Практическая работа №12 «Планирование. Декораторы. Флаги и дескрипторы. Геттеры и сеттеры. Прототипное наследование. Классы. Браузерное окружение. DOM-дерево. Атрибуты и свойства. Стили и классы»

Планирование: setTimeout и setInterval

Мы можем вызвать функцию не в данный момент, а позже, через заданный интервал времени. Это называется «планирование вызова».

Для этого существуют два метода:

- setTimeout позволяет вызвать функцию **один раз** через определённый интервал времени.
- setInterval позволяет вызывать функцию **регулярно**, повторяя вызов через определённый интервал времени.

Эти методы не являются частью спецификации JavaScript. Но большинство сред выполнения JS-кода имеют внутренний планировщик и предоставляют доступ к этим методам. В частности, они поддерживаются во всех браузерах и Node.js.

setTimeout

Синтаксис:

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...);
```

Параметры:

func|code

Функция или строка кода для выполнения. Обычно это функция. По историческим причинам можно передать и строку кода, но это не рекомендуется.

delay

Задержка перед запуском в миллисекундах (1000 мс = 1 с). Значение по умолчанию – 0.

arg1, arg2…

Аргументы, передаваемые в функцию (не поддерживается в IE9-)

Например, данный код вызывает sayHi() спустя одну секунду:

```
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}

setTimeout(sayHi, 1000, "Привет", "Джон"); // Привет, Джон

```

Если первый аргумент является строкой, то JavaScript создаст из неё функцию.

Передавайте функцию, но не запускайте её

Начинающие разработчики иногда ошибаются, добавляя скобки () после функции. Это не работает, потому что setTimeout ожидает ссылку на функцию. Здесь sayHi() запускает выполнение функции, и *результат выполнения* отправляется в setTimeout.

Отмена через clearTimeout

Вызов setTimeout возвращает «идентификатор таймера» timerId, который можно использовать для отмены дальнейшего выполнения.

Синтаксис для отмены:

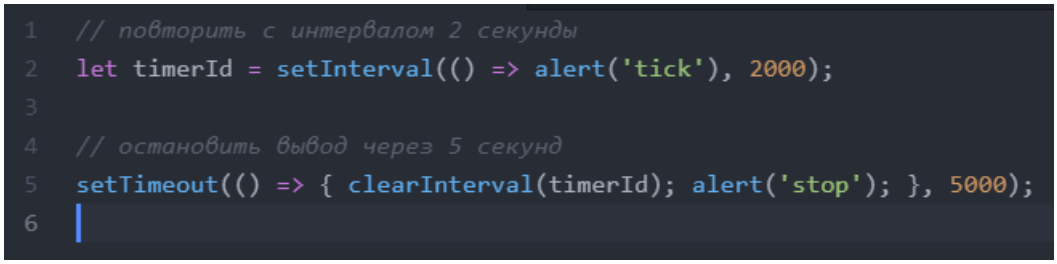
```
let timerId = setTimeout(...);
clearTimeout(timerId);

```

В коде ниже планируем вызов функции и затем отменяем его. В результате ничего не происходит.

setInterval

Метод setInterval имеет такой же синтаксис как setTimeout. Все аргументы имеют такое же значение. Но отличие этого метода от setTimeout в том, что функция запускается не один раз, а периодически через указанный интервал времени. Чтобы остановить дальнейшее выполнение функции, необходимо вызвать clearInterval(timerId).

Следующий пример выводит сообщение каждые 2 секунды. Через 5 секунд вывод прекращается:



```
// повторить с интервалом 2 секунды
let timerId = setInterval(() => alert('tick'), 2000);

// остановить вывод через 5 секунд
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

Во время показа alert время тоже идёт

- Методы setInterval(func, delay, ...args) и setTimeout(func, delay, ...args) позволяют выполнять func регулярно или только один раз после задержки delay, заданной в мс.
- Для отмены выполнения необходимо вызвать clearInterval/clearTimeout со значением, которое возвращают методы setInterval/setTimeout.
- Вложенный вызов setTimeout является более гибкой альтернативой setInterval. Также он позволяет более точно задать интервал между выполнениями.

- Планирование с нулевой задержкой setTimeout(func,0) или, что то же самое, setTimeout(func) используется для вызовов, которые должны быть исполнены как можно скорее, после завершения исполнения текущего кода.
- Браузер ограничивает 4-мя мс минимальную задержку между пятью и более вложенными вызовами setTimeout, а также для setInterval, начиная с 5-го вызова.

Обратим внимание, что все методы планирования *не гарантируют* точную задержку.

Например, таймер в браузере может замедляться по многим причинам:

- Перегружен процессор.
- Вкладка браузера в фоновом режиме.
- Работа ноутбука от аккумулятора.

Всё это может увеличивать минимальный интервал срабатывания таймера (и минимальную задержку) до 300 или даже 1000 мс в зависимости от браузера и настроек производительности ОС.

Декораторы и переадресация вызова, call/apply

JavaScript предоставляет исключительно гибкие возможности по работе с функциями: они могут быть переданы в другие функции, использованы как объекты, и сейчас мы рассмотрим, как *перенаправлять* вызовы между ними и как их декорировать.

Декоратор – это обёртка вокруг функции, которая изменяет поведение последней. Основная работа по-прежнему выполняется функцией.

Обычно безопасно заменить функцию или метод декорированным, за исключением одной мелочи. Если исходная функция предоставляет свойства,

такие как func.calledCount или типа того, то декорированная функция их не предоставит. Потому что это обёртка. Так что нужно быть осторожным в их использовании. Некоторые декораторы предоставляют свои собственные свойства.

Декораторы можно рассматривать как «дополнительные возможности» или «аспекты», которые можно добавить в функцию. Мы можем добавить один или несколько декораторов. И всё это без изменения кода оригинальной функции!

Прозрачное кеширование

Представим, что у нас есть функция slow(x), выполняющая ресурсоёмкие вычисления, но возвращающая стабильные результаты. Другими словами, для одного и того же x она всегда возвращает один и тот же результат. Если функция вызывается часто, то, вероятно, мы захотим кешировать (запоминать) возвращаемые ею результаты, чтобы сэкономить время на повторных вычислениях.

Вместо того, чтобы усложнять slow(x) дополнительной функциональностью, мы заключим её в функцию-обёртку – «wrapper» (от англ. «wrap» – обёртывать), которая добавит кеширование. Далее мы увидим, что в таком подходе масса преимуществ.

Вот код с объяснениями:

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

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

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

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

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

slow = cachingDecorator(slow);

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

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

```

В коде выше cachingDecorator – это *декоратор*, специальная функция, которая принимает другую функцию и изменяет её поведение. Идея состоит в том, что мы можем вызвать cachingDecorator с любой функцией, в результате чего мы получим кеширующую обёртку. Это здорово, т.к. у нас может быть множество функций, использующих такую функциональность, и всё, что нам нужно сделать – это применить к ним cachingDecorator. Отделяя кеширующий код от основного кода, мы также сохраняем чистоту и простоту последнего. Результат вызова cachingDecorator(func) является «обёрткой», т.е. function(x) «оборачивает» вызов func(x) в кеширующую логику:

С точки зрения внешнего кода, обёрнутая функция slow по-прежнему делает то же самое. Обёртка всего лишь добавляет к её поведению аспект кеширования.

Подводя итог, можно выделить несколько преимуществ использования отдельной cachingDecorator вместо изменения кода самой slow:

- Функцию cachingDecorator можно использовать повторно. Мы можем применить её к другой функции.
- Логика кеширования является отдельной, она не увеличивает сложность самой slow (если таковая была).
- При необходимости мы можем объединить несколько декораторов (речь об этом пойдёт позже).

Применение «func.call» для передачи контекста.

Упомянутый выше кеширующий декоратор не подходит для работы с методами объектов. Например, в приведённом ниже коде worker.slow() перестаёт работать после применения декоратора:

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

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

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

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

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

alert( worker.slow(2) ); // Ошибка: не удаётся прочитать свойство 'someMethod' из 'undefined'

```

Ошибка возникает в строке (*). Функция пытается получить доступ к this.someMethod и завершается с ошибкой. Причина в том, что в строке (**) декоратор вызывает оригинальную функцию как func(x), и она в данном случае получает this = undefined. Декоратор передаёт вызов оригинальному методу, но без контекста. Следовательно – ошибка.

Существует специальный встроенный метод функции func.call(context, …args), который позволяет вызывать функцию, явно устанавливая this.

