остаётся прежней, но компоненты этой строницы отображаются выборочно в зависимости от URL.

Такой подход называется Роутинг. Роутинг в контексте SPA – это функционал, который позволяет «переключать» страницы. Роутер – компонент, который, зная id страницы, отображает соответствующий компонент на экране.

Таким образом, в контексте роутинга понятие страниц и переходов – это «виртуальное» понятие. На самом деле, происходит скрытие одних компонентов и показ других в рамках одной и той же страницы браузера.

Библиотека React Router делает 2 вещи:
- читает url и выбирает, какие компоненты нужно отобразить пользователю;
- обновляет url, когда пользователь переходит на новую страницу.

Установка

```
npm install react-router-dom
версия для браузера

import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
```

В тег BrowserRouter оборачивается всё, что возвращает компонент App.
В тег Routes оборачиваются Route со страницами

```
const App = () => {
  return (
```

```
    <BrowserRouter>
      <div className='app'>
        <Header/>

        <Routes>
          <Route path="/" element={<Main />} />
          <Route path="/first" element={<PageOne />} />
          <Route path='/second' element={<PageOne />} />
        </Routes>
      </div>
    </BrowserRouter>
  );
};
```

Link
Компонент Link используется вместо тега <a> для перехода по ссылкам (для переключения страниц).
Если оставить тег <a>, то браузер будет перезагружать страницы как обычно.

Поскольку в SPA страницы в действительности не перезагружаются, компонент Link использует history API браузера, чтобы обновить адрес в строке, но при этом не обновлять страницу и не перезагружать приложение.

Внутри этого компонента находится тот же href, только он дополнительно обрабатывается так, чтобы страница браузера не перезагружалась.

```
import { Link } from 'react-router-dom';

<header className='header'>

  <Link to='/'>Главная</Link>
  <Link to='/first'>Первая</Link>
  <Link to='/second'>Вторая</Link>

</header>
```

</>

Redux

Общее

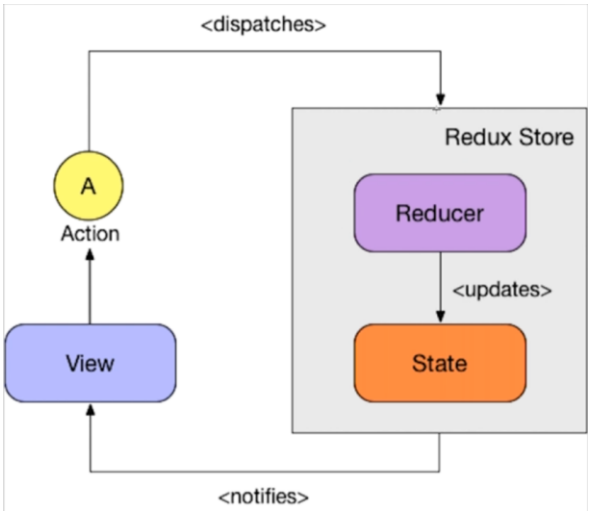
Redux – это независимая библиотека для управления глобальным state всего приложения.
Это паттерн и библиотека.
Решает проблему управления состоянием в приложении путём хранения state в одном «глобальном» объекте.
Управление реализуется через «actions», которые сообщают, какое действие по изменению state надо выполнить и, если нужно, предоставляют необходимые для изменения данные.

Redux рекомендуется к использованию, когда:
- приложение использует «большой» state
- приложение часто обновляет state
- логика обновления state сложная
- в приложении много кода и над ним работают много людей

Документация здесь.
Redux Fundamentals здесь.
Документация по React-Redux здесь.
Пример и описание минимального приложения здесь.
Годно про Redux на русском: rajdee.gitbooks.io

Главные элементы:
Store – объект, который хранит в себе глобальный state и координирует обновления (отвечает за логику изменения state).
Reducer – функция, которая занимается обновлением глобального state в ответ на Actions (события в redux называются действия).
Action – обычный JS-объект, который описывает, что именно надо сделать. Reducer его получает, изменяет state и ui перерендеривается.
Dispatch – операция по изменению state.

Компоненты могут только читать значения из глобального State, но не писать в него (непосредственно).
Для того, чтобы изменить глобальный State, мы создаём объекты Action и передаём их в Store. Эта операция называется dispatch.
Store использует функцию, которая называется Reducer, чтобы обновить глобальный State. Reducer реагирует на actions и обновляет State.



Установка библиотек

Redux – оригинальная небольшая JS-библиотека
React-Redux – библиотека для удобной интеграции с React.

Редакс обычно используется с плагинами, которые указаны тут.
Расширение для браузеров для отладки: Redux DevTools Extension здесь.
Redux Toolkit – набор пакетов, который рекомендуют создатели здесь.

```
Установка библиотек
npx create-react-app redux-sandbox
npm install redux react-redux

Подключение к чистому HTML
<script src="https://unpkg.com/redux@latest/dist/redux.min.js"></script>
```

Reducer

Функция, которая умеет реагировать на actions и обновлять state.
Словарь здесь.
Структура редъюсеров здесь.

```
const reducer = (state = initialState, action) => newState
```

```
const store = createStore(reducer);
```

state
Текущее состояние.
Он прописывается только один раз при создании функции reducer (указывается его первичное значение), а дальше redux использует его автоматом.

action
Это обычный объект, который должен содержать свойство .type.
Значение свойства type – это читаемая срока, которая описывает действие, которое надо совершить со state.

Логика работы:
Функция reducer читает action.type, выбирает функцию для его обработки, совершает необходимые действия и возвращает новый state. Старый state нельзя мутировать.
Если type неизвестен, то reducer должен вернуть state без изменений.

1. Проверить, работает ли reducer с переданным action.
2. Если да – сделать копию state, внести изменения и вернуть эту изменённую копию.
3. Если нет – вернуть исходный state без изменений.

Правила:
Reducer – это всегда чистая функция:
1. Reducer работает только с даными из аргументов state и action. Извне данные не берутся.
2. Существующий state не мутируется, возвращается его изменённая копия.
3. Не должно быть асинхронщины, рандомных значений, таймеров, записи значений в localStore, запросов на сервер и т.д.

Пример из курса:

```
const initialState = 0

const reducer = (state = initialState, action) => {
  switch (action.type) {
    case 'RND':
      return state + action.payload;

    case 'INC':
      return state + 1;

    case 'DEC':
      return state - 1;

    default:
      return state;
  }
};
```

Пример из руководства Redux:

```
const initialState = { value: 0 }

function reducer(state = initialState, action) {

  if (action.type === 'counter/incremented') {
    return { ...state, value: state.value + 1 }
  }

  return state
}
```

action

Документация здесь.
Действия – это простой JS объект с обязательным полем type, которые отправляются в reducer в целях изменения state. Свойства .type могут быть строками или символами, но строки предпочтительнее, потому что их можно сериализировать.

Функция reducer читает action.type, выбирает функцию для его обработки, совершает необходимые действия и возвращает новый state.

Любые данные типа событий интерфейса, сетевые запросы и др. должны в конечном итоге отправляться как действия (dispatched as actions). Через action можно передавать в reducer случайные числа и всё такое.

Кроме свойства type, action также может содержать любые другие дополнительные свойства (payload).

action creator
Функция, которая возвращает объект action.

```
простой объект
const action = {type: 'INC'}

как функция action creator
const action = () => ({type: 'INC'})

с дополнительными параметрами
const rnd = (payload) => ({type: 'RND', payload});

отправить action
store.dispatch(action)
```

Как правило, их складывают в отдельный файл, потому что со временем их набирается оч много.

```
actions.js
export const inc = () => ({type: 'INC'});
export const dec = () => ({type: 'DEC'});
export const rnd = (payload) => ({type: 'RND', payload});
```

Store

После создания функции reducer, создаётся store.
Store API здесь.

The Redux Store - центральный элемент каждого приложения, который хранит глобальный state. Это обычный js-объект с некоторыми специфичными функциями.

Правила:
- нельзя менять state внутри store напрямую.
- единственный способ поменять state – создать объект action, в котором записано, что именно в приложениии произошло, а затем отправить (dispatch) этот action в store.
- когда action отправлено, Store запускает функцию reducer. Эта функция создаёт новый state с использованием action.
- Store уведомляет подписчиков, что state обновился и UI может быть обновлён новыми данными.

Store создаётся через библиотеку Redux.
Предварительно должна быть готова функция reducer, потому что она передаётся в store при создании.

Создание

```
import { createStore } from 'redux';
const store = Redux.createStore(reducer)
```

Методы Store

```
store.getState()
возвращает текущий state

store.dispatch(action)
Внести изменения в store
(store вызовет reducer с указанным action)
Возвращает обновлённый state. Подписчики также уведомляются об обновлении.

store.subscribe(listener)
Подписывает функцию-коллбэк на «прослушку» state.
listener – это коллбэк, который будет вызываться каждый раз, когда обновляется state.

store.unsubscribe
Не понял, как работает.
To unsubscribe the change listener, invoke the function returned by subscribe.

store.replaceReducer(nextReducer)
Replaces the reducer currently used by the store to calculate the state.
```

createStore принимает в качестве аргумента функцию reducer.
С помощью этой функции в store создаётся initial state. Именно поэтому функция должна иметь аргумент по умолчанию.
Store будет использовать переданный reducer для дальнейших обновлений.

.dispatch:
При вызове dispatch, store вызовет reducer с указанным action.
Можно написать код, который будет диспатчить только если определённые условия true.
Или асинхронный код, который будет диспатчить после задержки.
Можно передавать .dispatch внутрь функций-коллбэков addEventListener.

```
store.dispatch({type: 'INC'});
```

UI

Redux может работать с любым ui, это самостоятельная, независимая библиотека.
Так, например, redux спокойно работает (и обновляет) нативный HTML. Для этого его надо подключить в header вот так:

```
<script src="https://unpkg.com/redux@latest/dist/redux.min.js"></script>
```

В примере используются функции getState и subscribe.

```
const valueEl = document.getElementById('value')

// Whenever the store state changes, update the UI by
// reading the latest store state and showing new data
function render() {
  const state = store.getState()
  valueEl.innerHTML = state.value.toString()
}

// Update the UI with the initial data
render()
```

```
// And subscribe to redraw whenever the data changes in the future
store.subscribe(render)
```

Как только state будет меняться, redux обновит интерфейс:
- Создаётся функция render.
- Внутри она считывает значение из state и вставляет его в html-элемент.
- Функция render подписывается на изменения в store и вызывается каждый раз, когда эти изменения происходят.

Redux пошагово

Импорт

```
import { createStore } from 'redux';
```

Создать **reducer**

```
const initialState = { value: 0 }

function reducer(state = initialState, action) {

  if (action.type === 'counter/incremented') {
    return { ...state, value: state.value + 1 }
  }

  return state
}
```

Создать **store**, передать в него reducer,
Смотреть текущее состояние:

```
const store = createStore(reducer);
store.getState()

store.getState()
store.dispatch(action)
store.subscribe(listener)
store.unsubscribe
store.replaceReducer(nextReducer)
```

Создать **action** и диспатчить их в store:

```
const action = {type: 'INC'};

const rnd = (payload) => ({type: 'RND', payload});

store.dispatch(action);
```

Подписать store на обновления.
store.subscribe позволяет получать нотификации, когда store изменился (при диспатче).
Аргументом передаётся функция, которая будет запускаться, когда store изменился.

```
// при изменении store будет выводиться печать его состояния
store.subscribe( () => {
```

```
  console.log(store.getState);
});
```

Раскидать actions и reducer по отдельным файлам.

Обернуть actionCreator в bindActionCreators

```
index.js

import { createStore, bindActionCreators } from 'redux';
import reducer from './reducer';
import * as actions from './actions';

const store = createStore(reducer);
const { dispatch } = store;

const {inc, dec, rnd} = bindActionCreators( actions, dispatch );

document.getElementById('inc').addEventListener('click', inc);
document.getElementById('dec').addEventListener('click', dec);

document.getElementById('rnd').addEventListener('click', () => {
  const payload = Math.floor(Math.random() * 10);
  rnd(payload);
});


const update = () => {
  const value = store.getState();
  document.getElementById('counter').innerHTML = value;
}

store.subscribe(update);
```

Паттерны

Страница в документации с паттернами тут.
React-Redux паттерны тут.

bindActionCreators

Документация здесь.
Используется для того, чтобы збавиться от постоянного store.dispatch(action) в коде.

Далее в примерах будут использоваться два action creators:

```
export const inc = () => ({type: 'INC'});
export const rnd = (payload) => ({type: 'RND', payload});
```

Чтобы сократить код и не дублировать действия, можно деструктурировать dispatch из store:

```
было
store.dispatch(action)

стало
const store = createStore(reducer);
const {dispatch} = store;

dispatch(action)
```

Следующее улучшение. Каждый раз, когда обрабатывается событие, сначала создаётся action, а потом – передача его в dispatch. Этот процесс повторяется каждый раз, когда надо что-то отправить. Можно создать одну функцию, которая будет выполнять оба этих действия:

```
было
document
  .getElementById('inc')
  .addEventListener('click', () => {
    dispatch(inc())
});

стало
const store = createStore(reducer);
const {dispatch} = store;
const incDispatch = () => dispatch(inc());

document
  .getElementById('inc')
  .addEventListener('click', incDispatch);
```

Создавать вручную такие action не удобно. Можно написать свою функцию, которая будет оборачивать каждый actionCreator в dispatch.

```
было
const incDispatch = () => dispatch(inc());
const rndDispatch = (payload) => dispatch(rnd(payload));

стало
const bindActionCreator = (creator, dispatch) => (...args) => {
  dispatch(creator(...args));
}
...args нужен на тот случай, если передаются payload. Функция будет одинаково хорошо работать
как с ас без аргументов, так и с аргументами.
Например, в функцию со случайным числом, которой нужны payload, аргументы будут передаваться
автоматом:

const incDispatch = bindActionCreator(inc, dispatch);
const rndDispatch = bindActionCreator(rnd, dispatch);
```

bindActionCreators
Такое связывание action creator с функцией dispatch является стандартным паттерном в redux. Для этого не надо писать свою функцию. Специальный метод bindActionCreators используется для того, чтобы обернуть каждый actionCreator в dispatch. В качестве результата он вернёт объект, свойством которого являются обёрнутые функции, а ключи – названия этих обёрток.

Важно! Не забыть импортировать:

```
import { createStore, bindActionCreators } from 'redux';
```

Можно использовать так же, как самописную функцию из примера выше, всё будет работать:

```
const incDispatch = bindActionCreators(inc, dispatch);
const rndDispatch = bindActionCreators(rnd, dispatch);
```

bindActionCreators позволяет обернуть сразу несколько action-creator.