Синтаксис:

```
func.call(context, arg1, arg2, ...)
```

Он запускает функцию func, используя первый аргумент как её контекст this, а последующие – как её аргументы. Проще говоря, эти два вызова делают почти то же самое:

```
func(1, 2, 3);
func.call(obj, 1, 2, 3)
```

Они оба вызывают func с аргументами 1, 2 и 3. Единственное отличие состоит в том, что func.call ещё и устанавливает this равным obj. Например, в приведённом ниже коде мы вызываем sayHi в контексте различных объектов: sayHi.call(user) запускает sayHi, передавая this=user, а следующая строка устанавливает this=admin:

```
function sayHi() {
  alert(this.name);
}

let user = { name: "John" };
let admin = { name: "Admin" };

// используем 'call' для передачи различных объектов в качестве 'this'
sayHi.call( user ); // John
sayHi.call( admin ); // Admin
```

Здесь мы используем call для вызова say с заданным контекстом и фразой:

```
function say(phrase) {
  alert(this.name + ': ' + phrase);
}

let user = { name: "John" };

// 'user' становится 'this', и "Hello" становится первым аргументом
say.call( user, "Hello" ); // John: Hello
```

В нашем случае мы можем использовать call в обёртке для передачи контекста в исходную функцию:

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

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

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

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

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

```

Теперь всё в порядке.

Чтобы всё было понятно, давайте посмотрим глубже, как передаётся this:

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

Переходим к нескольким аргументам с «func.apply»

Теперь давайте сделаем cachingDecorator ещё более универсальным. До сих пор он работал только с функциями с одним аргументом.

```
let worker = {
  slow(min, max) {
    return min + max; // здесь может быть тяжёлая задача
  }
};

// будет кешировать вызовы с одинаковыми аргументами
worker.slow = cachingDecorator(worker.slow);
```

Здесь у нас есть две задачи для решения.

Во-первых, как использовать оба аргумента min и max для ключа в коллекции cache? Ранее для одного аргумента x мы могли просто сохранить результат cache.set(x, result) и вызвать cache.get(x), чтобы получить его позже. Но теперь нам нужно запомнить результат для *комбинации аргументов* (min,max). Встроенный Map принимает только одно значение как ключ.

Есть много возможных решений:

1. Реализовать новую (или использовать стороннюю) структуру данных для коллекции, которая более универсальна, чем встроенный Map, и поддерживает множественные ключи.
2. Использовать вложенные коллекции: cache.set(min) будет Map, которая хранит пару (max, result). Тогда получить result мы сможем, вызвав cache.get(min).get(max).
3. Соединить два значения в одно. В нашем конкретном случае мы можем просто использовать строку "min,max" как ключ к Map. Для гибкости, мы можем позволить передавать *хеширующую функцию* в декоратор, которая знает, как сделать одно значение из многих.

Для многих практических применений третий вариант достаточно хорош, поэтому мы будем придерживаться его.

Также нам понадобится заменить func.call(this, x) на func.call(this, ...arguments), чтобы передавать все аргументы обёрнутой функции, а не только первый.

Вот более мощный cachingDecorator:

```
let worker = {
  slow(min, max) {
    alert(`Called with ${min},${max}`);
    return min + max;
  }
};

function cachingDecorator(func, hash) {
  let cache = new Map();
  return function() {
    let key = hash(arguments); // (*)
    if (cache.has(key)) {
      return cache.get(key);
    }

    let result = func.call(this, ...arguments); // (**)

    cache.set(key, result);
    return result;
  };
}

function hash(args) {
  return args[0] + ',' + args[1];
}

worker.slow = cachingDecorator(worker.slow, hash);

alert( worker.slow(3, 5) ); // работает
alert( "Again " + worker.slow(3, 5) ); // аналогично (из кеша)

```

Теперь он работает с любым количеством аргументов.

Есть два изменения:

- В строке (*) вызываем hash для создания одного ключа из arguments. Здесь мы используем простую функцию «объединения», которая

превращает аргументы (3, 5) в ключ "3,5". В более сложных случаях могут потребоваться другие функции хеширования.

- Затем в строке (**) используем func.call(this, ...arguments) для передачи как контекста, так и всех аргументов, полученных обёрткой (независимо от их количества), в исходную функцию.

Вместо func.call(this, ...arguments) мы могли бы написать func.apply(this, arguments).

Синтаксис встроенного метода func.apply:

```
1  func.apply(context, args)
```

Он выполняет func, устанавливая this=context и принимая в качестве списка аргументов псевдомассив args. Единственная разница в синтаксисе между call и apply состоит в том, что call ожидает список аргументов, в то время как apply принимает псевдомассив.

Флаги и дескрипторы свойств

Как мы знаем, объекты могут содержать свойства. До этого момента мы рассматривали свойство только как пару «ключ-значение». Но на самом деле свойство объекта гораздо мощнее и гибче.

Флаги свойств

Помимо значения **value**, свойства объекта имеют три специальных атрибута (так называемые «флаги»).

- **writable** – если true, свойство можно изменить, иначе оно только для чтения.
- **enumerable** – если true, свойство перечисляется в циклах, в противном случае циклы его игнорируют.

- **configurable** – если true, свойство можно удалить, а эти атрибуты можно изменять, иначе этого делать нельзя.

Мы ещё не встречали эти атрибуты, потому что обычно они скрыты. Когда мы создаём свойство «обычным способом», все они имеют значение true. Но мы можем изменить их в любое время.

Метод Object.getOwnPropertyDescriptor позволяет получить *полную* информацию о свойстве.

Его синтаксис:

```
let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);
```

obj

Объект, из которого мы получаем информацию.

propertyName

Имя свойства.

Возвращаемое значение – это объект, так называемый «дескриптор свойства»: он содержит значение свойства и все его флаги.

Например:

```
let user = {
  name: "John"
};

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );
/* дескриптор свойства:
{
  "value": "John",
  "writable": true,
  "enumerable": true,
  "configurable": true
}
*/
```

Чтобы изменить флаги, мы можем использовать метод Object.defineProperty.

Его синтаксис:

```
Object.defineProperty(obj, propertyName, descriptor)
```

obj, propertyName

Объект и его свойство, для которого нужно применить дескриптор.

descriptor

Применяемый дескриптор.

Если свойство существует, defineProperty обновит его флаги. В противном случае метод создаёт новое свойство с указанным значением и флагами; если какой-либо флаг не указан явно, ему присваивается значение false.

Например, здесь создаётся свойство name, все флаги которого имеют значение false:

```
let user = {};

Object.defineProperty(user, "name", {
  value: "John"
});

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
  "value": "John",
  "writable": false,
  "enumerable": false,
  "configurable": false
}
 */
```

Сравните это с предыдущим примером, в котором мы создали свойство user.name «обычным способом»: в этот раз все флаги имеют значение false. Если это не то, что нам нужно, надо присвоить им значения true в параметре descriptor. Теперь давайте рассмотрим на примерах, что нам даёт использование флагов.

Только для чтения

Сделаем свойство user.name доступным только для чтения. Для этого изменим флаг writable:

```
let user = {
  name: "John"
};

Object.defineProperty(user, "name", {
  writable: false
});

user.name = "Pete"; // Ошибка: Невозможно изменить доступное только для чтения свойство 'name'
```

Теперь никто не сможет изменить имя пользователя, если только не обновит соответствующий флаг новым вызовом defineProperty.

Ошибки появляются только в строгом режиме

В нестрогом режиме, без use strict, мы не увидим никаких ошибок при записи в свойства «только для чтения» и т.п. Но эти операции всё равно не будут выполнены успешно. Действия, нарушающие ограничения флагов, в нестрогом режиме просто молча игнорируются.

Неперечислимое свойство

Теперь добавим собственный метод toString к объекту user. Встроенный метод toString в объектах – неперечислимый, его не видно в цикле for..in. Но если мы напишем свой собственный метод toString, цикл for..in будет выводить его по умолчанию:

```
let user = {
  name: "John",
  toString() {
    return this.name;
  }
};

// По умолчанию оба свойства выведутся:
for (let key in user) alert(key); // name, toString

```

Если мы этого не хотим, можно установить для свойства enumerable:false. Тогда оно перестанет появляться в цикле for..in аналогично встроенному toString:

```
let user = {
  name: "John",
  toString() {
    return this.name;
  }
};

Object.defineProperty(user, "toString", {
  enumerable: false
});

// Теперь наше свойство toString пропало из цикла:
for (let key in user) alert(key); // name

```

Неперечислимые свойства также не возвращаются Object.keys:

```
alert(Object.keys(user)); // name
```

Неконфигурируемое свойство

Флаг неконфигурируемого свойства (configurable:false) иногда предустановлен для некоторых встроенных объектов и свойств. Неконфигурируемое свойство не может быть удалено. Например, свойство Math.PI – только для чтения, неперечислимое и неконфигурируемое:

```
let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');

alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
  "value": 3.141592653589793,
  "writable": false,
  "enumerable": false,
  "configurable": false
}
*/

```

То есть программист не сможет изменить значение Math.PI или перезаписать его.

```
Math.PI = 3; // Ошибка

// delete Math.PI тоже не сработает

```

Определение свойства как неконфигурируемого – это дорога в один конец. Мы не сможем отменить это действие, потому что defineProperty не работает с неконфигурируемыми свойствами.

Ошибки отображаются только в строгом режиме

В нестрогом режиме мы не увидим никаких ошибок при записи в свойства «только для чтения» и т.п. Эти операции всё равно не будут выполнены успешно. Действия, нарушающие ограничения флагов, в нестрогом режиме просто молча игнорируются.

Метод Object.defineProperties

Существует метод Object.defineProperties(obj, descriptors), который позволяет определять множество свойств сразу.

Его синтаксис:

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

```

Таким образом, мы можем определить множество свойств одной операцией.

Object.getOwnPropertyDescriptors

Чтобы получить все дескрипторы свойств сразу, можно воспользоваться методом Object.getOwnPropertyDescriptors(obj). Вместе с Object.defineProperties этот метод можно использовать для клонирования объекта вместе с его флагами:

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

Обычно при клонировании объекта мы используем присваивание, чтобы скопировать его свойства. Но это не копирует флаги. Так что, если нам нужен клон «получше», предпочтительнее использовать Object.defineProperties.

Глобальное запечатывание объекта

Дескрипторы свойств работают на уровне конкретных свойств.

Но ещё 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.

На практике эти методы используются редко.

Свойства - геттеры и сеттеры

Есть два типа свойств объекта.

Первый тип это *свойства-данные (data properties)*. Мы уже знаем, как работать с ними. Все свойства, которые мы использовали до текущего момента, были свойствами-данными.

Второй тип свойств мы ещё не рассматривали. Это *свойства-аксессоры (accessor properties)*. По своей сути это функции, которые используются для присвоения и получения значения, но во внешнем коде они выглядят как обычные свойства объекта.

Геттеры и сеттеры

Свойства-аксессоры представлены методами: «геттер» – для чтения и «сеттер» – для записи. При литеральном объявлении объекта они обозначаются get и set:

```
let obj = {
  get propName() {
    // геттер, срабатывает при чтении obj.propName
  },

  set propName(value) {
    // сеттер, срабатывает при записи obj.propName = value
  }
};

```

Геттер срабатывает, когда obj.propName читается, сеттер – когда значение присваивается. Например, у нас есть объект user со свойствами name и surname:

```
let user = {
  name: "John",
  surname: "Smith"
};
```

А теперь добавим свойство объекта fullName для полного имени, которое в нашем случае "John Smith". Само собой, мы не хотим дублировать уже имеющуюся информацию, так что реализуем его при помощи аксессора:

```
let user = {
  name: "John",
  surname: "Smith",

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

alert(user.fullName); // John Smith
```

Снаружи свойство-аксессор выглядит как обычное свойство. В этом и заключается смысл свойств-аксессоров. Мы не *вызываем* user.fullName как функцию, а *читаем* как обычное свойство: геттер выполнит всю работу за кулисами. На данный момент у fullName есть только геттер. Если мы попытаемся назначить user.fullName=, произойдёт ошибка.

Дескрипторы свойств доступа

Дескрипторы свойств-аксессоров отличаются от «обычных» свойств-данных.

Свойства-аксессоры не имеют value и writable, но взамен предлагают функции get и set.

То есть, дескриптор аксессора может иметь:

- **get** – функция без аргументов, которая сработает при чтении свойства,
- **set** – функция, принимающая один аргумент, вызываемая при присвоении свойства,
- **enumerable** – то же самое, что и для свойств-данных,
- **configurable** – то же самое, что и для свойств-данных.

Например, для создания аксессора fullName при помощи defineProperty мы можем передать дескриптор с использованием get и set.

При попытке указать и get, и value в одном дескрипторе будет ошибка

Умные геттеры/сеттеры

Геттеры/сеттеры можно использовать как обёртки над «реальными» значениями свойств, чтобы получить больше контроля над операциями с ними. Например, если мы хотим запретить устанавливать короткое имя для user, мы можем использовать сеттер name для проверки, а само значение хранить в отдельном свойстве _name:

```
let user = {
  get name() {
    return this._name;
  },

  set name(value) {
    if (value.length < 4) {
      alert("Имя слишком короткое, должно быть более 4 символов");
      return;
    }
    this._name = value;
  }
};

user.name = "Pete";
alert(user.name); // Pete

user.name = ""; // Имя слишком короткое...
```

Таким образом, само имя хранится в _name, доступ к которому производится через геттер и сеттер. Технически, внешний код всё ещё может получить доступ к имени напрямую с помощью user._name, но существует широко известное соглашение о том, что свойства, которые начинаются с символа "_", являются внутренними, и к ним не следует обращаться из-за пределов объекта.

Прототипное наследование

В программировании мы часто хотим взять что-то и расширить. Например, у нас есть объект user со своими свойствами и методами, и мы хотим создать объекты admin и guest как его слегка изменённые варианты. Мы хотели бы повторно использовать то, что есть у объекта user, не копировать/переопределять его методы, а просто создать новый объект на его основе.

Прототипное наследование — это возможность языка, которая помогает в этом.

[[Prototype]]

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

- Когда мы хотим прочитать свойство из object, а оно отсутствует, JavaScript автоматически берёт его из прототипа. В программировании такой механизм называется «прототипным наследованием». Многие интересные возможности языка и техники программирования основываются на нём.
- Свойство [[Prototype]] является внутренним и скрытым, но есть много способов задать его.

Одним из них является использование __proto__, например так:

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

rabbit.__proto__ = animal;

```

Если мы ищем свойство в rabbit, а оно отсутствует, JavaScript автоматически берёт его из animal.

Например:

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

rabbit.__proto__ = animal; // (*)

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

```

Здесь строка (*) устанавливает animal как прототип для rabbit. Затем, когда alert пытается прочитать свойство rabbit.eats (**), его нет в rabbit, поэтому JavaScript следует по ссылке [[Prototype]] и находит его в animal. Так что если у animal много полезных свойств и методов, то они автоматически становятся доступными у rabbit. Такие свойства называются «унаследованными».

Есть только два ограничения:

1. Ссылки не могут идти по кругу. JavaScript выдаст ошибку, если мы попытаемся назначить __proto__ по кругу.

2. Значение __proto__ может быть объектом или null. Другие типы игнорируются.

Это вполне очевидно, но всё же: может быть только один [[Prototype]]. Объект не может наследоваться от двух других объектов.

Свойство __proto__ — исторически обусловленный геттер/сеттер для [[Prototype]]

Это распространённая ошибка начинающих разработчиков – не знать разницы между этими двумя понятиями. Обратите внимание, что __proto__ — *не то же самое*, что внутреннее свойство [[Prototype]]. Это геттер/сеттер для [[Prototype]]. Позже мы увидим ситуации, когда это имеет значение, а пока давайте просто будем иметь это в виду, поскольку мы строим наше понимание языка JavaScript.

Свойство __proto__ немного устарело, оно существует по историческим причинам. Современный JavaScript предполагает, что мы должны использовать функции Object.getPrototypeOf/Object.setPrototypeOf вместо того, чтобы получать/устанавливать прототип. Мы также рассмотрим эти функции позже. По спецификации __proto__ должен поддерживаться только браузерами, но по факту все среды, включая серверную, поддерживают его.

В итоге про прототипы:

- В JavaScript все объекты имеют скрытое свойство [[Prototype]], которое является либо другим объектом, либо null.
- Мы можем использовать obj.__proto__ для доступа к нему (исторически обусловленный геттер/сеттер, есть другие способы, которые скоро будут рассмотрены).
- 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.

Класс: базовый синтаксис

На практике нам часто надо создавать много объектов одного вида, например пользователей, товары или что-то ещё. Как мы уже знаем из главы Конструктор, оператор "new", с этим может помочь new function.

Но в современном JavaScript есть и более продвинутая конструкция «class», которая предоставляет новые возможности, полезные для объектно-ориентированного программирования.

Синтаксис «class»

Базовый синтаксис выглядит так:

Затем используйте вызов new MyClass() для создания нового объекта со всеми перечисленными методами. При этом автоматически вызывается метод constructor(), в нём мы можем инициализировать объект.

Например:

```
class User {

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

  sayHi() {
    alert(this.name);
  }

}

// Использование:
let user = new User("Иван");
user.sayHi();
```

Когда вызывается new User("Иван"):

1. Создаётся новый объект.
2. constructor запускается с заданным аргументом и сохраняет его в this.name.

Методы в классе не разделяются запятой

Частая ошибка начинающих разработчиков – ставить запятую между методами класса, что приводит к синтаксической ошибке. Синтаксис классов отличается от литералов объектов, не путайте их. Внутри классов запятые не требуются.

new MyClass().

Базовый синтаксис для классов выглядит так:

```
class MyClass {
  prop = value; // свойство
  constructor(...) { // конструктор
    // ...
  }
  method(...) {} // метод
  get something(...) {} // геттер
  set something(...) {} // сеттер
  [Symbol.iterator]() {} // метод с вычисляемым именем (здесь - символом)
  // ...
}
```

MyClass технически является функцией (той, которую мы определяем как constructor), в то время как методы, геттеры и сеттеры записываются в MyClass.prototype.