Для этого, вместо первого аргумента, нужно передать объект. Ключи этого объекта – название новых функций-обёрток, которые хотим получить, а значения – актуальные action creators.
Функция вернёт объект с ключами, которые имеют такие же названия, как новые функции-обёртки.

```
было
const incDispatch = bindActionCreators(inc, dispatch);
const rndDispatch = bindActionCreators(rnd, dispatch);

стало
import { createStore, bindActionCreators } from 'redux';
import { inc, rnd  } from './actions';

const actionCreatorsObject = bindActionCreators({
  incDispatch: inc,
  rndDispatch: rnd,
}, dispatch);


actionCreatorsObject.inc()
actionCreatorsObject.rnd()
```

Можно сразу деструктуризировать этот объект:

```
const {incDispatch, decDispatch, rndDispatch} = bindActionCreators({
  incDispatch: inc,
  decDispatch: dec,
  rndDispatch: rnd,
}, dispatch);
```

Можно ещё сильнее сократить этот код:

```
import * as actions from './actions';
импортировать всё из файла

const {inc, dec, rnd} = bindActionCreators(actions, dispatch);
```

Разнести actions и reducer по отдельный файлам, а затем применить bondActionCreators – хорошая практика, потому что теперь компоненты, которые отвечают за визуальную часть, зависят только от функций {inc, dec, rnd} и ничего не знают о том, как работает их логика.
Более того, приложение можно переписать без использования redux: просто передать другие функции.
Маленькие компоненты с небольшим количеством зависимостей – залог чистого кода, который легко поддерживать.

Юрий Бура

Reducer

Функция, которая умеет реагировать на actions и обновлять state.
Словарь здесь.
Структура редъюсеров здесь.

```
Создание
const reducer = (state, action) => newState

state
текущее состояние
action
обычный JS-объект, в котором свойство type описывает действие, которое нужно совершить.
```

```
Кроме обязательного свойства type, объект action также может содержать любые другие
дополнительные свойства. Обычно поле с доп. параметрами называется payload.
```

state
этот аргумент – текущее состояние, тут всё понятно.
Если state – undefined, то нужно вернуть первоначальный (initial) state.

action
Это обычный объект, который обязательно должен содержать свойство type.
С помощью свойства type функция reducer будет понимать, что именно нужно сделать. Это как «имя» для действия, которое надо совершить. Например, увеличить счётчик (action.type === 'INC').
Если тип action неизвестен – нужно вернуть state без изменений.
Кроме свойства type, объект action может содержать почти любую другую вспомогательную информацию.

Правила:
Если state – undefined, то нужно вернуть первоначальный (initial) state.
Если тип action неизвестен – нужно вернуть state без изменений.

Reducer должна быть чистой функцией:
1. Возвращаемое значение завсит только от аргументов. Аргументы – это текущий state и action. Результат работы функции reducer – новый state. Он должен зависить исключительно от старого state и action и не от каких других параметров, которые берутся извне.
2. У reducer не может быть побочных эффектов: таймеров, записи значений в localStore, запросов на сервер и т.д. Это помогает в крупных приложениях.

Пример:

```
const reducer = (state = 0, action) => {
  switch (action.type) {
    case 'RND':
      return state + action.payload;

    case 'INC':
      return state + 1;

    case 'DEC':
      return state - 1;

    default:
      return state;
  }
};
```

Redux Store

Это центральный объект в системе Redux. Его задача – координировать работу других компонентов.
Центральное место в нём занимает reducer.
Что касается state, то его можно получить из функции reducer, как в примере выше.
Документация по Store API здесь.

Store содержит внутри себя state tree. Чтобы изменить state, нужно отправить action.
Это обычный объект с несколькими методами, не класс.
Создание store – это передача reduce-функции в функцию createStore().

```
Создать Store
import { createStore } from 'redux';
const store = createStore(reducer);
```

```
Методы Store
.getState()
возвращает текущий state

.dispatch(action)
Обработать новый action и внести изменения в store.
action – это всегда объект типа {type: 'ACTION'}

.subscribe(listener)
Получать нотификации об изменениях store.
Change listener. It will be called any time an action is dispatched.

.replaceReducer(nextReducer)
```

```
Пример
// создание
const store = createStore(reducer);

// будет срабатывать при изменении store
store.subscribe(() => {
  console.log(store.getState());
});

// изменение store
store.dispatch({type: 'INC'});
store.dispatch({type: 'INC'});
```

UI для Redux

В качестве UI может использоваться любая библиотека или фреймворк.
Например:

```
HTML
<h2 id="counter">0</h2>
<button id="dec">DEC</button>
<button id="inc">INC</button>

JS
const reducer = (state = 0, action) => {
  switch (action.type) {
    case 'INC': return state + 1;
    case 'DEC': return state - 1;
    case 'RND': return state + action.payload;

    default: return state;
  };
};

const store = createStore(reducer);

document
  .getElementById('inc')
  .addEventListener('click', () => {
    store.dispatch({type: 'INC'});
  });

document
  .getElementById('dec')
  .addEventListener('click', () => {
    store.dispatch({type: 'DEC'});
  });
```

```
document
  .getElementById('rnd')
  .addEventListener('click', () => {
    const payload = Math.floor(Math.random() * 10);
    store.dispatch({ type: 'RND', payload });
});


const update = () => {
  document
    .getElementById('counter')
    .innerHTML = store.getState();
}

store.subscribe(update);
```

Передача payload

Если нужно передать случайные данные в reducer, то это делается не в reducer, а где угодно снаружи.
В примере из курса функция создаёт значение до action-объекта, а потом добавляет в него готовое значение.

```
document.getElementById('rnd').addEventListener('click', () => {
  const payload = Math.floor(Math.random() * 10);
  store.dispatch({ type: 'RND', payload });
});
```

Action creator

Объекты action – это интерфейс общения между компонентами приложения и объектом store. Они используются для изменения состояния. В большом приложении экшены проще создавать через функции.

Функции, создающие action-объекты, называются action creator.

Самый элементарный способ их создать:

```
const inc = () => ( {type: 'INC'} );
const dec = () => ( {type: 'DEC'} );
const rnd = (payload) => ( {type: 'RND', payload} );


document.getElementById('inc').addEventListener('click', () => {
  store.dispatch( inc() );
});

document.getElementById('dec').addEventListener('click', () => {
  store.dispatch( dec() );
});

document.getElementById('rnd').addEventListener('click', () => {
  const payload = Math.floor(Math.random() * 10);
  store.dispatch( rnd(payload) );
});
```

Структура проекта

АС и ред выносятся в отдельные файлы.
Они не зависят ни от редакс, ни от других библиотек.

```
actions.js
export const inc = () => ( {type: 'INC'} );
export const dec = () => ( {type: 'DEC'} );
export const rnd = (payload) => ( {type: 'RND', payload} );


reducer.js
const reducer = (state = 0, action) => {
  switch (action.type) {
    case 'INC': return state + 1;
    case 'DEC': return state - 1;
    case 'RND': return state + action.payload;

    default: return state;
  };
};

export default reducer;


index.js
import { createStore } from 'redux';
import reducer from './reducer';
import {inc, dec, rnd } from './actions';
```

bindActionCreators

чтобы постоянно не использовать store.dispatch, можно вынести эту функцию:

```
было:
store.dispatch( inc() );


стало:
const store = createStore(reducer);
const { dispatch } = store;

document.getElementById('inc').addEventListener('click', () => {
  dispatch( inc() );
});
```

В каждой функции повторяется один и тот же процесс: dispatch(actionCreator()).
Его можно сократить, сделав композиция функций вручную:

```
было:
dispatch( inc() );


стало:
import {inc, dec, rnd } from './actions';
const store = createStore(reducer);
const { dispatch } = store;

const incDispatch = () => dispatch(inc());
const decDispatch = () => dispatch(dec());
const rndDispatch = (payload) => dispatch(rnd(payload));


document.getElementById('inc').addEventListener('click', incDispatch);
document.getElementById('dec').addEventListener('click', decDispatch);

document.getElementById('rnd').addEventListener('click', () => {
```

```
  const payload = Math.floor(Math.random() * 10);
  rndDispatch(payload);
});
```

Если приложение будет большое, то все функции не обернёшь вручную в dispatch.
Можно написать функцию для этого. Она принимает 2 параметра: ac и dispatch, и возвращает функцию, которая их оборачивает.
Некоторым ac Нужен payload и заранее неизвестно, сколько аргументов хочет получить ac. Поэтому во второй возвращаемой функции надо использовать rest и spread операторы.

```
было:
const incDispatch = () => dispatch(inc());
const decDispatch = () => dispatch(dec());
const rndDispatch = (payload) => dispatch(rnd(payload));


стало:
import {inc, dec, rnd } from './actions';
const store = createStore(reducer);
const { dispatch } = store;

const bindActionCreator = (actionCreator, dispatch) => (...args) => {
  dispatch(actionCreator(...args));
}

const incDispatch = bindActionCreator(inc, dispatch);
const decDispatch = bindActionCreator(dec, dispatch);
const rndDispatch = bindActionCreator(rnd, dispatch);
```

Использование точно такое же, как и раньше (см. выше).

bindActionCreators из Redux
Связывание ac и диспатч – это типичный паттерн для redux, поэтому в redux есть своя версия такой функции.
Её можно импортировать и использовать так же, как и свою рукописную:

```
import { createStore, bindActionCreators } from 'redux';

const incDispatch = bindActionCreators(inc, dispatch);
const decDispatch = bindActionCreators(dec, dispatch);
const rndDispatch = bindActionCreators(rnd, dispatch);
```

bindAC позволяет оборачивать за один раз много функций. В него можно передать объект первым параметром, а вторым – dispatch.
Ключи – название функций, которые будут использоваться.
Значения – название ac, которые надо обернуть. Всё как при деструктуризации.

Если передать в эту функцию объект из ac, то они автоматически будут обёрнуты в dispatch.

В качестве результата bindAC вернёт объект с такой же структурой, как описана выше:

```
import { createStore, bindActionCreators } from 'redux';
import reducer from './reducer';
import {inc, dec, rnd } from './actions';

const store = createStore(reducer);
const { dispatch } = store;

const bindObject = bindActionCreators(
```

```
  { incDispatch: inc, decDispatch: dec, rndDispatch: rnd },
  dispatch
);

// использование
bindObject.incDispatch;
bindObject.decDispatch;
bindObject.rnd(payload);
```

Можно сразу деструктуризировать возвращаемый объект и использовать готовые функции по отдельности:

```
const {incDispatch, decDispatch, rndDispatch} = bindActionCreators(
  { incDispatch: inc, decDispatch: dec, rndDispatch: rnd },
  dispatch
);

// использование
incDispatch;
decDispatch;
rndDispatch(payload);
```

Этот код можно ещё улучшить, если сделать импорт вот так:

```
import * as actions from './actions';

console.log( actions )  // Object { inc: Getter, dec: Getter, rnd: Getter }

const {inc, dec, rnd} = bindActionCreators( actions, dispatch );
```

Итоговый код (чтобы не искать в другом месте):

```
index.js

import { createStore, bindActionCreators } from 'redux';
import reducer from './reducer';
import * as actions from './actions';

const store = createStore(reducer);
const { dispatch } = store;

const {inc, dec, rnd} = bindActionCreators( actions, dispatch );

document.getElementById('inc').addEventListener('click', inc);
document.getElementById('dec').addEventListener('click', dec);

document.getElementById('rnd').addEventListener('click', () => {
  const payload = Math.floor(Math.random() * 10);
  rnd(payload);
});


const update = () => {
  const value = store.getState();
  document.getElementById('counter').innerHTML = value;
}

store.subscribe(update);
```

Использование React и Redux

Вместо базового DOM API будет использоваться React для отрисовки.
В этом уроке не используется react-redux, только ручное совмещение.
Использование чистого redux (не рекомендуется)

Сейчас в index.js есть 2 части:
- которая работает с redux
- которая работает с dom-деревом (всё, что начинается с document…)

Создаётся react-компонент для отрисовки, куда будут передаваться все нужные функции для отрисовки и вычисления значений. Поскольку компонент не обладает state (вместо него используется store), компонент будет обновляться через подписку store.subscribe(update).
Первый запуск функции update для отрисовки делается вручную. Повторные обновления делает store благодаря подписке.

React-компоненты должны как можно меньше использовать внутри себя библиотеку redux. В идеале - вообще ничего о нём не знать, только получать значения извне.

В примере ниже компонент не знает, как именно он будет обновляться. Он только отображает какое-то значение counter и при клике умеет вызывать функции-диспатчи.

Создаётся файл counter.js с компонентом:

```
counter.js
import React from 'react';

const Counter = ({ counter, inc, dec, rnd }) => {
  return (
    <div id="root" className="jumbotron">
      <h2 id="counter">{counter}</h2>

      <button
      onClick={dec}
      id="dec" className="btn btn-primary btn-lg">DEC</button>

      <button
      onClick={inc}
      id="inc" className="btn btn-primary btn-lg">INC</button>

      <button
      onClick={rnd}
      id="rnd" className="btn btn-primary btn-lg">RND</button>
    </div>
  );
};

export default Counter;
```

Файл index.js
При клике на кнопки, компонент будет вызывать функции-диспатчи, которые поменяют store.
Функция update подписана на изменение store и через ReactDOM.render каждый раз при обновлении store добавляет на страницу обновлённый компонент. На самом деле, react будет использовать свой reconciliation алгоритм и обновлять только изменённые элементы.
Первый раз udate надо вызвать вручную.

```
index.js

import React from 'react';
import ReactDOM from 'react-dom';
import Counter from './counter';
```

```
import { createStore, bindActionCreators } from 'redux';
import reducer from './reducer';
import * as actions from './actions';

const store = createStore(reducer);
const { dispatch } = store;

const {inc, dec, rnd} = bindActionCreators( actions, dispatch );


const update = () => {

  const value = store.getState();
  const payload = Math.floor(Math.random()*10);

  ReactDOM.render(
    <Counter
      counter={value}
      inc={inc}
      dec={dec}
      rnd={ () => (rnd(payload)) }
    />,

    document.getElementById('root'));
};

store.subscribe(update);
update();
```

Другие файлы (чтобы не искать):

```
reducer.js
const reducer = (state = 0, action) => {
  switch (action.type) {
    case 'INC': return state + 1;
    case 'DEC': return state - 1;
    case 'RND': return state + action.payload;
    default: return state;
  };
};

export default reducer;


actions.js
export const inc = () => ({type: 'INC'});
export const dec = () => ({type: 'DEC'});
export const rnd = (payload) => ({type: 'RND', payload});
```

Генерацию случайных чисел и значения value можно прописать прям в пропсах. Я вынес для удобства.

React-redux

В этом уроке используется react-redux.
Документация здесь.

Кратко

```
index.js
import { createStore } from 'redux';
```

```
import { Provider } from 'react-redux';
import reducer from './reducer';

const store = createStore(reducer);

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);



component.js
import { connect } from 'react-redux';
const MyComponent = () => {…};

export default connect(mapState, mapDispatch)(Component)
```

Подробно
Для начала, надо как обычно переместить компоненты в components и всё организовать как обычно:

```
actions.js
reducer.js
index.js

components
  app.js
  counter.js
```

<**Provider** store={}>
Центральный элемент, вокруг которого всё происходит в Redux – это store. Он должен быть доступен для всех элементов одновременно.
Чтобы сделать его доступным, его нужно разместить в index.js и использовать контекст api. Библиотека react-redux предоставляет готовый инструмент, который основан на контексте и упрощает интеграцию, самому ничего писать не надо. Все детали интеграции, которые раньше приходилось писать, уже реализованы в нём, достаточно просто один раз отрендерить приложение.