Наследование классов

Наследование классов – это способ расширения одного класса другим классом. Таким образом, мы можем добавить новый функционал к уже существующему.

Ключевое слово «extends»

Допустим, у нас есть класс Animal:

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed = speed;
    alert(`${this.name} бежит со скоростью ${this.speed}.`);
  }
  stop() {
    this.speed = 0;
    alert(`${this.name} стоит неподвижно.`);
  }
}

let animal = new Animal("Мой питомец");
```

Поскольку кролики – это животные, класс Rabbit должен быть основан на Animal, и иметь доступ к методам животных, так чтобы кролики могли делать то, что могут делать «общие» животные. Синтаксис для расширения другого класса следующий: class Child extends Parent.

Создадим class Rabbit, который наследуется от Animal:

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

let rabbit = new Rabbit("Белый кролик");

rabbit.run(5); // Белый кролик бежит со скоростью 5.
rabbit.hide(); // Белый кролик прячется!
```

Объект класса Rabbit имеет доступ как к методам Rabbit, таким как rabbit.hide(), так и к методам Animal, таким как rabbit.run(). Внутри ключевое слово extends работает по старой доброй механике прототипов. Оно устанавливает Rabbit.prototype.[[Prototype]] в Animal.prototype. Таким образом, если метода не оказалось в Rabbit.prototype, JavaScript берет его из Animal.prototype.

Например, чтобы найти метод rabbit.run, движок проверяет (снизу вверх на картинке):

1. Объект rabbit (не имеет run).
2. Его прототип, то есть Rabbit.prototype (имеет hide, но не имеет run).
3. Его прототип, то есть (вследствие extends) Animal.prototype, в котором, наконец, есть метод run.

После extends разрешены любые выражения

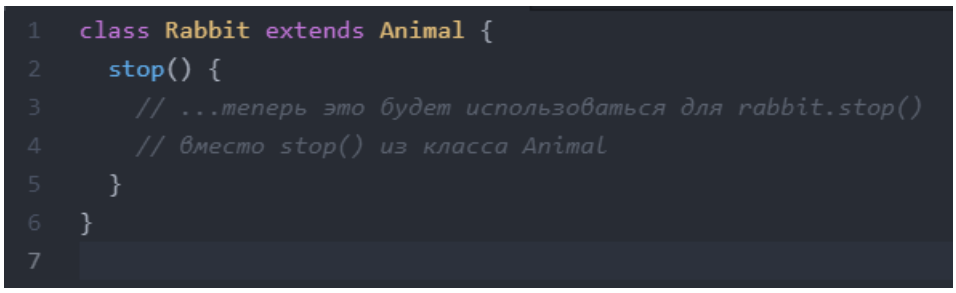
Синтаксис создания класса допускает указывать после extends не только класс, но и любое выражение. Пример вызова функции, которая генерирует родительский класс:

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

class User extends f("Привет") {}

new User().sayHi(); // Привет
```

Здесь class User наследует от результата вызова f("Привет"). 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.

Переопределение методов

Теперь давайте продвинемся дальше и переопределим метод. По умолчанию все методы, не указанные в классе Rabbit, берутся непосредственно «как есть» из класса Animal. Но если мы укажем в Rabbit собственный метод, например stop(), то он будет использован вместо него:



```
class Rabbit extends Animal {
  stop() {
    // ...теперь это будет использоваться для rabbit.stop()
    // вместо stop() из класса Animal
  }
}
```

Впрочем, обычно мы не хотим полностью заменить родительский метод, а скорее хотим сделать новый на его основе, изменяя или расширяя его функциональность. Мы делаем что-то в нашем методе и вызываем родительский метод до/после или в процессе.

У классов есть ключевое слово "super" для таких случаев.

- super.method(...) вызывает родительский метод.
- super(...) для вызова родительского конструктора (работает только внутри нашего конструктора).

Краткая справка для самостоятельного изучения.

1. Чтобы унаследовать от класса: class Child extends Parent:
    - При этом Child.prototype.__proto__ будет равен Parent.prototype, так что методы будут унаследованы.
2. При переопределении конструктора:
    - Обязателен вызов конструктора родителя super() в конструкторе Child до обращения к this.
3. При переопределении другого метода:
    - Мы можем вызвать super.method() в методе Child для обращения к методу родителя Parent.
4. Внутренние детали:
    - Методы запоминают свой объект во внутреннем свойстве [[HomeObject]]. Благодаря этому работает super, он в его прототипе ищет родительские методы.

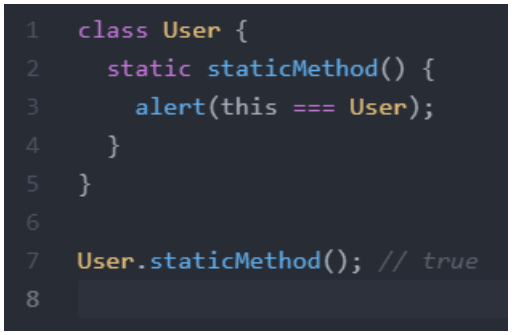
- Поэтому копировать метод, использующий super, между разными объектами небезопасно.

Также:

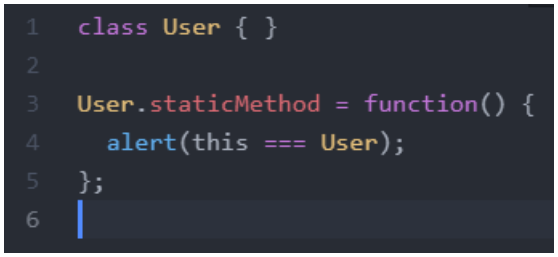
- У функций-стрелок нет своего this и super, поэтому они «прозрачно» встраиваются во внешний контекст.

Статические свойства и методы

Мы также можем присвоить метод самому классу. Такие методы называются *статическими*. В объявление класса они добавляются с помощью ключевого слова static, например:



Это фактически то же самое, что присвоить метод напрямую как свойство функции:



Значением this при вызове User.staticMethod() является сам конструктор класса User (правило «объект до точки»). Обычно статические методы используются для реализации функций, принадлежащих классу целиком, вообще, и при этом не относящимся к каким-то отдельным объектам.

Например, есть объекты статей Article, и нужна функция для их сравнения. Естественное решение – сделать для этого 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);

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

Здесь метод Article.compare стоит «над» статьями, как средство для их сравнения. Это метод не отдельной статьи, а всего класса. Другим примером может быть так называемый «фабричный» метод. Скажем, нам нужно несколько способов создания статьи:

1. Создание через заданные параметры (title, date и т. д.).
2. Создание пустой статьи с сегодняшней датой.
3. …

Первый способ может быть реализован через конструктора. А для второго можно использовать статический метод класса. Такой как Article.createTodays() в следующем примере:

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

  static createTodays() {
    // помним, что this = Article
    return new this("Сегодняшний дайджест", new Date());
  }
}

let article = Article.createTodays();

alert( article.title ); // Сегодняшний дайджест
```