Что делает:
1. передаёт store по иерархии
Компонент Provider основан на контексте и может передавать store по всей иерархии приложения.

2. автоматом обновляет приложение при изменении store.
Как только любой компонент из иерархии ниже вызовет dispatch, компонент Provider узнает об этом и обновит приложение.
Внутри себя он уже реализует подписку на изменение store и делает так, чтобы всё приложение обновлялось, как только store изменяется, поэтому вручную следить за обновлениями больше не надо.

```
index.js

import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/app';

import { createStore } from 'redux';
import { Provider } from 'react-redux';
import reducer from './reducer';
```

```
const store = createStore(reducer);

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);


// больше не надо
const update = () => { … }
store.subscribe(update);
update();
```

Connect()
Компоненты должны использовать диспатчи и информацию из store. Сейчас все компоненты по иерархии через контекст имеют доступ к store, но чтобы использовать его, надо знать, какие поля и диспатчи передавать в пропсы. Чтобы их передавать в компонент, используется функция connect.

Connect – это HOC, т.е. функция, которая делает какие-то подготовительные действия, а потом оборачивает компонент и предоставляет ему нужную инфо и возвращает новый, готовый компонент.
Документация здесь.

В качестве параметров она принимает параметры конфигурации:

map**State**ToProps – это функция, которая получает текущий state из store и её задача – вернуть объект с теми свойствами, которые нужны react-компоненту.

map**Dispatch**ToProps
Может быть функцией или объектом.
Эта функция получит на вход функцию dispatch автоматом. Возвращает объект, в качестве ключей – имена создаваемых функций-диспатчей, передаваемых в пропсы, а значения – action или actionCreator, которые будут обёрнуты в dispatch.

```
import { connect } from 'react-redux';

function connect(mapStateToProps, mapDispatchToProps, mergeProps, options)(Component)

export default connect(mapStateToProps, mapDispatchToProps)(Counter);


const mapStateToProps = (state) => {
  return {
    name: state.name,
    age: state.age };
};


const mapDispatchToProps = (dispatch) => {
  return {
    inc: () => dispatch({type: 'INC'})
  }
}


// с actionCreators
const mapDispatchToProps = (dispatch) => {
  return {
```

```
    inc: () => dispatch(inc()),
    dec: () => dispatch(dec()),
    rnd: () => {
      const randomValue = Math.floor(Math.random() * 10);
      dispatch(rnd(randomValue));
    }
  };
};


// Можно сократить ещё сильнее
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';
import * as actions from '../actions';

const mapDispatchToProps = (dispatch) => {

  const { inc, dec, rnd } = bindActionCreators(actions, dispatch);

  return {
    inc,
    dec,
    rnd: () => {
      const randomValue = Math.floor(Math.random()*10);
      rnd(randomValue);
    }
  }
};


rnd – это функция, которая сначала считает случайное число.
```

На практике выглядит так:

```
import React from 'react';
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux'

import * as actions from '../actions';


const Counter = ({ counter, inc, dec, rnd }) => {
  return (
    <div className="jumbotron">
      <h2 id="counter">{counter}</h2>

      <button
        onClick={dec}
        id="dec" className="btn btn-primary btn-lg">DEC</button>

      <button
        onClick={inc}
        id="inc" className="btn btn-primary btn-lg">INC</button>

      <button
        onClick={rnd}
        id="rnd" className="btn btn-primary btn-lg">RND</button>
    </div>
  );
};


const mapStore = (store) => ( {counter: store } );
```

```
const mapDispatch = (dispatch) => {
  const {inc, dec, rnd} = bindActionCreators(actions, dispatch);

  return {
    inc,
    dec,
    rnd: () => {
      const randomValue = Math.floor(Math.random() * 10);
      rnd(randomValue);
    }
  };
};


export default connect(mapStore, mapDispatch)(Counter);
```

</>

React + Redux

Установка

Установка:

```
npx create-react-app re-store
npm install prop-types react-router-dom redux react-redux
```

Универсальные компоненты

Большинству приложений необходимы на старте:

```
ErrorBoundry          отлавливатель ошибок
Контекст              для передачи инфо к нижестоящим по иерархии компонентам
HOC для контекста     withContextService для удобной работы с consumer
```

Index.js:

```
ReactDOM.render(
  <Provider store={store}>
    <ErrorBoundry>
      <BookstoreServiceProvider value={bookstoreService}>
        <BrowserRouter>
          <App />
        </BrowserRouter>
      </BookstoreServiceProvider>
    </ErrorBoundry>
  </Provider>
);
```

Папки и файлы на старте:

```
store.js

actions
```

```
  index.js

reducers
  index.js

components
  app
  error-boundry               отлавливатель ошибок
  error-indicator             элемент с визуалом ошибки
  spinner                     отображается во время загрузки по сети
  bookstore-service-context   будет передавать по иерархии сервис
  hoc
    with-bookstore-service    компонент-обёртка упрощает использование сервиса
  pages

services
  bookstore-service.js    сетевые запросы, трансформация полученных данных

utils
```

Порядок создания:
bookstore-service
app
error-indicator
spinner
error-boundry
bookstore-service-context

hoc => bookstore-service
pages (просто каталог)

Error-boundry максимально простой:

```
import React, { Component } from 'react';
import ErrorIndicator from '../error-indicator/';

export default class ErrorBoundry extends Component {

  state = { hasError: false };

  componentDidCatch() {
    this.setState({ hasError: true });
  };

  render() {
    if (this.state.hasError) return <ErrorIndicator/>

    return this.props.children;
  };
}
```

bookstore-service-context

```
import React from 'react';

const {
  Provider: BookstoreServiceProvider,
  Consumer: BookstoreServiceConsumer
```

```
} = React.createContext();

export {
  BookstoreServiceProvider,
  BookstoreServiceConsumer
};
```

WithBookstoreService
Это hoc-компонент-обёртка. Нужен для того, чтобы было удобнее использовать сервис: он обернёт в Consumer компонент, чтобы он мог использовать данные из Provider.
Не забыть, что в consumer в качестве дочернего элемента передаётся функция, которая получает в качестве аргумента value из провайдера.

```
import React from 'react';
import { BookstoreServiceConsumer } from '../bookstore-service-context/bookstore-service-
context';

const withBookstoreService = (ошибка?) => (Wrapped) => {

  return (props) => {
    return (
      <BookstoreServiceConsumer>
        {
          (bookstoreService) => {
            return (<Wrapped {...props} bookstoreService={bookstoreService} />);
          }
        }
      </BookstoreServiceConsumer>
    );
  };
};

export default withBookstoreService;
```

Необходимые компоненты готовы.

Redux компоненты

Порядок создания:
reducer
action
store

reducer.js

```
const initialState = {
  books: [],
};

const reducer = (state = initialState, action) => {
  switch (action.type) {

    case 'BOOKS LOADED':
      return { ...state, books: action.payload };

    default:
      return state;
  }
};
```

```
export default reducer;
```

booksLoaded – это функция action-creator, которой для работы нужно передать список книг. Она его получает, обращается в reducer, а reducer смотрит на type и записывает полученные payload куда надо.

```
const booksLoaded = (newBooks) => ({ type: 'BOOKS_LOADED', payload: newBooks });

export { booksLoaded };
```

store.js

```
import { createStore } from 'redux';
import reducer from './reducers';

const store = createStore(reducer);

export default store;
```

Каркас приложения

Сейчас будет сборка компонентов, чтобы приложение что-то отображало на странице и использовало redux. Пока без асинхронщины.

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom';

import store from './store';
import { Provider } from 'react-redux';
import { BrowserRouter } from 'react-router-dom';

import BookstoreService from './services/bookstore-service';
import { BookstoreServiceProvider } from './components/bookstore-service-context';

import ErrorBoundry from './components/error-boundry/error-boundry';
import App from './components/app';


const bookstoreService = new BookstoreService();

ReactDOM.render(
  <Provider store={store}>
    <ErrorBoundry>
      <BookstoreServiceProvider value={bookstoreService}>
        <BrowserRouter>

          <App />

        </BrowserRouter>
      </BookstoreServiceProvider>
    </ErrorBoundry>
  </Provider>,

  document.getElementById('root')
);
```

Если всё работает и на экране отображается App, то самое время дописать компонент app: компоненту надо передать контекст для использования сетевого запроса, используется hoc. После обёртывания, экземпляр сетевого запроса будет доступен через пропсы компонента.

Код ниже – это пример, как будет работать редакс. На самом деле, с редаксом будут работать другие компоненты, а не app, но сейчас можно проверить его работу.

```
import React from 'react';
import { withBookstoreService  } from '../hoc';
import './app.css';

const App = ({ bookstoreService }) => {
  console.log(bookstoreService.getBooks())
  return <div>App</div>;
};

export default withBookstoreService()(App);
```

В service можно докинуть dummy-данные:

```
const books = [
  {id: 1, title: 'Alice in Wonderland', author: 'Lewis Carroll'},
  {id: 2, title: 'Call of Cthulhu', author: 'Howard Lovecraft'},
];

export default class BookstoreService {

  getBooks() {
    return books;
  };
};
```

Роутинг

Привести App в исходное состояние (без redux), потому что с редаксом будут работать другие компоненты.

Создаётся папка pages, в неё добавляются 2 файла: home-page.js и cart-page.js.
Пока они ничего не делают, просто отображают своё название.

```
app.js
import React from 'react';
import { Routes, Route } from 'react-router-dom';
import './app.css';
import { HomePage, CartPage } from '../pages';

const App = () => {
  return (
  <div>
    <Routes>
      <Route path="/" element={HomePage} />
      <Route path="/cart" element={CartPage} />
    </Routes>
  </div>);
};

export default App;


home-page.js
import React, { Component } from 'react';
```

```
export default class HomePage extends Component {
  render() {
    return <div>Home</div>
  };
};


cart-page.js
import React from 'react';
const CartPage = () => {
  return <div>Cart</div>;
};
export default CartPage;
```

Внутри homepage будет находиться список книг.
Эти 2 компонента надо дописать и передать в него.

```
book-list-item.js
import React, { Fragment } from 'react';
import './book-list-item.css';
const BookListItem = ({ book }) => {
  const { title, author } = book;

  return (
    <Fragment>
      <span>{title}</span>
      <span>{author}</span>
    </Fragment>
  );
};
export default BookListItem;
```

BookList – более сложный компонент. Он будет делать сетевой запрос в componentDidMount, поэтому это должен быть класс-компонент.

```
book-list.js
```

Redux компоненты
Сначала reducer, потом action и store.
Reducer будет записывать в store полученный список книг.

```
reducers => index.js
```

```
const initialState = { books: [] };

const reducer = (state = initialState, action) => {
  switch (action.type) {

    case 'BOOKS_LOADED':
      return { books: action.payload };

    default: return state;
  }
};

export default reducer;


actions => index.js
const booksLoaded = (newBooks) => ({ type: 'BOOKS_LOADED', payload: newBooks });
export { booksLoaded };


store.js
import { createStore } from 'redux';
import reducer from './reducers';

const store = createStore(reducer);

export default store;
```

Provider store
На самом верхнем уровне – провайдер из react-redux, потому что это центр приложения.

ErrorBoundry
Как можно выше, чтобы отлавливал любые ошибки.

BookstoreServiceProvider
Это контекст, который будет передавать экземпляр сетевого сервиса ниже по иерархии.

Router
Чтобы все компоненты имели доступ к функциональности роутинга.

```
index.js

import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { BrowserRouter as Router } from 'react-router-dom';

import App from './components/app';
import ErrorBoundry from './components/error-boundry';
import BookstoreService from './services/bookstore-service';
import { BookstoreServiceProvider } from './components/bookstore-service-context'

import store from './store';

const bookstoreService = new BookstoreService();

ReactDOM.render(
  <Provider store={store}>
    <ErrorBoundry>
      <BookstoreServiceProvider value={bookstoreService}>
        <Router>
```

```
          <App />
        </Router>
      </BookstoreServiceProvider>
    </ErrorBoundry>
  </Provider>,
  document.getElementById('root')
);
```

Работа с асинхронными данными

Service – это эмуляция сетевого запроса. Чтобы сделать запрос асинхронным, надо прописать service так:

```
export default class BookstoreService {

  data = [ массив объектов с книгами ];

  getBooks() {
    return new Promise((resolve) => {
      return setTimeout(() => resolve(this.data), 500);
    });
  }
```

Дописать

Обработка ошибок

Статус ошибки хранится в store.
Ошибка – это null по умолчанию, а также если начался запрос к серверу или книги успешно получены.

В reducer добавляется новое действие. Через payload будут переданы детали ошибки.

```
Reducer

const initialState = {
  books: [],
  loading: true,
  error: null
};

const reducer = (state = initialState, action) => {
  switch (action.type) {

    case 'BOOKS_REQUESTED':
      return { books: [], loading: true, error: null };

    case 'BOOKS_LOADED':
      return { books: action.payload, loading: false, error: null };

    case 'BOOKS_ERROR':
      return { books: [], loading: false, error: action.payload };

    default: return state;
  }
};

export default reducer;
```

actions

```
const booksError = (error) => ({ type: 'BOOKS_ERROR', payload: error });
```

book-list
Протаскивание в пропсы – как всегда через mapDispatch и mapState.
Протаскивание через mapState надо, чтобы отобразить текст ошибки. Перед этим она запишется в state, где раньше был null.

Не забыть обновить render, если есть ошибка. Обновление тривиальное.

```
const mapState = (state) => {
  return {books: state.books, loading: state.loading, error: state.error};
};

const mapDispatch = {booksLoaded, booksRequested, booksError};


  componentDidMount() {
    const { bookstoreService, booksLoaded, booksRequested, booksError } = this.props;
    booksRequested();
    bookstoreService.getBooks()
      .then((data) => booksLoaded(data))
      .catch((err) => booksError(err));
  }


  render() {
    const { books, loading, error } = this.props;

    if (loading) return <Spinner />

    if (error) {
      return <ErrorIndicator/>
    }
```

аргумент ownProps

Сейчас компонент book-list в методе componentDidMount получает много свойств и выполняет много действий, которые его прямой обязанностью не являются. По факту ему нужно только то, что содержится в финкции render.

Всю логику сетевого запроса и обработки ошибки можно вынести в отдельную функцию.

```
books-list.js

componentDidMount() {

//  const { bookstoreService, booksLoaded, booksRequested, booksError } = this.props;
//  booksRequested();
//  bookstoreService.getBooks()
//    .then((data) => booksLoaded(data))
//    .catch((err) => booksError(err));

    this.props.fetchBooks();
  }
```

mapDispatch может быть либо объектом, либо функцией.
В случае функции, он принимает первым параметром dispatch и возвращает объект {названиеСвойства: ac}

Прикол в том, что в качестве значения в этот объект не обязательно оборачивать action-creator, а можно передать что угодно.