Теперь каждый раз, когда нам нужно создать сегодняшний дайджест, нужно вызывать Article.createTodays(). Ещё раз, это не метод одной статьи, а метод всего класса. 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 по id:
Article.remove({id: 12345});
```

Статические методы недоступны для отдельных объектов

Статические методы могут вызываться для классов, но не для отдельных объектов.

Статические свойства

Статические свойства также возможны, они выглядят как свойства класса, но с static в начале:

Это то же самое, что и прямое присваивание Article.

Что в итоге?

- Статические методы используются для функциональности, принадлежат классу «в целом», а не относятся к конкретному объекту класса. Например, метод для сравнения двух статей Article.compare(article1, article2) или фабричный метод Article.createTodays().
- В объявлении класса они помечаются ключевым словом static.
- Статические свойства используются в тех случаях, когда мы хотели бы сохранить данные на уровне класса, а не какого-то одного объекта.
- Технически, статическое объявление – это то же самое, что и присвоение классу.
- Статические свойства и методы наследуются.
- Для class B extends A прототип класса B указывает на A: B.[[Prototype]] = A. Таким образом, если поле не найдено в B, поиск продолжается в A.

Приватные и защищённые методы и свойства

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

Внутренний и внешний интерфейсы

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.
- *Внешний интерфейс* – методы и свойства, доступные снаружи класса.

Всё, что нам нужно для использования объекта, это знать его внешний интерфейс. Мы можем совершенно не знать, как это работает внутри, и это здорово.

В JavaScript есть два типа полей (свойств и методов) объекта:

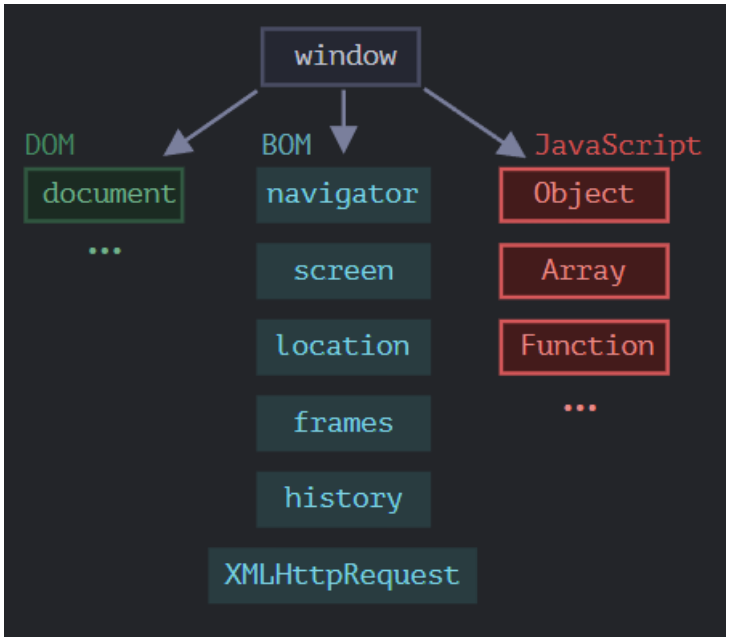
- Публичные: доступны отовсюду. Они составляют внешний интерфейс. До этого момента мы использовали только публичные свойства и методы.
- Приватные: доступны только внутри класса. Они для внутреннего интерфейса.

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.

Браузерное окружение, спецификации

Язык JavaScript изначально был создан для веб-браузеров. Но с тех пор он значительно эволюционировал и превратился в кроссплатформенный язык программирования для решения широкого круга задач. Сегодня JavaScript может использоваться в браузере, на веб-сервере или в какой-то другой среде. Каждая среда предоставляет свою функциональность, которую спецификация JavaScript называет *окружением*.

Окружение предоставляет свои объекты и дополнительные функции, в дополнение базовым языковым. Браузеры, например, дают средства для управления веб-страницами. Node.js делает доступными какие-то серверные возможности и так далее. На картинке ниже в общих чертах показано, что доступно для JavaScript в браузерном окружении:



Как мы видим, имеется корневой объект `window`, который выступает в 2 ролях:

1. Во-первых, это глобальный объект для JavaScript-кода.
2. Во-вторых, он также представляет собой окно браузера и располагает методами для управления им.

Например, здесь мы используем `window` как глобальный объект:

```
function sayHi() {
  alert("Hello");
}

// глобальные функции доступны как методы глобального объекта:
window.sayHi();
```

А здесь мы используем `window` как объект окна браузера, чтобы узнать его высоту:

```
alert(window.innerHeight); // внутренняя высота окна браузера
```

Существует гораздо больше свойств и методов для управления окном браузера.

DOM (Document Object Model)

Document Object Model, сокращённо DOM – объектная модель документа, которая представляет все содержимое страницы в виде объектов, которые можно менять. Объект `document` – основная «входная точка». С его помощью мы можем что-то создавать или менять на странице.

Например:

```
// заменим цвет фона на красный,
document.body.style.background = "red";

// а через секунду вернём как было
setTimeout(() => document.body.style.background = "", 1000);
```

Мы использовали в примере только `document.body.style`, но на самом деле возможности по управлению страницей намного шире. Различные свойства и методы описаны в спецификации:

- **DOM Living Standard** на https://dom.spec.whatwg.org

DOM – не только для браузеров

Спецификация DOM описывает структуру документа и предоставляет объекты для манипуляций со страницей. Существуют и другие, отличные от браузеров, инструменты, использующие DOM.

Например, серверные скрипты, которые загружают и обрабатывают HTML-страницы, также могут использовать DOM. При этом они могут поддерживать спецификацию не полностью.

CSSOM для стилей

Правила стилей CSS структурированы иначе чем HTML. Для них есть отдельная спецификация CSSOM, которая объясняет, как стили должны представляться в виде объектов, как их читать и писать.

CSSOM используется вместе с DOM при изменении стилей документа. В реальности CSSOM требуется редко, обычно правила CSS статичны. Мы редко добавляем/удаляем стили из JavaScript, но и это возможно.

BOM (Browser Object Model)

Объектная модель браузера (Browser Object Model, BOM) – это дополнительные объекты, предоставляемые браузером (окружением), чтобы работать со всем, кроме документа.

Например:

- Объект navigator даёт информацию о самом браузере и операционной системе. Среди множества его свойств самыми известными являются: navigator.userAgent – информация о текущем браузере, и navigator.platform – информация о платформе (может помочь в понимании того, в какой ОС открыт браузер – Windows/Linux/Mac и так далее).
- Объект location позволяет получить текущий URL и перенаправить браузер по новому адресу.

Вот как мы можем использовать объект `location`:

```
alert(location.href); // показывает текущий URL
if (confirm("Перейти на Wikipedia?")) {
  location.href = "https://wikipedia.org"; // перенаправляет браузер на другой URL
}

```

Функции `alert/confirm/prompt` тоже являются частью BOM: они не относятся непосредственно к странице, но представляют собой методы объекта окна браузера для коммуникации с пользователем.

Спецификация DOM

описывает структуру документа, манипуляции с контентом и события, подробнее на https://dom.spec.whatwg.org.

Спецификация CSSOM

Описывает файлы стилей, правила написания стилей и манипуляций с ними, а также то, как это всё связано со страницей, подробнее на https://www.w3.org/TR/cssom-1/.

Спецификация HTML

Описывает язык HTML (например, теги) и BOM (объектную модель браузера) – разные функции браузера: setTimeout, alert, location и так далее, подробнее на https://html.spec.whatwg.org. Тут берётся за основу спецификация DOM и расширяется дополнительными свойствами и методами.

Кроме того, некоторые классы описаны отдельно на https://spec.whatwg.org/.

DOM-дерево

Основой HTML-документа являются теги. В соответствии с объектной моделью документа («Document Object Model», коротко DOM), каждый HTML-тег является объектом. Вложенные теги являются «детьми» родительского элемента. Текст, который находится внутри тега, также является объектом. Все эти объекты доступны при помощи JavaScript, мы можем использовать их для изменения страницы.

Например, `document.body` – объект для тега `<body>`.

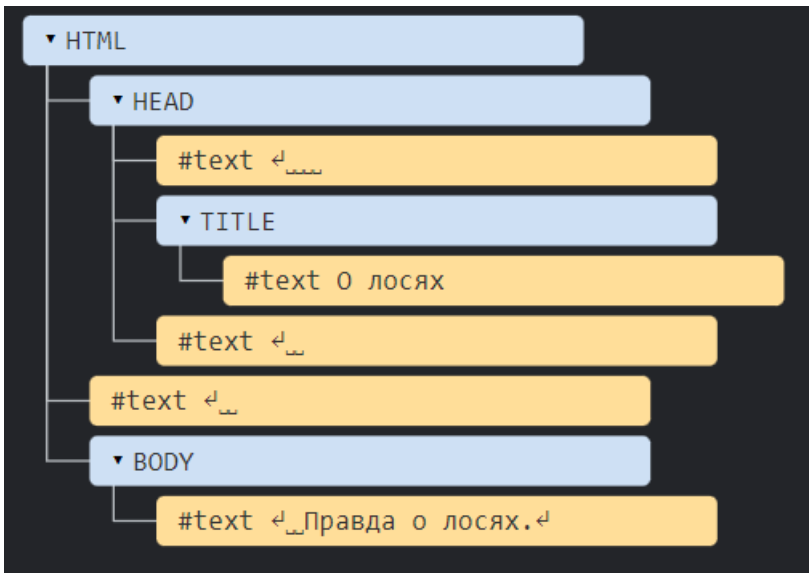
Если запустить этот код, то `<body>` станет красным на 3 секунды:

```
document.body.style.background = 'red'; // сделать фон красным

setTimeout(() => document.body.style.background = '', 3000); // вернуть назад

```

Пример DOM