```
Books-list

Было:
```

```
const mapDispatch = {booksLoaded, booksRequested, booksError};

стало:
const mapDispatch = (dispatch) => {
  return {
    fetchBooks: () => {
      dispatch(booksRequested());

      bookstoreService.getBooks()
        .then((data) => dispatch(booksLoaded(data)) )
        .catch((err) => dispatch(booksError(err)) );
    }
  }
}
```

АС надо обернуть в dispatch, чтобы они заработали.
Сейчас есть одна проблема: bookstoreService недоступен.

У функций mapState и mapDispatch есть второй параметр: ownProperty. Если компонент во что-то обёрнут, то через второй параметр будут доступны собственные свойства создаваемого компонента.

Получается, что в ownProps будет bookstoreService.

```
const mapDispatch = (dispatch, ownProps) => {
  const { bookstoreService } = ownProps;

  return {
    fetchBooks: () => {
      dispatch(booksRequested());

      bookstoreService.getBooks()
        .then((data) => dispatch(booksLoaded(data)) )
        .catch((err) => dispatch(booksError(err)) );
    }
  }
}

export default compose(
  withBookstoreService(),
  connect(mapState, mapDispatch),
)(BookList);
```

Naming Convention

Несмотря на то, что функция fetchBooks не связана с работой store, она используется только там, где используются action-creators. Поэтому её можно положить с ними в один файл.

Аргументы передаются в первую функцию, чтобы компонент не заботился об этих пргументах, а просто вызывал функцию (с пустыми скобками).
Больше не имеет смысла экспортировать ac, потому что единственное место, где они используются – это функция fetchBooks.

```
Actions

const booksLoaded = (newBooks) => ({ type: 'BOOKS_LOADED', payload: newBooks });
const booksRequested = () => ({ type: 'BOOKS_REQUESTED' });
const booksError = (error) => ({ type: 'BOOKS_ERROR', payload: error });

const fetchBooks = (bookstoreService, dispatch) => () => {
  dispatch(booksRequested());
```

```
  bookstoreService.getBooks()
    .then((data) => dispatch(booksLoaded(data)) )
    .catch((err) => dispatch(booksError(err)) );
}

export {
  fetchBooks
};
```

Book-list

```
const mapDispatch = (dispatch, ownProps) => {

  const { bookstoreService } = ownProps;
  return { fetchBooks: fetchBooks(bookstoreService, dispatch) };
};


Можно сразу деструктурировать:
const mapDispatch = (dispatch, { bookstoreService } ) => {
  return { fetchBooks: fetchBooks(bookstoreService, dispatch) };
};
```

Именование
Действия, которые отправляют запрос,

```
BOOKS_REQUESTED => FETCH_BOOKS_REQUEST   данные именно получаются, а не обновляются.
                   UPDATE_BOOKS_REQUEST  обновление данных

BOOKS_LOADED    => FETCH_BOOKS_SUCCESS   успешное получение данных
BOOKS_ERROR     => FETCH_BOOKS_FAILURE   ошибка
```

Компоненты-контейнеры

В React считается хорошей практикой разделять компоненты, которые отвечают за поведение и компоненты, которые отвечают за отображение.

Сейчас book-list делает всё сразу: получает данные, обрабатывает состояния loading и error и формирует внешний вид компонента.

Book-list разбивается на 2 компонента:
- book-list-container отвечает за логику (спиннер, ошибка, запрос данных)
- book-list отвечает за отрисовку

Документация redux предписывает называть компоненты-обёртки container и размещать их в папке containers рядом с components. Но в этом проекте компоненты будут в одном файле.

```
const BookList = ({ books }) => {
  return (
    <ul className="book-list">
      {
        books.map((book) => {
          return (
            <li key={book.id}><BookListItem book={book} /></li>
          );
        })
```

```
      }
    </ul>
  );
};


class BookListContainer extends Component {

  componentDidMount() {
    this.props.fetchBooks();
  }

  render() {
    const { books, loading, error } = this.props;

    if (loading) return <Spinner />
    if (error) return <ErrorIndicator/>

    return <BookList books={books}/>
  }
};
```

```

```

```

```

```

```

```

```

</>

Модули

Модуль обычно содержит класс или библиотеку с функциями. Система модулей на уровне языка появилась в стандарте JavaScript в 2015 году. На данный момент она поддерживается большинством браузеров и Node.js.

Модуль – это просто файл. Один скрипт – это один модуль. Модули могут загружать друг друга и использовать директивы export и import, чтобы обмениваться функциональностью:
export отмечает переменные и функции, которые должны быть доступны вне текущего модуля.
import позволяет импортировать функциональность из других модулей.

Так как модули поддерживают ряд специальных ключевых слов, и у них есть ряд особенностей, то необходимо явно сказать браузеру, что скрипт является модулем, при помощи атрибута <script type="module">.
Браузер автоматически загрузит и запустит импортированный модуль (и те, которые он импортирует, если надо), а затем запустит скрипт.

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

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

Udemy модули

Локальные

Для того, чтобы импортировать что-то из файла, используется указание относительного пути.

```
// именованный экспорт и импорт указанных функций или констант (просто перечислить)
export { add, subtract, multiply };
import { add, subtract } form './file1';


// переименование экспортируемых и импортируемых сущностей
export { add as newAddName, subtract as newSubName };
import { add as newAddName, subtract as newSubName } form './file1';


// импорт всего и сразу
import * as calc from './file1';
calc.add()


// экспорт по умолчанию без имён. Скобки не ставятся. Используется как переменная.
export default add;
import add from './file1';        // импорт по умолчанию можно переименовать
import newAddName from './file1';
add()


// можно комбинировать синтаксис импорта по умолчанию и именованные импорты
export { subtract, multiply, divide };
export default add;

import add, {subtract, multiply} from './file1';
import add, * as calc from './file1';
```

```
// можно прописывать экспорт прям перед объявлением переменной
export default class Graph {…}
```

Если требуется просто запустить код, который находится в другом файле (например, для сайд-эффектов), можно сделать так:

```
file1.js
console.log('Side effect');

запуск из другого файла
import ' ./file1';
```

Некоторые сборщики могут прописывать такой паттерн, чтобы информировать о том, что они от чего-то зависят. Например, от CSS-файла. После этого сборщик будет знать: чтобы запустить это файл, нужно в проект включить зависимость. Но это поведение специфично для разных сборщиков.

Глобальные

Если есть внешняя библиотека, установленная, например, через NPM, синтаксис импорта будет отличаться.
Для того, чтобы импортировать что-то из файла, использовалось указание относительного пути.
Для библиотеки, которая установлена, как зависимость, указывается только её наименование.

```
import 'joke' from 'one-line=joke'

// использование
joke.getRandomJoke().body;
```

```

```

```

```

```

```

</>

Особенности модулей

Отличие модулей от обычных скриптов:

Всегда «use strict»
В модулях всегда используется режим use strict. Например, присваивание к необъявленной переменной вызовет ошибку.

```
<script type="module">
  a = 5;
  // ошибка
</script>
```

Своя область видимости переменных
Каждый модуль имеет свою собственную область видимости. Переменные и функции, объявленные в модуле, не видны в других скриптах. Поэтому нельзя импортировать модуль и просто так обращаться к его переменным.

Модули должны экспортировать функциональность, предназначенную для использования извне. А другие модули могут её импортировать.

Правильный вариант импорта из модуля:

```
user.js
export let user = "John";

hello.js
import {user} from './user.js';
document.body.innerHTML = user; // John

index.html
<!doctype html>
<script type="module" src="hello.js"></script>
```

В браузере также существует независимая область видимости для каждого оидельного скрипта <script type="module">:

```
<script type="module">
  // Переменная доступна только в этом модуле
  let user = "John";
</script>

<script type="module">
  alert(user);
  // Error: user is not defined
</script>
```

Если нам нужно сделать глобальную переменную уровня всей страницы, можно явно присвоить её объекту window, тогда получить значение переменной можно обратившись к window.user. Но это должно быть исключением, требующим веской причины.

Код в модуле выполняется только один раз при импорте
Если один и тот же модуль используется в нескольких местах, то его код выполнится только один раз, после чего экспортируемая функциональность передаётся всем импортёрам.

Это очень важно для понимания работы модулей.
Во-первых, если при запуске модуля возникают побочные эффекты, например выдаётся сообщение, то импорт модуля в нескольких местах покажет его только один раз – при первом импорте:

```
// alert.js
alert("Модуль выполнен!");
// Импорт одного и того же модуля в разных файлах

// 1.js
import `./alert.js`;
// Модуль выполнен!

// 2.js
import `./alert.js`;
// (ничего не покажет)
```

На практике, задача кода модуля – это обычно инициализация, создание внутренних структур данных, а если мы хотим, чтобы что-то можно было использовать много раз, то экспортируем это.

Более продвинутый пример – модуль экспортирует объект admin.
Если модуль импортируется в нескольких файлах, то объект admin станет общим для них всех. Если этот объект как-то изменится в одном модуле, то все остальные модули тоже увидят эти изменения.

В примере ниже оба файла 1 и 2 импортируют один и тот же объект. Изменения, сделанные в файле 1, будут видны в файле 2:

```
// 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
```

Такое поведение позволяет конфигурировать модули при первом импорте. Мы можем установить его свойства один раз, и в дальнейших импортах он будет уже настроенным.

Например, модуль admin.js предоставляет определённую функциональность, но ожидает передачи учётных данных в объект admin извне.
В init.js, первом скрипте нашего приложения, мы установим admin.name. Тогда все это увидят, включая вызовы, сделанные из самого admin.js
Другой модуль тоже увидит admin.name:

```
// admin.js
export let admin = { };

export function sayHi() {
  alert(`Ready to serve, ${admin.name}!`);
}

// init.js
import {admin} from './admin.js';
admin.name = "Pete";

// other.js
import {admin, sayHi} from './admin.js';

alert(admin.name); // Pete
sayHi();           // Ready to serve, Pete!
```

import.meta
Объект import.meta содержит информацию о текущем модуле. Содержимое зависит от окружения. В браузере он содержит ссылку на скрипт или ссылку на текущую веб-страницу, если модуль встроен в HTML:

```
<script type="module">
  alert(import.meta.url); // ссылка на html страницу для встроенного скрипта
</script>
```

В модуле «this» не определён
В модуле на верхнем уровне this не определён (undefined).
Сравним с не-модульными скриптами, там this – глобальный объект:

```
<script>
  alert(this); // window
</script>

<script type="module">
```

```
  alert(this); // undefined
</script>
```

Особенности в браузерах

Есть и несколько других, именно браузерных особенностей скриптов с type="module" по сравнению с обычными скриптами.

#? Не понимаю эту хуйню, тут тема переплетается с DOM.

Модули являются отложенными (deferred)
Модули *всегда* выполняются в отложенном (deferred) режиме, точно так же, как скрипты с атрибутом defer (описан в главе Скрипты: async, defer). Это верно и для внешних и встроенных скриптов-модулей.

Другими словами:
- загрузка внешних модулей, таких как <script type="module" src="...">, не блокирует обработку HTML.
- модули, даже если загрузились быстро, ожидают полной загрузки HTML документа, и только затем выполняются.
- сохраняется относительный порядок скриптов: скрипты, которые идут раньше в документе, выполняются раньше.

Как побочный эффект, модули всегда видят полностью загруженную HTML-страницу, включая элементы под ними.

```
<script type="module">
  alert(typeof button);
  // object
  // скрипт может 'видеть' кнопку под ним так как модули являются отложенными, то скрипт начнёт выполнятся только после полной загрузки страницы
</script>

Сравните с обычным скриптом:
<script>
  alert(typeof button);
  // Ошибка, кнопка не определена
  // скрипт не видит элементы под ним обычные скрипты запускаются сразу, не дожидаясь полной загрузки страницы
</script>

<button id="button">Кнопка</button>
```

Модули начинают выполняться после полной загрузки страницы. Обычные скрипты запускаются сразу же, поэтому сообщение из обычного скрипта мы видим первым: второй скрипт выполнится раньше, чем первый. Поэтому мы увидим сначала undefined, а потом object.

При использовании модулей нам стоит иметь в виду, что HTML-страница будет показана браузером до того, как выполнятся модули и JavaScript-приложение будет готово к работе. Некоторые функции могут ещё не работать. Нам следует разместить «индикатор загрузки» или что-то ещё, чтобы не смутить этим посетителя.

Атрибут async работает во встроенных скриптах
Для не-модульных скриптов атрибут async работает только на внешних скриптах. Скрипты с ним запускаются сразу по готовности, они не ждут другие скрипты или HTML-документ.

Для модулей атрибут async работает на любых скриптах.
Например, в скрипте ниже есть async, поэтому он выполнится сразу после загрузки, не ожидая других скриптов.
Скрипт выполнит импорт (загрузит ./analytics.js) и сразу запустится, когда будет готов, даже если HTML документ ещё не будет загружен, или если другие скрипты ещё загружаются.
Это очень полезно, когда модуль ни с чем не связан, например для счётчиков, рекламы, обработчиков событий.

```
<!-- загружаются зависимости (analytics.js) и скрипт запускается -->
<!-- модуль не ожидает загрузки документа или других тэгов <script> -->
<script async type="module">
  import {counter} from './analytics.js';

  counter.count();
</script>
```

Внешние скрипты
Внешние скрипты с атрибутом type="module" имеют два отличия:
1. Внешние скрипты с одинаковым атрибутом src запускаются только один раз:

```
<!-- скрипт my.js загрузится и будет выполнен только один раз -->
<script type="module" src="my.js"></script>
<script type="module" src="my.js"></script>
```

2. Внешний скрипт, который загружается с другого домена, требует указания заголовков CORS. Другими словами, если модульный скрипт загружается с другого домена, то удалённый сервер должен установить заголовок Access-Control-Allow-Origin означающий, что загрузка скрипта разрешена.
Это обеспечивает лучшую безопасность по умолчанию.