```
<!DOCTYPE HTML>
<html>
<head>
  <title>О лосях</title>
</head>
<body>
  Правда о лосях.
</body>
</html>

```

DOM – это представление HTML-документа в виде дерева тегов. Вот как оно выглядит:



На рисунке выше узлы-элементы можно кликать, и их дети будут скрываться и раскрываться. Каждый узел этого дерева – это объект. Теги являются *узлами-элементами* (или просто элементами). Они образуют структуру дерева: <html> – это корневой узел, <head> и <body> его дочерние узлы и т.д.

Текст внутри элементов образует *текстовые узлы*, обозначенные как #text. Текстовый узел содержит в себе только строку текста. У него не может быть потомков, т.е. он находится всегда на самом нижнем уровне.

Обратите внимание на специальные символы в текстовых узлах:

- перевод строки: ↲ (в JavaScript он обозначается как \n)
- пробел: ␣

Пробелы и переводы строки – это полноправные символы, как буквы и цифры. Они образуют текстовые узлы и становятся частью дерева DOM. Так, в примере выше в теге <head> есть несколько пробелов перед <title>, которые образуют текстовый узел #text (он содержит в себе только перенос строки и несколько пробелов).

Существует всего два исключения из этого правила:

1. По историческим причинам пробелы и перевод строки перед тегом <head> игнорируются

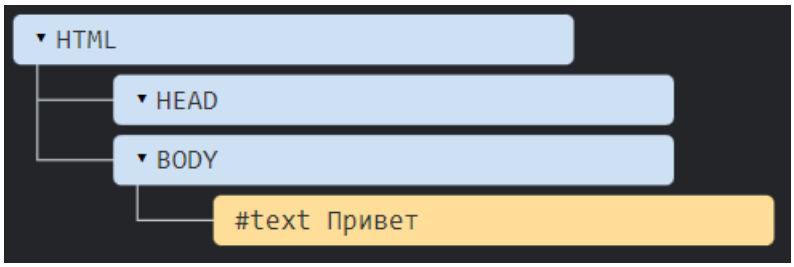
2. Если мы записываем что-либо после закрывающего тега </body>, браузер автоматически перемещает эту запись в конец body, поскольку спецификация HTML требует, чтобы всё содержимое было внутри <body>. Поэтому после закрывающего тега </body> не может быть никаких пробелов.

В остальных случаях всё просто – если в документе есть пробелы (или любые другие символы), они становятся текстовыми узлами дерева DOM, и если мы их удалим, то в DOM их тоже не будет.

Автоисправление

Если браузер сталкивается с некорректно написанным HTML-кодом, он автоматически корректирует его при построении DOM. Например, в начале документа всегда должен быть тег `<html>`. Даже если его нет в документе – он будет в дереве DOM, браузер его создаст. То же самое касается и тега `<body>`.

Например, если HTML-файл состоит из единственного слова `"Привет"`, браузер обернёт его в теги `<html>` и `<body>`, добавит необходимый тег `<head>`, и DOM будет выглядеть так:



При генерации DOM браузер самостоятельно обрабатывает ошибки в документе, закрывает теги и так далее.

Таблицы всегда содержат <tbody>

Важный «особый случай» – работа с таблицами. По стандарту DOM у них должен быть <tbody>, но в HTML их можно написать без него. В этом случае браузер добавляет <tbody> в DOM самостоятельно.

Все, что есть в HTML, даже комментарии, является частью DOM.

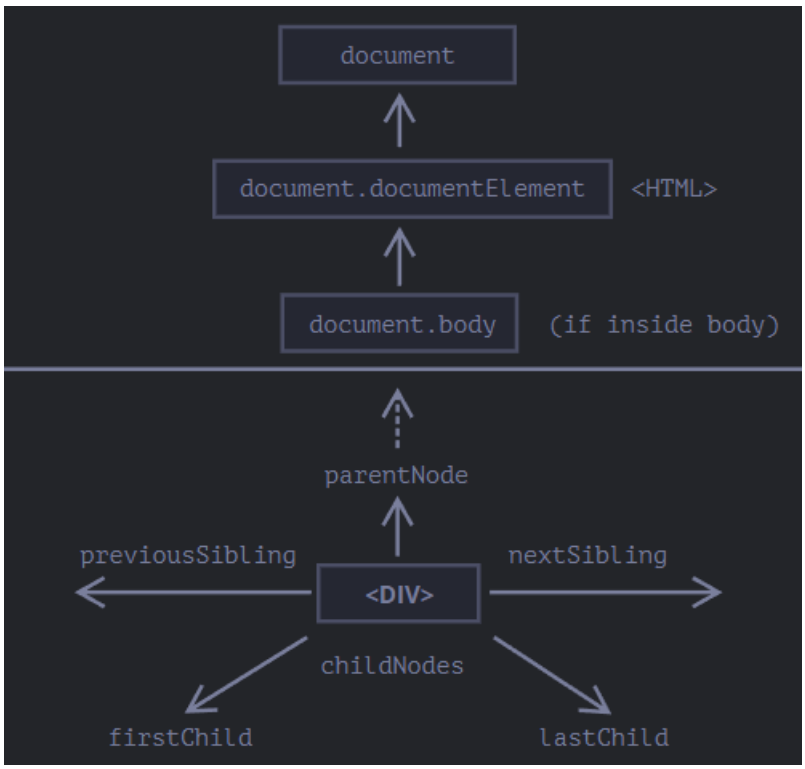
Даже директива `<!DOCTYPE...>`, которую мы ставим в начале HTML, тоже является DOM-узлом. Она находится в дереве DOM прямо перед `<html>`.

Существует 12 типов узлов. Но на практике, в основном, работают с 4 из них:

1. document – «входная точка» в DOM.
2. узлы-элементы – HTML-теги, основные строительные блоки.
3. текстовые узлы – содержат текст.
4. комментарии – иногда в них можно включить информацию, которая не будет показана, но доступна в DOM для чтения JS.

Навигация по DOM-элементам

DOM позволяет нам делать что угодно с элементами и их содержимым, но для начала нужно получить соответствующий DOM-объект. Все операции с DOM начинаются с объекта `document`. Это главная «точка входа» в DOM. Из него мы можем получить доступ к любому узлу. Так выглядят основные ссылки, по которым можно переходить между узлами DOM:



Сверху: documentElement и body

Самые верхние элементы дерева доступны как свойства объекта `document`:

<html> = document.documentElement

Самый верхний узел документа: document.documentElement. В DOM он соответствует тегу <html>.

<body> = document.body

Другой часто используемый DOM-узел – узел тега <body>: document.body.

<head> = document.head

Тег <head> доступен как document.head.

Есть одна тонкость: document.body может быть равен null

Нельзя получить доступ к элементу, которого ещё не существует в момент выполнения скрипта. В частности, если скрипт находится

в <head>, document.body в нём недоступен, потому что браузер его ещё не прочитал.

В мире DOM null означает «не существует»

Дети: childNodes, firstChild, lastChild

- **Дочерние узлы (или дети)** – элементы, которые являются непосредственными детьми узла. Другими словами, элементы, которые лежат непосредственно внутри данного. Например, <head> и <body> являются детьми элемента <html>.
- **Потомки** – все элементы, которые лежат внутри данного, включая детей, их детей и т.д.

DOM-коллекции

Как мы уже видели, childNodes похож на массив. На самом деле это не массив, а *коллекция* – особый перебираемый объект-псевдомассив. И есть два важных следствия из этого:

1. Для перебора *коллекции* мы можем использовать for..of:

```
for (let node of document.body.childNodes) {
  alert(node); // покажет все узлы из коллекции
}

```

Это работает, потому что коллекция является перебираемым объектом (есть требуемый для этого метод Symbol.iterator).

2. Методы массивов не будут работать, потому что коллекция – это не массив:

```
alert(document.body.childNodes.filter); // undefined (у коллекции нет метода filter!)
```

Первый пункт – это хорошо для нас. Второй – бывает неудобен, но можно пережить.

DOM-коллекции – только для чтения

DOM-коллекции, и даже более – *все* навигационные свойства доступны только для чтения. Мы не можем заменить один дочерний узел на другой, просто написав `childNodes[i] = ...`. Для изменения DOM требуются другие методы.

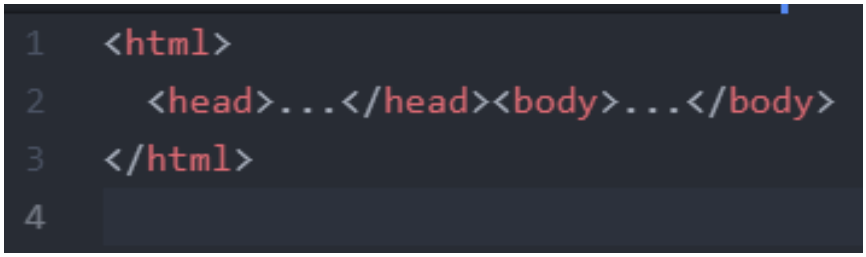
DOM-коллекции живые

Почти все DOM-коллекции, за небольшим исключением, *живые*. Другими словами, они отражают текущее состояние DOM. Если мы сохраним ссылку на `elem.childNodes` и добавим/удалим узлы в DOM, то они появятся в сохранённой коллекции автоматически.

Соседи и родитель

Соседи – это узлы, у которых один и тот же родитель.

Например, здесь `<head>` и `<body>` соседи:



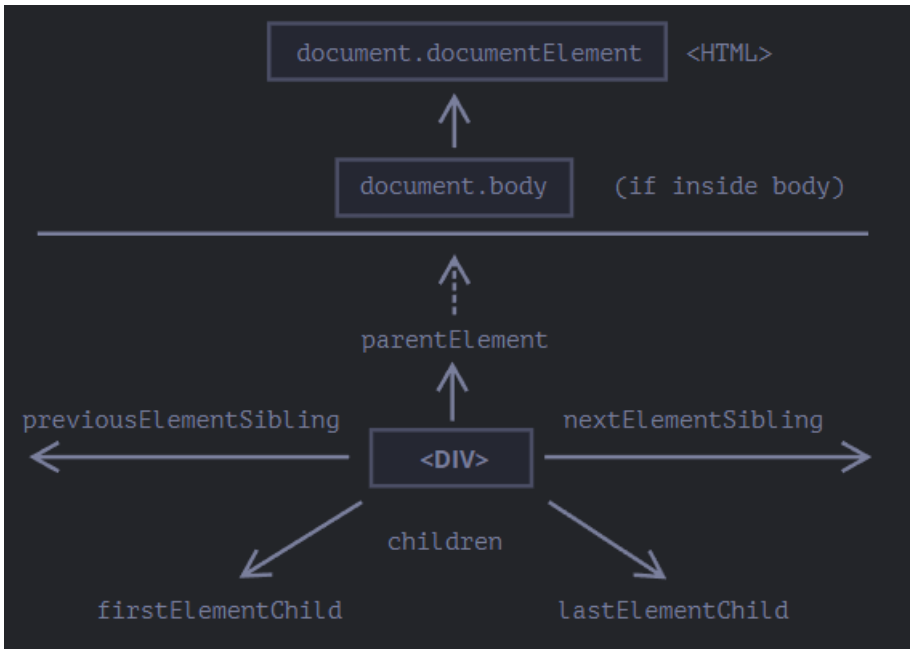
- говорят, что <body> – «следующий» или «правый» сосед <head>
- также можно сказать, что <head> «предыдущий» или «левый» сосед <body>.

Следующий узел того же родителя (следующий сосед) – в свойстве `nextSibling`, а предыдущий – в `previousSibling`. Родитель доступен через `parentNode`. Например:

```
// родителем <body> является <html>
alert( document.body.parentNode === document.documentElement ); // выведет true

// после <head> идёт <body>
alert( document.head.nextSibling ); // HTMLBodyElement

// перед <body> находится <head>
alert( document.body.previousSibling ); // HTMLHeadElement

```

Навигация только по элементам

Навигационные свойства, описанные выше, относятся ко *всем* узлам в документе. В частности, в `childNodes` находятся и текстовые узлы и узлы-элементы и узлы-комментарии, если они есть. Но для большинства задач текстовые узлы и узлы-комментарии нам не нужны. Мы хотим манипулировать узлами-элементами, которые представляют собой теги и формируют структуру страницы. Поэтому давайте рассмотрим дополнительный набор ссылок, которые учитывают только *узлы-элементы*:



Эти ссылки похожи на те, что раньше, только в ряде мест стоит слово `Element`:

- children – коллекция детей, которые являются элементами.

- firstElementChild, lastElementChild – первый и последний дочерний элемент.
- previousElementSibling, nextElementSibling – соседи-элементы.
- parentElement – родитель-элемент.

Получив DOM-узел, мы можем перейти к его ближайшим соседям используя навигационные ссылки. Есть два основных набора ссылок:

- Для всех узлов: parentNode, childNodes, firstChild, lastChild, previousSibling, nextSibling.
- Только для узлов-элементов: parentElement, children, firstElementChild, lastElementChild, previousElementSibling, nextElementSibling.

Некоторые виды DOM-элементов, например, таблицы, предоставляют дополнительные ссылки и коллекции для доступа к своему содержимому. Каждый DOM-узел принадлежит определённому классу. Классы формируют иерархию. Весь набор свойств и методов является результатом наследования.

Главные свойства DOM-узла:

nodeType

Свойство nodeType позволяет узнать тип DOM-узла. Его значение – числовое: 1 для элементов,3 для текстовых узлов, и т.д. Только для чтения.

nodeName/tagName

Для элементов это свойство возвращает название тега (записывается в верхнем регистре, за исключением XML-режима). Для узлов-неэлементов nodeName описывает, что это за узел. Только для чтения.

innerHTML

Внутреннее HTML-содержимое узла-элемента. Можно изменять.

outerHTML

Полный HTML узла-элемента. Запись в elem.outerHTML не меняет elem. Вместо этого она заменяет его во внешнем контексте.

nodeValue/data

Содержимое узла-неэлемента (текст, комментарий). Эти свойства практически одинаковые, обычно мы используем data. Можно изменять.

textContent

Текст внутри элемента: HTML за вычетом всех <тегов>. Запись в него помещает текст в элемент, при этом все специальные символы и теги интерпретируются как текст. Можно использовать для защиты от вставки произвольного HTML кода.

hidden

Когда значение установлено в true, делает то же самое, что и CSS display:none.

В зависимости от своего класса DOM-узлы имеют и другие свойства. Например, у элементов `<input>` (`HTMLInputElement`) есть свойства `value`, `type`, у элементов `<a>` (`HTMLAnchorElement`) есть `href` и т.д. Большинство стандартных HTML-атрибутов имеют соответствующие свойства DOM.

Атрибуты и свойства

Когда браузер загружает страницу, он «читает» HTML и генерирует из него DOM-объекты. Для узлов-элементов большинство стандартных HTML-атрибутов автоматически становятся свойствами DOM-объектов. Например, для такого тега `<body id="page">` у DOM-объекта будет такое свойство `body.id="page"`.

Но преобразование атрибута в свойство происходит не один-в-один!

- Атрибуты – это то, что написано в HTML.
- Свойства – это то, что находится в DOM-объектах.

Небольшое сравнение:

	Свойства	**Атрибуты**
Тип	Любое значение, стандартные свойства имеют типы, описанные в спецификации	Строка
Имя	Имя регистрозависимо	Имя регистронезависимо

Методы для работы с атрибутами:

- elem.hasAttribute(name) – проверить на наличие.
- elem.getAttribute(name) – получить значение.
- elem.setAttribute(name, value) – установить значение.
- elem.removeAttribute(name) – удалить атрибут.
- elem.attributes – это коллекция всех атрибутов.

В большинстве ситуаций предпочтительнее использовать DOM-свойства. Нужно использовать атрибуты только тогда, когда DOM-свойства не подходят, когда нужны именно атрибуты, например:

- Нужен нестандартный атрибут. Но если он начинается с data-, тогда нужно использовать dataset.
- Мы хотим получить именно то значение, которое написано в HTML. Значение DOM-свойства может быть другим, например, свойство href – всегда полный URL, а нам может понадобиться получить «оригинальное» значение.

Изменение документа

Модификации DOM – это ключ к созданию «живых» страниц.

Пример: показать сообщение

Рассмотрим методы на примере – а именно, добавим на страницу сообщение, которое будет выглядеть получше, чем `alert`.

Вот такое:

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<div class="alert">
  <strong>Всем привет!</strong> Вы прочитали важное сообщение.
</div>

```

Это был пример HTML. Теперь давайте создадим такой же `div`, используя JavaScript (предполагаем, что стили в HTML или во внешнем CSS-файле).

Создание элемента

DOM-узел можно создать двумя методами:

document.createElement(tag)

Создаёт новый *элемент* с заданным тегом:

```
let div = document.createElement('div');
```

document.createTextNode(text)

Создаёт новый *текстовый узел* с заданным текстом:

```
let textNode = document.createTextNode('А вот и я');
```

Большую часть времени нам нужно создавать узлы элементов, такие как `div` для сообщения.

Создание сообщения

В нашем случае сообщение – это `div` с классом `alert` и HTML в нём:

```
let div = document.createElement('div');
div.className = "alert";
div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение.";
```

Мы создали элемент, но пока он только в переменной. Мы не можем видеть его на странице, поскольку он не является частью документа.

Методы вставки

Чтобы наш `div` появился, нам нужно вставить его где-нибудь в `document`. Например, в `document.body`. Для этого есть метод `append`, в нашем случае: `document.body.append(div)`. Вот полный пример:

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<script>
  let div = document.createElement('div');
  div.className = "alert";
  div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение.";

  document.body.append(div);
</script>
```

Вот методы для различных вариантов вставки:

- node.append(...nodes or strings) – добавляет узлы или строки в конец node,
- node.prepend(...nodes or strings) – вставляет узлы или строки в начало node,
- node.before(...nodes or strings) –- вставляет узлы или строки до node,
- node.after(...nodes or strings) –- вставляет узлы или строки после node,
- node.replaceWith(...nodes or strings) –- заменяет node заданными узлами или строками.

Удаление узлов

Для удаления узла есть методы `node.remove()`. Например, сделаем так, чтобы наше сообщение удалялось через секунду:

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<script>
  let div = document.createElement('div');
  div.className = "alert";
  div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение.";

  document.body.append(div);
  setTimeout(() => div.remove(), 1000);
</script>
```

Если нам нужно *переместить* элемент в другое место – нет необходимости удалять его со старого.