```
<!-- another-site.com должен указать заголовок Access-Control-Allow-Origin -->
<!-- иначе, скрипт не выполнится -->
<script type="module" src="http://another-site.com/their.js"></script>
```

Не допускаются «голые» модули
В браузере import должен содержать относительный или абсолютный путь к модулю. Модули без пути называются «голыми» (bare). Они не разрешены в import.
Например, этот import неправильный:

```
import {sayHi} from 'sayHi';

// Ошибка, "голый" модуль
// путь должен быть, например './sayHi.js' или абсолютный
```

Другие окружения, например Node.js, допускают использование «голых» модулей, без путей, так как в них есть свои правила, как работать с такими модулями и где их искать. Но браузеры пока не поддерживают «голые» модули.

Совместимость, «nomodule»
Старые браузеры не понимают атрибут type="module". Скрипты с неизвестным атрибутом type просто игнорируются. Мы можем сделать для них «резервный» скрипт при помощи атрибута nomodule:

```
<script type="module">
  alert("Работает в современных браузерах");
</script>

<script nomodule>
  alert("Современные браузеры понимают оба атрибута - и type=module, и nomodule, поэтому
пропускают этот тег script")
  alert("Старые браузеры игнорируют скрипты с неизвестным атрибутом type=module, но выполняют
этот.");
</script>
```

Инструменты сборки

В реальной жизни модули в браузерах редко используются в «сыром» виде. Обычно, мы объединяем модули вместе, используя специальный инструмент, например Webpack и после выкладываем код на рабочий сервер. Одно из преимуществ использования сборщика – он предоставляет больший контроль над тем, как модули ищутся, позволяет использовать «голые» модули и многое другое «своё», например CSS/HTML-модули.

Сборщик делает следующее:
1. Берёт «основной» модуль, который мы собираемся поместить в <script type="module"> в HTML.
2. Анализирует зависимости (импорты, импорты импортов и так далее)
3. Собирает один файл со всеми модулями (или несколько файлов, это можно настроить), перезаписывает встроенный import функцией импорта от сборщика, чтобы всё работало. «Специальные» типы модулей, такие как HTML/CSS тоже поддерживаются.

В процессе могут происходить и другие трансформации и оптимизации кода:
- Недостижимый код удаляется.
- Неиспользуемые экспорты удаляются («tree-shaking»).
- Специфические операторы для разработки, такие как console и debugger, удаляются.
- Современный синтаксис JavaScript также может быть трансформирован в предыдущий стандарт, с похожей функциональностью, например, с помощью Babel.
- Полученный файл можно минимизировать (удалить пробелы, заменить названия переменных на более короткие и т.д.).

Если мы используем инструменты сборки, то они объединяют модули вместе в один или несколько файлов, и заменяют import/export на свои вызовы. Поэтому итоговую сборку можно подключать и без атрибута type="module", как обычный скрипт:

```
<!-- Предположим, что мы собрали bundle.js, используя например утилиту Webpack -->
<script src="bundle.js"></script>
```

Итого

Подводя итог, основные понятия:

1. Модуль – это файл. Чтобы работал import/export, нужно для браузеров указывать атрибут <script type="module">. У модулей есть ряд особенностей:
    - Отложенное (deferred) выполнение по умолчанию.
    - Атрибут async работает во встроенных скриптах.
    - Для загрузки внешних модулей с другого источника, он должен ставить заголовки CORS.
    - Дублирующиеся внешние скрипты игнорируются.
2. У модулей есть своя область видимости, обмениваться функциональностью можно через import/export.
3. В модулях всегда включена директива use strict.
4. Код в модулях выполняется только один раз. Экспортируемая функциональность создаётся один раз и передаётся всем импортёрам.

Когда мы используем модули, каждый модуль реализует свою функциональность и экспортирует её. Затем мы используем import, чтобы напрямую импортировать её туда, куда необходимо. Браузер загружает и анализирует скрипты автоматически.
В реальной жизни часто используется сборщик Webpack, чтобы объединить модули: для производительности и других «плюшек».

Экспорт и импорт

Документация по импорту здесь.
Документация по экспорту здесь.

Директивы экспорт и импорт имеют несколько вариантов вызова.

Экспорт при объявлении
Можно пометить любое объявление как экспортируемое, разместив export перед ним, будь то переменная, функция или класс. Например, все следующие экспорты допустимы:

```
// экспорт массива
export let months = ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

// экспорт константы
export const MODULES_BECAME_STANDARD_YEAR = 2015;

// экспорт класса
export class User {
  constructor(name) {
    this.name = name;
  }
}
```

Экспорт отдельно от объявления

Также можно написать export отдельно: сначала объявить, а затем экспортировать.
Можно располагать и export выше функций.

```
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

function sayBye(user) {
  alert(`Bye, ${user}!`);
}

export {sayHi, sayBye}; // список экспортируемых переменных
```

Импорт отдельных элементов { }
список того, что нужно импортировать, перечисляется в фигурных скобках import {...}
вот так:

```
import {sayHi, sayBye} from './say.js';

sayHi('John');  // Hello, John!
sayBye('John'); // Bye, John!
```

Импорт всего сразу *
Можно импортировать всё сразу **в виде объекта**, используя import * as <obj>
вот так:

```
import * as say from './say.js';

say.sayHi('John');
say.sayBye('John');
```

В учебнике после этого примера пишется о том, что так делать не круто и приводятся доказательства. См. учебник, если понадобится.

Импорт «как» as

Можно использовать «as», чтобы импортировать под другими именами.
Например, для краткости импортируем sayHi в локальную переменную hi, а sayBye импортируем как bye:

```
import {sayHi as hi, sayBye as bye} from './say.js';

hi('John');  // Hello, John!
bye('John'); // Bye, John!
```

Экспортировать «как»

Аналогичный синтаксис «as» существует и для export.

```
// say.js
...
export {sayHi as hi, sayBye as bye};


// main.js
import * as say from './say.js';

say.hi('John');  // Hello, John!
say.bye('John'); // Bye, John!
```

Теперь hi и bye – официальные имена для внешнего кода, их нужно использовать при импорте.

Экспорт по умолчанию

На практике модули встречаются в основном одного из двух типов:
1. Модуль, содержащий библиотеку или набор функций, как say.js выше.
2. Модуль, который объявляет что-то одно, например модуль user.js экспортирует только class User.

Удобнее второй подход, когда каждая «вещь» находится в своём собственном модуле. Потребуется много файлов, но это не проблема: навигация по проекту становится проще, особенно, если у файлов хорошие имена, и они структурированы по папкам.
Модули предоставляют специальный синтаксис export default («экспорт по умолчанию») для второго подхода.

Ставим export default перед тем, что нужно экспортировать и потом импортируем без фигурных скобок. В файле может быть только один export default:

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


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

new User('John');
```

Запомним: фигурные скобки необходимы в случае именованных экспортов, для export default они не нужны.

Именованный экспорт	Экспорт по умолчанию
`export class User {...}`	`export default class User {...}`
`import {User} from ...`	`import User from ...`

Технически в одном модуле может быть как экспорт по умолчанию, так и именованные экспорты, но на практике обычно их не смешивают.

Так как в файле может быть максимум один export default, то экспортируемая сущность не обязана иметь имя. Это нормально, потому что может быть только один export default на файл, так что import без фигурных скобок всегда знает, что импортировать. Без default такой экспорт выдал бы ошибку

Примеры снизу – это корректные экспорты по умолчанию:

```
export default class { // у класса нет имени
  constructor() { ... }
}

export default function(user) { // у функции нет имени
  alert(`Hello, ${user}!`);
}

// экспортируем, не создавая переменную
export default ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
```

Имя «default»

В некоторых ситуациях для обозначения экспорта по умолчанию в качестве имени используется default. Например, чтобы экспортировать функцию отдельно от её объявления:

```
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

// то же самое, как если бы мы добавили "export default" перед функцией
export {sayHi as default};
```

Или, ещё ситуация, давайте представим следующее: модуль user.js экспортирует одну сущность «по умолчанию» и несколько именованных (редкий, но возможный случай):

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

export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

Вот как импортировать экспорт по умолчанию вместе с именованным экспортом:

```
// main.js
```

```
import {default as User, sayHi} from './user.js';

new User('John');
```

И если мы импортируем всё как объект import *, тогда его свойство default – как раз и будет экспортом по умолчанию:

```
// main.js
import * as user from './user.js';

let User = user.default; // экспорт по умолчанию
new User('John');
```

Довод против экспортов по умолчанию

Именованные экспорты «включают в себя» своё имя. Эта информация является частью модуля, говорит нам, что именно экспортируется. Именованные экспорты вынуждают нас использовать правильное имя при импорте, в то время как для экспорта по умолчанию мы выбираем любое имя при импорте. Так что члены команды могут использовать разные имена для импорта одной и той же вещи, и это не очень хорошо.

Обычно, чтобы избежать этого и соблюсти единообразие кода, есть правило: имена импортируемых переменных должны соответствовать именам файлов:

```
import User from './user.js';
import LoginForm from './loginForm.js';
import func from '/path/to/func.js';
```

Реэкспорт

Синтаксис «реэкспорта» export ... from ... позволяет импортировать что-то и тут же экспортировать, возможно под другим именем, вот так:

```
export {sayHi} from './say.js';
// реэкспортировать sayHi

export {default as User} from './user.js';
// реэкспортировать default
```

</>

GIT

Добавление репозитория с Github

```
git init
git remote add origin https://github.com/Join-Red-Army/mogo-landing.git
git add .
git commit -am 'added images for the page'

git clone 'https'
```

Откатиться к последнему коммиту

команда безвозвратно удаляет несохраненные текущие изменения из рабочей области и из индекса

```
git reset --hard HEAD
```

Удалить коммит из github

```
git reset --hard a3775a5485af    хэш-код коммита, к которому хотим вернуться
git push --force                 отсылаем эту правку на гитхаб
```

Получить изменения с сервера

```
git pull
```

Узнать текущую ветку

```
git branch
```

Переключиться на конкретный коммит

```
git checkout [номер]
```

```

```

```

```

```

```

</>

SASS

Документация: [sass-lang]
Это метаязык на основе CSS, предназначенный для увеличения уровня абстракции CSS-кода и упрощения файлов каскадных таблиц стилей.

```
sass --watch app/sass:public/stylesheets
```

Копирование названий селекторов
&:hover
Не нашёл это в документации. & указывает на селектор выше по вложенности, поэтому вместо btn:hover можно прописать &:hover.

Переменные

В переменных хранятся любые css-значения. Создаются и вставляются через символ $.

Вложеннось
SASS позволяет вкладывать селекторы, как это делается в html.

Фрагментирование
Можно создавать файлы с фрагментами CSS, которые подключаются к другим sass.
Такие файлы создаются с начальным подчёркиванием: _partial.scss.
Это сообщить компилятору, что файл не неадо генерировать в css.
Фрагменты используются с правилом @use.

Модули
Не обязательно хранить весь sass в одном файле. Можно ссылаться на mixins и functions.
Модули будут компилироваться в итоговый css-файл.
Расширение файла не указывается.
Модуль подключается через `@use 'module_name'`

Mixins
Миксин нужен для создания сразу нескольких строк свойств CSS, которые можно все вместе куда-то вставлять.
В миксины также можно передавать значения извне, назначив переменную, которая будет использоваться внутри.

```
@mixin name
Создание директивы и присвоение ей имени.
@include name
Использование миксина
```

Extend/Inheritance
@extend позволяет передавать набор свойств из одного селектора в другой.
A placeholder class – специальный тип класса, который печатает только при расширении.
Ничего не понял, см. документацию.

```
Документация: sass-lang

Установка
npm install -g sass

Использование
sass input.scss output.css
компилировать sass файл в css

--watch
sass --watch input.scss output.css
следить за указанными файлами или папками.

sass --watch app/sass:public/stylesheets
следить за всеми файлами в папке app/sass и компилировать их в папку public/stylesheets


Переменные
$  символ для создания и вставки переменной

Создание
$font-stack: Helvetica, sans-serif;
$primary-color: #333;

Использование
body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

Копирование названий селекторов

```
&:hover

.btn {
  background: …
  color: …

  &:hover {
    background-color: …
  }
}
```

Вложенность

```
nav {
  ul {
    margin: 0;
    list-style: none;
  }
  li { display: inline-block; }
  a {
    padding: 6px 12px;
  }
}
```

Модули

```
_base.scss
body {
  font: 100% $font-stack;
  color: $primary-color;
}

styles.scss
@use 'base';
Body будет вставлен, как если бы он был написан тут.
Далее пишешь обычный код
```

Mixins

```
@mixin name
Создание директивы и присвоение ей имени.
@include name
Использование миксина

($theme)
В скобках указывается имя переменной и её значение по умолчанию.
При использовании миксина, этой переменной можно передать другое значение.

Объявление
@mixin theme($theme: DarkGray) {
  background: $theme;
  box-shadow: 0 0 1px rgba($theme, .25);
}

Использование
.info {
  @include theme;
.success {
  @include theme($theme: red);
}
```

</>

Gulp

Gulp — это таск-менеджер для автоматического выполнения часто используемых задач (например, минификации, тестирования, объединения файлов).

API в документации здесь.

Установка

```
Установка
npm install --global gulp-cli

npx mkdirp my-project
cd my-project

npm init
npm install --save-dev gulp


Create a gulpfile
function defaultTask(cb) {
  cb();
}

подключение плагинов
const autoprefixer = require("gulp-autoprefixer");

консоль
gulp
gulp <task> <othertask>
```

gulpfile.js

Файл gulp.js автоматом запускается при команде gulp.
Внутри этого файла могут быть API типа src(), dest(), series(), parallel().
Кроме API, можно импортировать обычные JS или Node модули.
Любые экспортированные функции будут зарегистрированы в системе задач gulp.

Транспиляция
Новые версии ноды поддерживают большинство функций TypeScript или Babel, кроме синтакса import-export.
gulpfile transpilation документация здесь.

Разделение файлов
Если тасков будет много, каждая таска может быть описана в отдельном файле, затем импортирована в основной. Это также позволит тестить каждую таску независимо.
Можно поместить файл gulpfile.js в папку, которая тоже называется gulpfile.js. Сам файл при этом должен быть назван index.js. Здесь же могут быть модули для тасков.
Если используется трансплаер, надо назвать папку и файл соответственно.

Создание тасков

Каждая таска – это асинхронная функция, которая принимает первый коллбэк-ошибку или возвращает промис, stream, event emitter, child process, or observable.
Синхронные таски не поддерживаются.

Экспорт
Таски могут быть публичными или приватными.
Публичные – экспортируются из gulpfile и могут быть запущены через команду gulp.
Приватные – используются только внутри gulpfile, обычно в апи series() и parallel(). В остальном используются так же, как и публичные, просто не могут быть запущенными из консоли пользователем.

Чтобы сделать таску публичной, её надо эспортировать из gulpfile:

```
const { series } = require('gulp');

clean – приватная функция, используется в series()
function clean(cb) {
  cb();
}

build – публичная функция, используется в series()
function build(cb) {
  cb();
}

exports.build = build;
exports.default = series(clean, build);
```

будут запущены и выполнены последовательно: default (clean, build).

Составление тасков
Gulp позволяет объединять индивидуальные таски в одну большую операцию с помощью методов series() и parallel(). Эти методы принимают любое кол-во тасков, а также могут быть вложены друг в друга.

series()
Позволяет выполнять таски строго последовательно. Сначала импортируешь, затем используешь:

```
// сначала импротируешь
const { series } = require('gulp');

function transpile(cb) {
  cb();
}
function bundle(cb) {
  cb();
}

// затем экспортируешь и используешь
exports.build = series(transpile, bundle);
```

parallel()
Ну, соответственно

```
const { parallel } = require('gulp');

function javascript(cb) {
  cb();
}
function css(cb) {
  cb();
}

exports.build = parallel(javascript, css);
```

Можно комбинировать таски и запускать под разными условиями:

```
const { series, parallel } = require('gulp');


if (process.env.NODE_ENV === 'production') {
  exports.build = series(transpile, minify);
} else {
  exports.build = series(transpile, livereload);
}


exports.build = series(
  clean,
  parallel(
    cssTranspile,
    series(jsTranspile, jsBundle)
  ),
  parallel(cssMinify, jsMinify),
  publish
);
```

Async Completion

Пропустил. Тут говорится про возврат асинхронных функций из гальпа.

Работа с файлами

Методы src() и dest() предоставляются для работы с файлами.
Обычно разные плагины размещаются между ними и с помощью метода .pipe() передаются в потоке.
Тем не менее, их можно размещать в разных местах потока, не только в начале и конце.

src() может работать в нескольких режимах:
- Buffering по умолчанию. Загружает файлы в память.
- Streaming рабтает с большими файлами типа огромных изображений или видеофайлов. Контент обрабатывается потоком в виде маленьких чанков и хранится не в памяти, а на жёстком диске.
- Empty не содержит контента и полезен когда обрабатывается мета информация.