Все методы вставки автоматически удаляют узлы со старых мест.

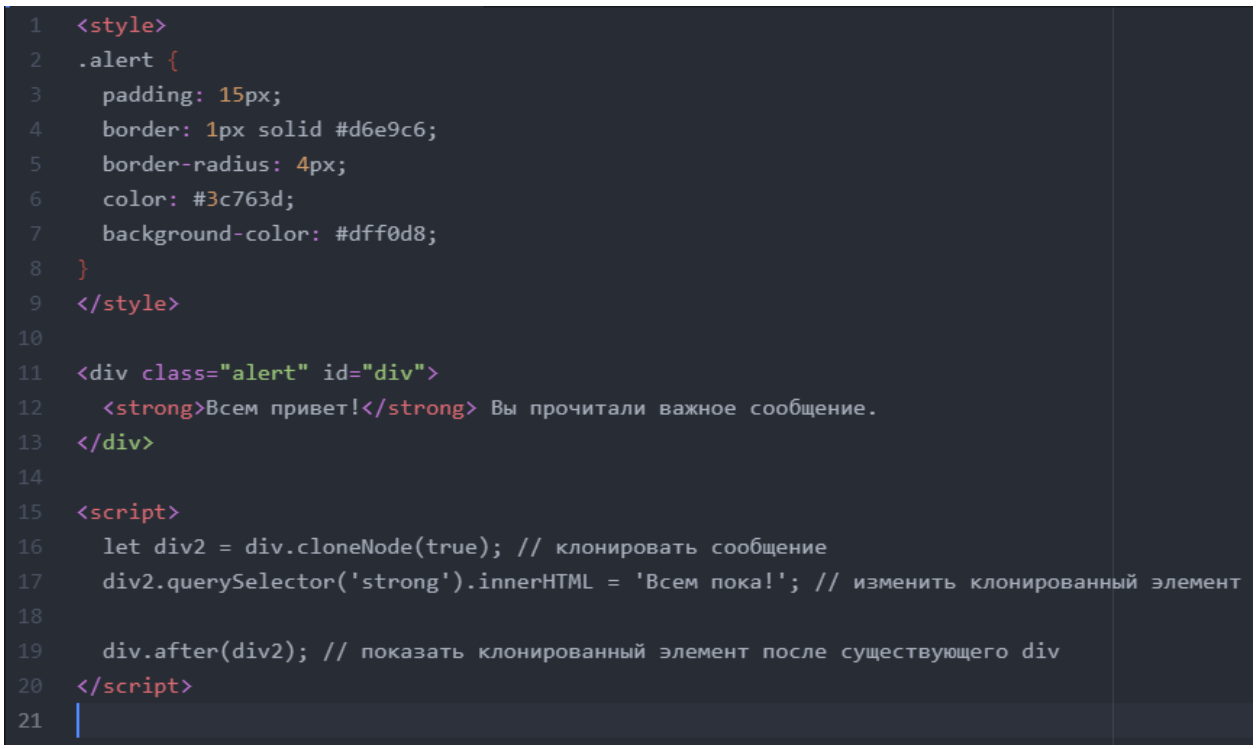
Например, давайте поменяем местами элементы:

```
<div id="first">Первый</div>
<div id="second">Второй</div>
<script>
  // нет необходимости вызывать метод remove
  second.after(first); // берёт #second и после него вставляем #first
</script>
```

Клонирование узлов: cloneNode

- Вызов elem.cloneNode(true) создаёт «глубокий» клон элемента – со всеми атрибутами и дочерними элементами. Если мы вызовем elem.cloneNode(false), тогда клон будет без дочерних элементов.

Пример копирования сообщения:



```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<div class="alert" id="div">
  <strong>Всем привет!</strong> Вы прочитали важное сообщение.
</div>

<script>
  let div2 = div.cloneNode(true); // клонировать сообщение
  div2.querySelector('strong').innerHTML = 'Всем пока!'; // изменить клонированный элемент

  div.after(div2); // показать клонированный элемент после существующего div
</script>
```

- Методы для создания узлов:
    - document.createElement(tag) – создаёт элемент с заданным тегом,
    - document.createTextNode(value) – создаёт текстовый узел (редко используется),
    - elem.cloneNode(deep) – клонирует элемент, если deep==true, то со всеми дочерними элементами.
- Вставка и удаление:
    - node.append(...nodes or strings) – вставляет в node в конец,
    - node.prepend(...nodes or strings) – вставляет в node в начало,
    - node.before(...nodes or strings) – вставляет прямо перед node,
    - node.after(...nodes or strings) – вставляет сразу после node,
    - node.replaceWith(...nodes or strings) – заменяет node.
    - node.remove() – удаляет node.

- Если нужно вставить фрагмент HTML, то `elem.insertAdjacentHTML(where, html)` вставляет в зависимости от `where`:
    - "beforebegin" – вставляет html прямо перед elem,
    - "afterbegin" – вставляет html в elem в начало,
    - "beforeend" – вставляет html в elem в конец,
    - "afterend" – вставляет html сразу после elem.

    Также существуют похожие методы `elem.insertAdjacentText` и `elem.insertAdjacentElement`, они вставляют текстовые строки и элементы, но они редко используются.
- Чтобы добавить HTML на страницу до завершения её загрузки:
    - document.write(html)

    После загрузки страницы такой вызов затирает документ. В основном встречается в старых скриптах.

Стили и классы

Как правило, существует два способа задания стилей для элемента:

1. Создать класс в CSS и использовать его:
2. Писать стили непосредственно в атрибуте style: .

JavaScript может менять и классы, и свойство `style`. Классы – всегда предпочтительный вариант по сравнению со `style`. Мы должны манипулировать свойством `style` только в том случае, если классы «не могут справиться». Например, использование `style` является приемлемым, если

мы вычисляем координаты элемента динамически и хотим установить их из JavaScript:

```
let top = /* сложные расчёты */;
let left = /* сложные расчёты */;

elem.style.left = left; // например, '123px', значение вычисляется во время работы скрипта
elem.style.top = top; // например, '456px'
```

В других случаях, например, чтобы сделать текст красным, добавить значок фона – описываем это в CSS и добавляем класс (JavaScript может это сделать). Это более гибкое и лёгкое в поддержке решение.

Element style

Свойство `elem.style` – это объект, который соответствует тому, что написано в атрибуте `"style"`. Установка стиля `elem.style.width="100px"` работает так же, как наличие в атрибуте `style` строки `width:100px`. Для свойства из нескольких слов используется camelCase:

```
background-color  => elem.style.backgroundColor
z-index           => elem.style.zIndex
border-left-width => elem.style.borderLeftWidth
```

Например:

```
document.body.style.backgroundColor = prompt('background color?', 'green');
```

Свойства с префиксом

Стили с браузерным префиксом, например, `-moz-border-radius`, `-webkit-border-radius` преобразуются по тому же принципу: дефис означает заглавную букву.

Например:

```
button.style.MozBorderRadius = '5px';
button.style.WebkitBorderRadius = '5px';
```

Для управления классами существуют два DOM-свойства:

- className – строковое значение, удобно для управления всем набором классов.
- classList – объект с методами add/remove/toggle/contains, удобно для управления отдельными классами.

Чтобы изменить стили:

- Свойство `style` является объектом со стилями в формате camelCase. Чтение и запись в него работают так же, как изменение соответствующих свойств в атрибуте `"style"`. Чтобы узнать, как добавить в него `important` и делать некоторые другие редкие вещи – смотрите документацию.
- Свойство `style.cssText` соответствует всему атрибуту `"style"`, полной строке стилей.

Для чтения окончательных стилей (с учётом всех классов, после применения CSS и вычисления окончательных значений) используется:

- Метод getComputedStyle(elem, [pseudo]) возвращает объект, похожий по формату на style. Только для чтения.

Практическое задание

1. В элементе уведомлений, созданном ранее, добавьте появление новых уведомлений каждые 3 секунды (счетчик новых уведомлений должен обновляться, содержание элемента на ваше усмотрение).

2. При нажатии на кнопку уведомлений, с использованием задерживающего декоратора, остановите счетчик новых уведомлений на 10 секунд.

3. Напишите интерфейс для создания списка.
Для каждого пункта:

1. Запрашивайте содержимое пункта у пользователя с помощью prompt.
2. Создавайте элемент <li> и добавляйте его к <ul>.
3. Продолжайте до тех пор, пока пользователь не отменит ввод (нажатием клавиши Esc или введя пустую строку).
4. Все элементы должны создаваться динамически.
5. Если пользователь вводит HTML-теги, они должны обрабатываться как текст.

4. Напишите функцию showNotification(options), которая создаёт уведомление: с заданным содержимым. Уведомление должно автоматически исчезнуть через 1,5 секунды.