```
src()
получает glob (адрес в формате строки) и считывает файлы для передачи в Node stream.
Этот поток должен быть возвращён из таска.
Импорт: const { src } = require('gulp');

dest()
Получает адрес директории для вывода файлов (в формате строки).
Также создаёт Node stream.
Когда он получает файл, прошедший через pipeline, он записывает результат в указанную
директорию.
Испорт: const { dest } = require('gulp');

symlink()
Работает как dest, но но создаёт ссылки вместо файлов.

.pipe()
Для передачи потока из одного плагина в другой.
Размещается в пайплайне.
```

```
const { src, dest } = require('gulp');
```

```
exports.default = function() {
  return src('src/*.js')
    .pipe(dest('output/'));
}


const { src, dest } = require('gulp');
const babel = require('gulp-babel');

exports.default = function() {
  return src('src/*.js')
    .pipe(babel())
    .pipe(dest('output/'));
}


dest также может содержаться внутри .pipe
exports.default = function() {
  return src('src/*.js')
    .pipe(babel())
    .pipe(src('vendor/*.js'))
    .pipe(dest('output/'))
    .pipe(uglify())
    .pipe(rename({ extname: '.min.js' }))
    .pipe(dest('output/'));
}
```

Explaining Globs

Глоб – это строка (или literal или wildcard characters), которые используются для указания путей к файлам.

Метод src() ожидает glob или массив globs чтобы определить, какой файл пускать в пайплайн. Как минимум, один глоб должен вести к файлам для работы, чтобы src() не выдал ошибку.

Micromatch Documentation
node-glob's Glob Primer
Begin's Globbing Documentation
Wikipedia's Glob Page

```
Сепаратор: /
Экран: //
Сегмент: всё, что между сепараторами.

Нельзя использовать path.join


*
Означает любое кол-во (в т.ч. ни одного) символов в одном сегменте. Обычно используетяся для
добавления всех файлов из одного каталога.
'*.js' – выбрать все файлы в каталоге с расширением .js

**
Означает все файлы, в т.ч. во вложенных каталогах.
'scripts/**/*.js' – выбрать все файлы с расширением .js в текущем и во вложенных подкаталогах.


!
Символ отрицания. Обязательно должен располагаться после «положительного» символа. Логика:
положительный сначала ищет массив результатов, а ! удаляет лишние из него.
Чтобы исключить все файлы в каталоге, надо добавить /** после имени каталога.

['scripts/**/*.js', '!scripts/vendor/**']
```

```
Если любые неотрицательные глобусы следуют за отрицательным, ничего не будет удалено из более позднего набора совпадений.

Отрицание можно использовать для ограничения поиска по двойным звёздам **
['**/*.js', '!node_modules/**']
Все js файлы в подпапках, кроме любых файлов в папке node_modules.


#
Перекрывающиеся файлы (коллизии). Когда несколько глобов укзывают на одни и те же файлы
Когда в одном src () используются перекрывающиеся глобусы, gulp делает все возможное, чтобы удалить дубликаты, но не пытается дедуплицировать отдельные вызовы src ().
```

Using Plugins

Гальповские плагины – это [Node Transform Streams](), которые трансформирубт файлы в пайплайн. Они могут менять наименование файла, мета данные или контент каждого файла, который проходи через них.
Плагины из NPM можно найт с приставкой "gulpplugin" и "gulpfriendly".

Их можно объединять, чтобы получить желаемый результат.

```
const { src, dest } = require('gulp');
const uglify = require('gulp-uglify');
const rename = require('gulp-rename');

exports.default = function() {
  return src('src/*.js')
    // The gulp-uglify plugin won't update the filename
    .pipe(uglify())
    // So use gulp-rename to change the extension
    .pipe(rename({ extname: '.min.js' }))
    .pipe(dest('output/'));
}
```

Conditional plugins
Плагины не учитывают тип передаваемых в них файлов. Поэтому надо использовать gulp-if чтобы преобразовывать подмножество файлов.

```
const { src, dest } = require('gulp');
const gulpif = require('gulp-if');
const uglify = require('gulp-uglify');

function isJavaScript(file) {
  // Check if file extension is '.js'
  return file.extname === '.js';
}

exports.default = function() {
  // Include JavaScript and CSS files in a single pipeline
  return src(['src/*.js', 'src/*.css'])
    // Only apply gulp-uglify plugin to JavaScript files
    .pipe(gulpif(isJavaScript, uglify()))
    .pipe(dest('output/'));
}
```

Watching Files

Дохуя всего.

Валак

```
Установка
npm install --global gulp-cli

перейти в директорию с проектом
npx mkdirp my-project
cd my-project

Необходимо создать пакет с json-файлом и зависимостями
npm init
package.json {
  "name": "gulp-web",
  "version": "1.0.0",
  "description": "test using gulp",
  "author": "Red Army"
}

установить gulp как зависимость в проект
npm install --save-dev gulp


Создать файл и в нём написать:
gulpfile.js
const {src, dest} = require("gulp");
const gulp = require("gulp");
src – где читать файлы-исходники
dest – куда экспортировать обработанные файлы


Поставить дополнения
gulp-autoprefixer
npm install --save-dev gulp-autoprefixer

gulp-cssbeautify
npm install --save-dev gulp-cssbeautify
Красивое форматирование css-файла на выходе

gulp-strip-css-comments
npm install --save-dev gulp-strip-css-comments
для удаления комментариев из css-файла при создании минифицированной версии

gulp-rename
```

```
npm install --save-dev gulp-rename
для изменения названия файлов (min и не min файлы)

gulp-sass
npm install sass gulp-sass --save-dev
компилятор sass в css

gulp-cssnano
npm install gulp-cssnano --save-dev
для сжатия css-файлов (минификатор)

gulp-rigger
npm install --save-dev gulp-rigger
для склеивания разных js-файлов (компонентов) в один

gulp-uglify
npm install --save-dev gulp-uglify
отвечает за сжатие (минификацию) js-файлов

gulp-plumber
npm install --save-dev gulp-plumber
не будут слетать гальповские таски, если есть ошибки

gulp-imagemin
npm install --save-dev gulp-imagemin
сжатие и оптимизация изображений

del
npm install --save-dev del
будет очищать папку с проектом для свежих файлов

panini
npm install --save-dev panini
работа с HTML, создание шаблонов, фрагментов кода (темплейтов), и подключать их везде.

browser-sync
npm install --save-dev browser-sync
локальный сервер для работы в прямом эфире


подключить эти плагины в gulp-файле
const {src, dest} = require("gulp");
const gulp = require("gulp");

const del = require(del);
const autoprefixer = require("gulp-autoprefixer");
const cssbeautify = require("gulp-cssbeautify");
const cssnano = require("gulp-cssnano");
const imagemin = require("gulp-imagemin");
const plumber = require("gulp-plumber");
const rename = require("gulp-rename");
const rigger = require("gulp-rigger");
const sass = require("gulp-sass");
const stripCssComments = require("gulp-strip-css-comments");
const uglify = require("gulp-uglify");
const panini = require("panini");
const sass = require("sass");
const browsersync = require("browser-sync").create();


Структура проекта

dist          хранятся скомпилированные файлы

src           все html файлы + рабочие файлы
  assets      все необходимые файлы
```

```
    sass
      blocks
      style.scss
    js
      components
      app.js
    img
```

Прописать в гальпе пути
к файлам, где он должен брать исходники и перемещать в готовом виде. Пишется под импортами

```
var path = {

  build: { // это какие файлы куда копировать после обработки
    html: "dist/",
    js: "dist/assets/js/",
    css: "dist/assets/css/",
    images: "dist/assets/img"
  },

  src: { // пути для исходников
    html: "src/*.html",
    js: "src/assets/js/*.js",
    css: "src/assets/sass/style.scss",
    images: "src/assets/img/**/*.{jpg, png, svg, gif, ico}"
  },

  watch: { // за какими файлами наблюдать
    html: "src/**/*.html",
    js: "src/assets/js/**/*.js",
    css: "src/assets/sass/**/*.scss",
    images: "src/assets/img/**/*.{jpg, png, svg, gif, ico}"
  },

  // папка, в которой будут храниться скомпилированные файлы
  // она будет очищаться перед загрузкой в неё н овых файлов
  clean: "./dist"

}
```

gulpfile.js от Валака

```
const {src, dest} = require("gulp");
const gulp = require("gulp");

const del = require("del");
const autoprefixer = require("gulp-autoprefixer");
const cssbeautify = require("gulp-cssbeautify");
const cssnano = require("gulp-cssnano");
const plumber = require("gulp-plumber");
const rename = require("gulp-rename");
const rigger = require("gulp-rigger");
var sass = require('gulp-sass')(require('sass'));
const stripCssComments = require("gulp-strip-css-comments");
const uglify = require("gulp-uglify");
const panini = require("panini");
const browsersync = require("browser-sync").create();
const removeComments = require("gulp-strip-css-comments");

// const sass = require("sass");
// import imagemin from "gulp-imagemin"; ошибка SyntaxError: Cannot use import statement outside a module

var path = {
  build: { // это какие файлы куда копировать после обработки
    html: "dist/",
    js: "dist/assets/js/",
    css: "dist/assets/css/",
    images: "dist/assets/img"
  },
```

```
  src: { // пути для исходников
    html: "src/*.html",
    js: "src/assets/js/*.js",
    css: "src/assets/sass/style.scss",
    images: "src/assets/img/**/*.{jpg, png, svg, gif, ico}"
  },

  watch: { // за какими файлами наблюдать
    html: "src/**/*.html",
    js: "src/assets/js/**/*.js",
    css: "src/assets/sass/**/*.scss",
    images: "src/assets/img/**/*.{jpg, png, svg, gif, ico}"
  },

  // папка, в которой будут храниться скомпилированные файлы
  // она будет очищаться перед загрузкой в неё н овых файлов
  clean: "./dist"
};


// таски
function browserSync(done) {
  browsersync.init({
      server: {
          baseDir: "./dist/"
      },
      port: 3000
  });
}

function browserSyncReload(done) {
  browsersync.reload();
}

function html() {
  return src(path.src.html, {base: "src/"})
    .pipe(plumber())
    .pipe(dest(path.build.html));
}

function css() {
  return src(path.src.css, {base: "src/assets/sass/"})
    .pipe(plumber())
    .pipe(sass())
    .pipe(autoprefixer({
      overrideBrowserslist: ['last 8 versions'],
      cascade: true
    }))
    .pipe(cssbeautify())
    .pipe(dest(path.build.css))
    .pipe(cssnano({
      zindex: false,
      discardComments: {
        removeAll: true
      }
    }))
    .pipe(removeComments())
    .pipe(rename({
      suffix: ".min",
      extname: ".css"
    }))
    .pipe(dest(path.build.css))
}

function js() {
  return src(path.src.js, {base: './src/assets/js/'})
      .pipe(plumber())
      .pipe(rigger())
      .pipe(gulp.dest(path.build.js))
      .pipe(uglify())
      .pipe(rename({
          suffix: ".min",
          extname: ".js"
      }))
      .pipe(dest(path.build.js))
      .pipe(browsersync.stream());
}



function images() {
  return src(path.src.images)
      // .pipe(imagemin())
```

```
      .pipe(dest(path.build.images));
}

function clean() {
  return del(path.clean);
}

function watchFiles() {
  gulp.watch([path.watch.html], html);
  gulp.watch([path.watch.css], css);
  gulp.watch([path.watch.js], js);
  gulp.watch([path.watch.images], images);
}

const build = gulp.series(clean, gulp.parallel(html, css, js, images));
const watch = gulp.parallel(build, watchFiles, browserSync);


/* Exports Tasks */
exports.html = html;
exports.css = css;
exports.js = js;
exports.images = images;
exports.clean = clean;
exports.build = build;
exports.watch = watch;
exports.default = watch;
```

</>

Babel

Установка и запуск

В пустой папке инициализировать новый проект:

```
npm init
npm init -y  инициализирует проект без дополнительных вопросов
```

Установка Babel

```
npm install --save-dev @babel/core @babel/cli

--save-dev
Означает, что те зависимости, которые сейчас будут установлены, устанавливаются исключительно
для целей разработки, а не для работы приложения.

@babel/core
Называется name space, или пространство имён. Это позволяет структурировать те пакеты, которые
разрабатывает одна организация. Babel, в данном случае, представлен двумя приложениями. В итоге
это будут самые обыкновенные пакеты.
```

Файл package.json будет обновлён. Появится блок «devDependencies», в него запишутся установленные пакеты.

Теперь в консоли можно использовать **npx**.
Npx – это утилита для запуска в качестве скрипта один из установленных пакетов, т.е. запустить пакет как обычное node.js приложение.

```
npx babel src --out-dir build

Первый параметр – откуда надо брать исходные файлы: src
Второй параметр – куда складывать трансформированные файлы: build
```

Результат – сообщение о том, что файл успешно преобразован. В папке будут те же файлы, которые лежат в папке src, но преобразованные. Без дополнительных модулей Babel не трансформирует код, поэтому файлы останутся без изменений.

Плагины

Babel – это модульный компилятор.
Для каждого преобразования, для каждой конструкции языка нужно установить отдельный плагин.
Для того, чтобы он начал преображать код, ему надо указать, какие именно аспекты языка надо преобразовывать.
Есть список плагинов на сайте Babel здесь.

Babel-плагины – это самые обычные npm-пакеты и устанавливаются как всегда через npm.
После установки плагинов можно запустить трансплаинг, но обязательно с флагом --plugins, после которого перечислить те плагины, которые будут использоваться.
Важно! Перечисляются через запятую без пробелов.

```
Установить плагин как обычный пакет
npm install --save-dev @babel/наименование

Запустить и перечислить используемые плагины
npx babel src --out-dir build --plugins @babel/наименование, @babel/наименование
```

Пример:

```
@babel/plugin-transform-template-literals  заменяет конструкцию с бэктиками на обычные кавычки
@babel/plugin-transform-classes            классы преобразуются в обычные функции
@babel/plugin-transform-block-scoping      вместо const используется var


npm install --save-dev @babel/plugin-transform-template-literals @babel/plugin-transform-
classes @babel/plugin-transform-block-scoping

npx babel src --out-dir build --plugins @babel/plugin-transform-template-literals,
@babel/plugin-transform-classes, @babel/plugin-transform-block-scoping
```

Конфигурация .babelrc

В предыдущем виде, команда для запуска Babel становится громоздкой и её неудобно использовать, потому что надо перечислять все плагины в командной строке.
Конфигурационный файлы – простой механизм Babel, чтобы сразу указать конфигурацию для компилятора.
Документация здесь.

Babel поддерживает .json или .js файлы для конфигурации. Первые распространены больше.
Имя файла начинается с точки.
Внутри этого файла размещают обычный json объект с параметрами конфигурации.
Используется синтаксис json: двойные кавычки, нет висячих запятых, ключи берутся в кавычки и т.д.

```
.babelrc

{
  "plugins": [
    "@babel/plugin-transform-template-literals",
    "@babel/plugin-transform-classes",
    "@babel/plugin-transform-block-scoping"
  ],
```

```
  "presets": ["@babel/preset-env" (или просто "@babel/env")]
}

{ "presets": [ [пресет, {targets: {браузрер: версия} }] ] }


plugins   массив плагинов для использования, чтобы не перечислять их в командной строке
presets   массив пресетов

запуск
npx babel src --out-dir build
```

Babel Presets

Совершенно очевидно, что вручную устанавливать, хранить и поддерживать список необходимых плагинов неудобно. Чтобы упростить этот процесс, есть специальный механизм: presets.

Это просто заранее сконфигурированный список плагинов, который можно передать в Babel, чтобы не перечислять плагины по одному вручную.

@babel/preset-env
Этот пресет содержит плагины, для того чтобы поддерживать самый свежий стандарт ecma script. Он всегда отражает текущую стандартную версию языка. Экспериментальные возможности, которые пока не стали стандартом, в этом пресете не учитываются, их надо устанавливать отдельно.

```
npm install --save-dev @babel/preset-env
```

См. также конфигурацию.

Сборки под браузеры

Не всегда надо трансформировать все аспекты языка, если приложение разрабатывается под какой-то определённый браузер. Намного лучше трансформировать только тот код, который целевые браузеры не смогут понять.
Чтобы определить, какие именно трансформации нужны (например, поддерживает ли браузер async функции), можно использовать **can I use** здесь. Тем не менее, это очень трудозатратно.

Одна из возможностей пресета env – поддержка конкретных версий браузеров.
В .babelrc можно указать дополнительные опции: в массив пресетов передаётся ещё один массив. В нём первый элемент – сам пресет, а второй – объект с настройками для этого пресета.

```
{ "presets": [ [пресет, {targets: {браузрер: версия} }] ] }


{
  "presets": [
    ["@babel/env", { "targets": {"edge": "18", "chrome": "74"} }]
  ]
}
```

Динамический выбор браузеров, browserslist

Кроме указания на конкретные браузеры, можно указать выражение, которое будет выбирать браузеры по каким-то признакам.

Для этого надо изменить синтаксис targets, теперь это будет массив со строкой, а не объект:

```
Последние 2 версии Хрома и Фокса:
"presets": [ ["@babel/env", { "targets": ["last 2 chrome versions"] }] ]
```

```
Комбинации:
{ "targets": ["last 2 chrome versions", "last 2 firefox versions", "last 2 ios versions"] }
```

Можно вывести в консоль во время сборки с помощью debug, под какие именно браузеры перегоняется код:

```
{
  "presets": [[
    "@babel/env",
    {
      "targets": ["last 2 chrome versions", "last 2 firefox versions", "last 2 ios versions"],
      "debug": true
    }
  ]],
  "plugins": ["@babel/plugin-proposal-class-properties"]
}
```

Чтобы проверить, под какие именно браузеры бабель будет писать код, можно вбить target на сайте (или приложение) browserlist здесь.

Ещё:

```
"targets": ["> 0.3%"]
Все браузеры, у которых пользователей в мире больше, чем 0,3%

"targets": ["> 0.3%", "not ie > 0"]
Все браузеры, кроме ie
```

Файлы конфигурации browserslist

```

```

</>

TypeScript

Файлы создаются с расширением *.ts

Установка

TypeScript compiler

```
установка
npm install typescript

компиляция в js
npx tsc index.ts

создать конфигурационный файл tsconfig.json
npx tsc --init
```

Настройки комплаера

Документация по настройке комплаера тут.
Настройки хранятся в файле tsconfig.json

```
noEmitOnError      не создавать js файл, если есть ошибки в коде
outDir             указать output-папку

target             в какой стандарт ES перегонять output

noImplicitAny      нелья оставлять переменные без указания их типа
strictNullChecks   необходимо явно обрабатывать null и undefined
```

Типы

Явное указание типов

```
Примитивы
let value: string = 'hello'
let value: number = 123
let value: boolean = true
let value: any = …

Массивы
number[]      массив, состоящий только из цифр
string[]
boolean[]
any[]


Параметры и возврат из функций
function foo(person: string, date: Date)

function getFavoriteNumber(): number {
  …
  return 26;
}


Объекты
```

interface declaration:

```
interface User {
  name: string;
  id: number;
}

const user: User = {
  name: "Hayes",
  id: 0,
};



class UserAccount {
  name: string;
  id: number;
```

```
  constructor(name: string, id: number) {
    this.name = name;
    this.id = id;
  }
}

const user: User = new UserAccount("Murphy", 1);
```

```

```

</>

CSS для JavaScript

Единицы измерения: px, em, rem

```
px

em    относительно текущего шрифта
rem   относительно размера шрифта, указанного для элемента <html>

%     относительно размера родителя

vw    1% ширины окна
vh    1% высоты окна
vmin  наименьшее из (vw, vh), в IE9 обозначается vm
vmax  наибольшее из (vw, vh)
```

Пиксель px – это базовая единица измерения.
Количество пикселей задаётся в настройках разрешения экрана. Все значения браузер в итоге пересчитает в пиксели. Пиксели могут быть дробными: 16.5px. При окончательном отображении дробные пиксели округляются и становятся целыми.

1em – текущий размер шрифта.
Можно брать любые пропорции от текущего шрифта: 2em, 0.5em и т.п. Размеры в em – относительные, они определяются по текущему контексту. Если и у родителя, и у потомка размер 1.5em, то у потомка он будет больше, потому что принимает за единицу размер родителя.

rem
Задаёт размер относительно размера шрифта, указанного для элемента <html>.
Элементы, размер которых задан в rem, не зависят друг от друга и от контекста – и этим похожи на px, а с другой стороны они все заданы относительно размера шрифта <html>.
Единица rem не поддерживается в IE8-.

Проценты %
Относительные единицы. Как правило, процент будет от значения свойства родителя с тем же названием, но не всегда:
- при установке свойства margin-left, процент берётся от *ширины* родительского блока, а не от его margin-left.

- при установке свойства line-height в %, процент берётся от текущего *размера шрифта*, а вовсе не от line-height родителя.
- для width/height обычно процент от ширины/высоты родителя, но при position:fixed, процент берётся от ширины/высоты *окна*.

Относительно экрана: vw, vh, vmin, vmax
Эти значения были созданы, в первую очередь, для поддержки мобильных устройств. Их основное преимущество – в том, что любые размеры, которые в них заданы, автоматически масштабируются при изменении размеров окна.
vw – 1% ширины окна
vh – 1% высоты окна
vmin – наименьшее из (vw, vh), в IE9 обозначается vm
vmax – наибольшее из (vw, vh)

Что понимать под размером шрифта
Он обычно чуть больше, чем расстояние от верха самой большой буквы до низа самой маленькой. В эту высоту помещается любая буква. Но при этом «хвосты» букв, таких как p, g могут заходить за это значение, то есть вылезать снизу. Поэтому обычно высоту строки делают чуть больше, чем размер шрифта.

display

Свойство display (CSS) определяет тип отображения самого элемента, к которому применяется, и его дочерних элементов.
Бокс - это прямоугольная область, являющаяся изображением элемента.

Значений много, вот основные:

```
display: none;

display: block;
display: inline;
display: inline-block;

display: flex;
display: grid;
display: table;
```

none
Самое простое значение. Элемент не показывается, вообще. Как будто его и нет.

block
Блок стремится расшириться на всю доступную ширину. Можно указать ширину и высоту явно.
Блочные элементы располагаются один над другим, вертикально (если нет особых свойств позиционирования, например float). Блоки прилегают друг к другу вплотную, если у них нет margin.
Это значение многие элементы имеют по умолчанию: , заголовок <h1>, параграф <p>.

inline
Элементы располагаются на той же строке, последовательно.
Ширина и высота элемента определяются по содержимому. Поменять их нельзя.
Например, инлайновые элементы по умолчанию: , <a>.

inline-block
Означает элемент, который продолжает находиться в строке (inline), но при этом может иметь свойства блока.
Это значение используют, чтобы отобразить в одну строку блочные элементы, в том числе разных размеров.
Свойство `vertical-align` позволяет выровнять такие элементы внутри внешнего блока.

Как и инлайн-элемент:
- Располагается в строке.

- Размер устанавливается по содержимому.
Во всём остальном – это блок:
- Элемент всегда прямоугольный.
- Работают свойства width/height.

table-*
Современные браузеры (IE8+) позволяют описывать таблицу любыми элементами.
Для таблицы целиком table, для строки – table-row, для ячейки – table-cell и т.д.
С точки зрения современного CSS, обычные <table>, <tr>, <td> и т.д. – это просто элементы с предопределёнными значениями display.

Внутри ячеек table-cell свойство vertical-align выравнивает содержимое по вертикали. Это можно использовать для центрирования.

flex-box
Flexbox позволяет удобно управлять дочерними и родительскими элементами на странице, располагая их в необходимом порядке.

float
https://learn.javascript.ru/float

position
Позволяет сдвигать элемент со своего обычного места. Основные значения:

```
position: static
position: relative
position: absolute
position: fixed
```

position: static
производится по умолчанию, в том случае, если свойство position не указано. Такая запись встречается редко и используется для переопределения других значений position.

position: relative
Относительное позиционирование сдвигает элемент относительно его текущего положения в потоке.
Необходимо указать элементу CSS-свойство position: relative и координаты left/right/top/bottom.

position: absolute
Абсолютное позиционирование делает две вещи:
1. Элемент исчезает с того места, где он должен быть и позиционируется заново. Остальные элементы занимают его мето, будто этого элемента не было.
2. Координаты top/bottom/left/right для нового местоположения отсчитываются от ближайшего позиционированного родителя, т.е. родителя с позиционированием, отличным от static. Если такого родителя нет – то относительно документа.

Кроме того:
1. Ширина элемента с position: absolute устанавливается по содержимому.
2. Элемент получает display:block, который перекрывает почти все возможные display.
3. В абсолютно позиционированном элементе можно одновременно задавать противоположные границы left/right, top/bottom. Браузер растянет такой элемент до границ.

Важное отличие от relative: так как элемент удаляется со своего обычного места, то элементы под ним сдвигаются, занимая освободившееся пространство.
Иногда бывает нужно поменять элементу position на absolute, но так, чтобы элементы вокруг не сдвигались. Как правило, это делают, меняя соседей – добавляют margin/padding или вставляют в документ пустой элемент с такими же размерами.

position: fixed
Позиционирует объект точно так же, как absolute, но относительно window.
Когда страницу прокручивают, фиксированный элемент остаётся на своём месте и не прокручивается вместе со страницей.

Центрирование горизонтальное и вертикальное

Горизонтальное
text-align
Для центрирования инлайновых элементов – достаточно поставить родителю text-align: center
Для центрирования блока это уже не подойдёт, свойство просто не подействует.

margin: auto
Блок по горизонтали центрируется через margin: auto
Значение margin-left:auto/margin-right:auto заставляет браузер выделять под margin всё доступное сбоку пространство. А если и то и другое auto, то слева и справа будет одинаковый отступ, таким образом элемент окажется в середине.

Вертикальное
Вертикальное центрирование изначально не было предусмотрено в спецификации CSS и по сей день вызывает ряд проблем. Есть три основных решения.

position:absolute + margin
Центрируемый элемент позиционируем абсолютно и опускаем до середины по вертикали при помощи top:50% и смещения margin на половину ширины элемента.

Одна строка: line-height
Вертикально отцентрировать одну строку в элементе с известной высотой height можно, указав эту высоту в свойстве line-height. Это работает, но лишь до тех пор, пока строка одна, а если содержимое вдруг переносится на другую строку, то начинает выглядеть довольно уродливо.

Таблица с vertical-align
В таблицах свойство vertical-align указывает расположение содержимого ячейки. С vertical-align: middle содержимое находится по центру. Таким образом, можно обернуть нужный элемент в таблицу размера width:100%;height:100% с одной ячейкой, у которой указать vertical-align:middle, и он будет отцентрирован.

Но мы рассмотрим более красивый способ, который поддерживается во всех современных браузерах, и в IE8+. В них не обязательно делать таблицу, так как доступно значение display:table-cell. Для элемента с таким display используются те же алгоритмы вычисления ширины и центрирования, что и в TD. И, в том числе, работает vertical-align. Этот способ замечателен тем, что он не требует знания высоты элементов.

```
<div style="display: table-cell; vertical-align: middle; … >
```

Можно и в процентах, завернув «псевдоячейку» в элемент с display:table, которому и поставим ширину:

```
<div style="display: table; width: 100%">
  <div style="display: table-cell; vertical-align: middle; height: 100px; border: 1px solid
blue">
    <button>Кнопка<br>с любой высотой<br>и шириной</button>
  </div>
</div>
```

Центрирование в строке с vertical-align
Для инлайновых элементов (display:inline/inline-block), включая картинки, свойство vertical-align центрирует сам инлайн-элемент в окружающем его тексте.

```
vertical-align:
```

```
baseline     по умолчанию
middle       по середине
sub          вровень с <sub>
super        вровень с <sup>
text-top     верхняя граница вровень с текстом
text-bottom  нижняя граница вровень с текстом
```

Центрирование с vertical-align без таблиц
Какая-то ёбань, см. по ссылке.

С использованием модели flexbox

```
.outer {
    display: flex;
    justify-content: center; /*Центрирование по горизонтали*/
    align-items: center;     /*Центрирование по вертикали */
}
```

Свойства font-size и line-height

font-size – *размер шрифта*, в частности, определяющий высоту букв.
line-height – *высота строки*.

Размер шрифта обычно равен расстоянию от самой верхней границы букв до самой нижней, исключая «нижние хвосты» букв, таких как p, g. При размере строки, равном font-size, строка не будет размером точно «под букву». В зависимости от шрифта, «хвосты» букв при этом могут вылезать.

Обычно размер строки делают чуть больше, чем шрифт. По умолчанию в браузерах используется специальное значение line-height:normal. Как правило, оно будет в диапазоне 1.1 - 1.25, но стандарт не гарантирует этого, он говорит лишь, что оно должно быть «разумным» (reasonable).

Множитель для line-height
Значение line-height можно указать при помощи px или em, но гораздо лучше – задать его числом.
Значение-число интерпретируется как множитель относительно размера шрифта. Например, значение с множителем line-height: 2 при font-size: 16px будет аналогично line-height: 32px (=16px*2).

Установить font-size и line-height можно **одновременно**.
Соответствующий синтаксис выглядит так:

```
минимум
font: 20px/1.5 Arial,sans-serif;

максимум
font: italic bold 20px/1.5 Arial,sans-serif;
```

Свойство white-space

Свойство white-space
управляет тем, как обрабатываются пробельные символы внутри элемента.

white-space на MDN

```
white-space: normal;   несколько пробелов объединяются в один, перенос слова только автоматом

white-space: nowrap;   переносы строки запрещены, всё будет в одну строку
```

```
white-space: pre;      текст ведёт себя, будто оформлен в тег <pre>. Перенос слов только явно.

white-space: pre-wrap;  как и pre, но строки автоматом переносятся, если не влезают
white-space: pre-line;  как и pre, но строки переносятся вручную и автоматом + пробелы объед.

white-space: break-spaces;
Сохранены переводы строк, ничего не вылезает, но пробелы интерпретированы в режиме обычного
HTML. (не понял)
```

Свойство outline

Свойство outline
Задаёт дополнительную рамку вокруг элемента, за пределами его CSS-блока

Отличия от border:
1. рамка outline не участвует в блочной модели CSS: не занимает места и не меняет размер элемента.
2. можно задать только со всех сторон

свойство outline-offset задаёт отступ outline от внешней границы элемента

Часто используют для стилей :hover и других аналогичных, когда нужно выделить элемент, но чтобы ничего при этом не прыгало.

outline на MDN

```
color, style, width
outline: green solid 3px;

outline-offset  задаёт отступ outline от внешней границы элемента
```

Свойство box-sizing

Свойство box-sizing определяет как вычисляется общая ширина и высота элемента.

box-sizing на MDN

```
box-sizing: content-box
width и height задаются для контента, а ширина границ и внутренних отступов будет добавлена

box-sizing: border-box
width и height задают высоту ширину всего элемента (с отступами и границей)
```

Свойство margin

Определяет внешний отступ на всех четырёх сторонах элемента.

Особенности:
1. Вертикальные отступы поглощают друг друга, горизонтальные – нет.
Из двух вертикальных отступов выбирается и применяется наибольший.

2. Отрицательные значения margin-top и margin-left
Смещают элемент со своего обычного места. В отличие от position:relative, при сдвиге через margin соседние элементы занимают освободившееся пространство. Элемент продолжает полноценно участвовать в потоке. Исключительно полезное средство позиционирования

3. Отрицательные margin-right/bottom

Не сдвигают элемент, а «укорачивают» его. Хотя сам размер блока не уменьшается, но следующий элемент будет думать, что он меньше на указанное в margin-right/bottom значение.

margin на MDN

Свойство overflow

Управляет тем, как ведёт себя содержимое блочного элемента, если его размер превышает допустимую длину/ширину. Обычно блок увеличивается в размерах при добавлении в него элементов, заключая в себе всех потомков. Если размеры блока указаны явно и он переполняется, то можно использовать overflow.

overflow на MDN

```
overflow
overflow-x
overflow-y

visible
Содержимое может вылезать за границы блока.

hidden
Контент обрезается, без предоставления прокрутки.

scroll
Содержимое обрезается, элементы прокрутки отображены всегда.

auto
Предоставляется прокрутка, если содержимое переполняет блок.
```

Особенности height в %

Если высота внешнего блока вычисляется по содержимому, то высота в % не работает, и заменяется на height:auto. Кроме случая, когда у элемента стоит position:absolute.

то height % работает, если:
- высота внешнего блока задана точно
- внешний блок имеет position:absolute

Если у родительского элемента не установлено height, а указано min-height, то, чтобы дочерний блок занял 100% высоты, нужно родителю поставить свойство height: 1px;

Знаете ли вы селекторы?

Хорошая тема в учебнике здесь.

на MDN

```
Фильтр по месту среди соседей
:first-child   первый потомок своего родителя.
:last-child    последний потомок своего родителя.
:only-child    единственный потомок своего родителя, соседних элементов нет.
:nth-child(a)  потомок номер a своего родителя, например :nth-child(2) – второй потомок.

:nth-child(an+b) –указание номера потомка формулой, где a,b – константы, a под n
подразумевается любое целое число.


Фильтр по месту среди соседей с тем же тегом
аналогичные псевдоклассы, которые учитывают не всех соседей, а только с тем же тегом:
:first-of-type
```

```
:last-of-type
:only-of-type
:nth-of-type
:nth-last-of-type
```

CSS-спрайты

CSS-спрайт – способ объединить много изображений в одно, чтобы:
- Сократить количество обращений к серверу.
- Если у изображений сходная палитра, то объединённое изображение будет весить меньше, чем несколько картинок.

Сдвиг фона background-position позволяет выбирать, какую именно часть спрайта видно.
Для автоматизированной сборки спрайтов используются специальные сервисы.

</>

Перечень методов (новый)

Стандартные встроенные объекты

Перечень Стандартные встроенные объекты здесь.

Array

```
Array на MDN

.length
Отражает количество элементов в массиве.

Методы Array
Array(7)
Передан 1 аргумент-число – создаст пустые слоты.

Array.of()
Создаёт новый экземпляр массива из переданных аргументов.
Передан аргумент-число 7 – создаст один элемент «7», а не length 7.

Array.from(arrayLike)
Создаёт новый экземпляр Array из массивоподобного или итерируемого объекта.

Array.isArray(obj)
Возвращает true, если значение является массивом, иначе возвращает false.



Методы изменения
.pop()
Удаляет последний элемент из массива и возвращает его.
.push()
Добавляет один или более элементов в конец массива и возвращает новую длину массива.
.shift()
Удаляет первый элемент из массива и возвращает его.
.unshift()
Добавляет один или более элементов в начало массива и возвращает новую длину массива.

.reverse()
```

Переворачивает порядок элементов в массиве — первый элемент становится последним, а последний — первым.

.sort()
Сортирует элементы массива на месте и возвращает отсортированный массив.

.splice()
Добавляет и/или удаляет элементы из массива.

.copyWithin()
Копирует последовательность элементов массива внутри массива.
.fill()
Заполняет все элементы массива от начального индекса до конечного индекса указанным значением.

Проверки и поиск
.includes()
Определяет, содержится ли в массиве указанный элемент, возвращая true или false.

.indexOf()
Возвращает первый индекс элемента внутри массива, равный указанному значению или удовлетворяющий результату проверки. -1, если значение не найдено.
.lastIndexOf()
Возвращает последний (наибольший) индекс элемента внутри массива, равный указанному значению; или -1, если значение не найдено.
.findIndex()
Возвращает искомый индекс в массиве, если элемент в массиве удовлетворяет условию проверяющей функции или -1, если такое значение не найдено.
Отличается от .indexOf тем, что ждёт функцию в качестве первого параметра для поиска более сложных, не примитивных значений внутри массива.

.every()
Возвращает true, если каждый элемент в массиве удовлетворяет условию проверяющей функции.
.some()
Возвращает true, если хотя бы один элемент в массиве удовлетворяет условию проверяющей функции.

.find()
Возвращает искомое значение в массиве, если элемент в массиве удовлетворяет условию проверяющей функции или undefined, если такое значение не найдено.

Методы доступа
.concat()
Возвращает новый массив, состоящий из данного массива, соединённого с другим массивом и/или значением (списком массивов/значений).

.join()
Объединяет все элементы массива в строку.

.slice()
Извлекает диапазон значений и возвращает его в виде нового массива.

Методы обхода
.forEach()
Вызывает функцию для каждого элемента в массиве.

.map()
Создаёт новый массив с результатами вызова указанной функции на каждом элементе данного массива.
.filter()
Создаёт новый массив со всеми элементами этого массива, для которых функция фильтрации возвращает true.

.reduce()
Применяет функцию к аккумулятору и каждому значению массива (слева-направо), сводя его к одному значению.

.reduceRight()
Применяет функцию к аккумулятору и каждому значению массива (справа-налево), сводя его к одному значению.

Остальные
.toSource()
Возвращает литеральное представление указанного массива; вы можете использовать это значение для создания нового массива.
.toString()
Возвращает строковое представление массива и его элементов.
.toLocaleString()
Возвращает локализованное строковое представление массива и его элементов.

.entries()
Возвращает новый объект итератора массива Array Iterator, содержащий пары ключ / значение для каждого индекса в массиве.
.keys()
Возвращает новый итератор массива, содержащий ключи каждого индекса в массиве.
.values()
Возвращает новый объект итератора массива Array Iterator, содержащий значения для каждого индекса в массиве.

Array.prototype[@@iterator]()
Возвращает новый объект итератора массива Array Iterator, содержащий значения для каждого индекса в массиве.

Object

Object в **MDN**

delete user.age
это не свойство, а оператор.

Статические методы:
Object.assign()
Создаёт новый объект путём копирования значений всех собственных перечислимых свойств из одного или более исходных объектов в целевой объект.
Object.assign(target, ...sources)

Object.create(proto, [descriptors])
Создаёт новый объект с указанными объектом в качестве прототипа и дескрипторами.
Object.create(proto, [descriptors])

Object.fromEntries()
Returns a new object from an iterable of [key, value] pairs.

Object.is()
Определяет, являются ли два значения одинаковыми

Ключи и значения
Object.keys()
Возвращает массив, содержащий имена всех собственных перечислимых свойств переданного объекта.
Object.values()
Returns an array containing the values that correspond to all of a given object's **own** enumerable string properties.

Object.entries()
Returns an array containing all of the [key, value] pairs of a given object's **own** enumerable string properties.

Object.getOwnPropertyNames()
Возвращает массив, содержащий имена всех переданных объекту **собственных** перечисляемых и неперечисляемых свойств.
Object.getOwnPropertySymbols()
Возвращает массив всех символьных свойств, найденных непосредственно в переданном объекте.

Прототипы
Object.getPrototypeOf(obj)
Возвращает прототип указанного объекта: внутреннее свойство [[Prototype]].

Object.setPrototypeOf(obj, prototype)
Устанавливает прототип указанному объекту: внутреннее свойство [[Prototype]].

Object.create(proto, [descriptors])
Создаёт пустой объект со свойством [[Prototype]], указанным как proto, и необязательными дескрипторами свойств descriptors.

obj.__proto__
Это геттер/сеттер для [[Prototype]]. По спецификации __proto__ должен поддерживаться только браузерами, но по факту его поддерживают все среды. Это API не было стандартизовано.

instance.constructor
Specifies the function that creates an object's prototype.

Дескрипторы
Object.defineProperty()
Добавляет к объекту именованное свойство, описываемое переданным дескриптором.
Object.defineProperties()
Добавляет к объекту именованные свойства, описываемые переданными дескрипторами.
Object.getOwnPropertyDescriptor()
Возвращает дескриптор свойства для именованного свойства объекта.

Запечатывание
Object.freeze()
Замораживает объект: другой код не сможет удалить или изменить никакое свойство.
Object.isFrozen()
Определяет, был ли объект заморожен.

Object.preventExtensions()
Предотвращает любое расширение объекта.
Object.isExtensible()
Определяет, разрешено ли расширение объекта.

Object.seal()
Предотвращает удаление свойств объекта другим кодом.
Object.isSealed()
Определяет, является ли объект запечатанным (sealed).

Методы инстансов
.hasOwnProperty()
Returns a boolean indicating whether an object contains the specified property as a direct property of that object and not inherited through the prototype chain.

.isPrototypeOf()
Returns a boolean indicating whether the object this method is called upon is in the prototype chain of the specified object.

```
.toString()
Returns a string representation of the object.
.valueOf()
Returns the primitive value of the specified object.


Object.observe()
Асинхронно наблюдает за изменениями в объекте.
```

String

```
String на MDN

.charAt()      Возвращает символ по указанному индексу.
.repeat()      возвращает строку, повторённую указанное количество раз.
.trim()        Обрезает пробельные символы в начале и в конце строки.

Перегон в массив
.split()       разбивает String на массив, разделённый указанной строкой на подстроки.


Склейки и извлечение
.concat()      объединяет текст двух строк и возвращает новую строку.
.slice()       извлекает часть строки и возвращает новую строку.
.substring()   Возвращает символы в строке между двумя индексами.
.substr()      Возвращает указанное количество символов в строке, начинающихся с указанной
позиции.


Проверки
.includes()    находится ли строка внутри другой строки.
.startsWith()  начинается ли строка символами другой строки.
.endsWith()    заканчивается ли строка символами другой строки.

.indexOf()     индекс первого вхождения указанного значения или -1, если вхождений нет.
.lastIndexOf() индекс последнего вхождения указанного значения или -1, если вхождений нет.

Мцсм компания стек делала поверку

Регистр
.toLowerCase() все символы в нижнем регистре.
.toUpperCase() все символы в верхнем регистре.


Остальные
.toString()    Возвращает строковое представление указанного объекта.
.valueOf()     Возвращает примитивное значение указанного объекта.

.localeCompare()
???

.prototype[@@iterator]()
Возвращает новый объект итератора Iterator, который итерируется по кодовым точкам строки и
возвращает каждую кодовую точку в виде строкового значения.


Регулярные выражения
.match()     Используется для сопоставления строке регулярного выражения.
.matchAll()  Возвращает итератор по всем результатам при сопоставлении строки с регулярным
выражением.
.replace()   сопоставление строке рег. выражения и для замены совпавшей подстроки на новую
.search()    Выполняет поиск совпадения регулярного выражения со строкой.
```

Number

```
Number на MDN

Статические методы
Number.isNaN()     является ли переданное значение значением NaN.
Number.isFinite()  является ли переданное значение конечным числом
Number.isInteger() является ли число — целым значением.

Number.parseInt()   возвращает целое число из строки
Number.parseFloat() возвращает дробное число из строки


Статические свойства
Number.NEGATIVE_INFINITY
Number.POSITIVE_INFINITY


Методы инстансов
.toFixed(digits)     Returns a string representing the number in fixed-point notation.
```

Math

```
Math на MDN

Math.abs(x)     Возвращает абсолютное значение числа.
Math.max(x, y)  Возвращает наибольшее число из аргументов.
Math.min(x, y)  Возвращает наименьшее число из аргументов.
Math.pow(x, y)  Возвращает x в степени степени y.
Math.sqrt(x)    Возвращает положительный квадратный корень числа.

Math.random()   псевдослучайное число в диапазоне от 0 до 1


Округление
Math.ceil(x)  округление к большему целому
Math.floor(x) округление к меньшему целому
Math.round(x) округление к ближайшему целому
Math.trunc(x) возвращает целую часть числа, убирая дробные цифры.


Случайные числа
Случайное дробное число в заданном интервале
function getRandomArbitrary(min, max) {
  return Math.random() * (max - min) + min;
}

Случайное целое число в заданном интервале
function getRandomInt(min, max) {
  // Максимум не включается, минимум включается
  min = Math.ceil(min);
  max = Math.floor(max);
  return Math.floor(Math.random() * (max - min)) + min;
}

function getRandomIntInclusive(min, max) {
  // Максимум и минимум включаются
```

```
  min = Math.ceil(min);
  max = Math.floor(max);
  return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

Function

Function в MDN

Свойства экземпляра
.arguments
Массив с переданными функции аргументами. Это устаревшее свойство объекта Function. Вместо него используйте объект arguments (доступный внутри функции).
.length
Содержит количество аргументов в функции.
.name
Имя функции.

Методы экземпляра
.call
.apply
.bind

.toString() Возвращает строку с исходным кодом функции

Promise

Promise в MDN

Методы
Promise.all(iterable)
Ожидает исполнения всех промисов или отклонения любого из них. Если любой из промисов будет отклонён, Promise.all будет также отклонён.
Promise.allSettled(iterable)
Ожидает завершения всех полученных промисов (как исполнения так и отклонения).

Promise.race(iterable)
Возвращает промис, который будет исполнен или отклонён первым.

Promise.reject(reason)
Возвращает промис, отклонённый из-за reason.
Promise.resolve(value)
Возвращает промис, исполненный с результатом value.

</>

Чистые таблицы