

### 1. Функции `setTimeout` and `setInterval`.

Можно вызвать функцию не в данный момент, а позже, через заданный интервал времени. Это называется «планирование вызова». Для этого существует 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.

#### Метод `setTimeout`

Синтаксис:

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

Параметры:

`func|code` – функция или строка кода для выполнения. Обычно, это функция. 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() {  
    alert('Привет');  
}  
  
setTimeout(sayHi, 1000);
```

С аргументами:

```
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. Это также будет работать:

```
setTimeout("alert('Привет')", 1000);
```

Но использование строк не рекомендуется. Вместо этого используйте функции. Например, так:

```
setTimeout(() => alert('Привет'), 1000);
```

Передавайте функцию, но не запускайте её. Начинающие разработчики иногда ошибаются, добавляя скобки () после функции:

```
// не правильно!  
setTimeout(sayHi(), 1000);
```

Это не работает, потому что `setTimeout` ожидает ссылку на функцию. Здесь `sayHi()` запускает выполнение функции и результат выполнения отправляется в `setTimeout`. В нашем случае результатом выполнения `sayHi()` является `undefined` (так как функция ничего не возвращает), поэтому ничего не планируется.

### 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:

```
let timerId = setTimeout(() => alert("ничего не происходит"), 1000);  
alert(timerId); // идентификатор таймера  
  
clearTimeout(timerId);  
alert(timerId); // тот же идентификатор
```

Как видно из вывода `alert`, в браузере идентификатором таймера является число. В других средах это может быть что-то ещё. Например, `Node.js` возвращает объект таймера с дополнительными методами.

### Метод `setInterval`

Метод `setInterval` имеет такой же синтаксис как `setTimeout`:

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

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);
```

В большинстве браузеров, включая Chrome и Firefox внутренний счётчик продолжает тикать во время показа `alert/confirm/prompt`. Так что если запустить код выше и подождать с закрытием `alert` несколько секунд, то следующий `alert` будет показан сразу, как вы его закроете. Интервал времени между сообщениями `alert` будет короче, чем 2 секунды.

### Рекурсивный `setTimeout`

Есть два способа запускать что-то регулярно. Один из них `setInterval`. Другим является рекурсивный `setTimeout`. Например:

```
/* вместо:
let timerId = setInterval(() => alert('tick'), 2000);
*/

let timerId = setTimeout(function tick() {
    alert('tick');
    timerId = setTimeout(tick, 2000); // (*)
}, 2000);
```

Метод `setTimeout` выше планирует следующий вызов прямо после окончания, текущего (\*). 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. Например:

```
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);
```

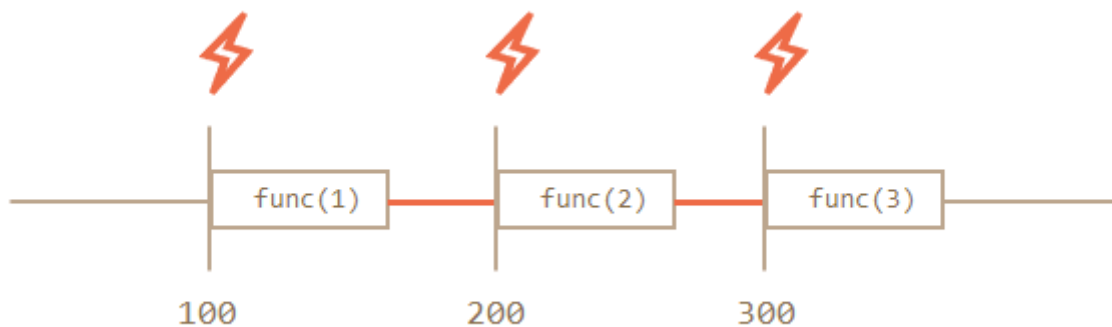
Рекурсивный `setTimeout` позволяет задать задержку между выполнениями более точно, чем `setInterval`. Сравним два фрагмента кода. Первый использует `setInterval`:

```
let i = 1;
setInterval(function() {
  func(i);
}, 100);
```

Второй использует рекурсивный `setTimeout`:

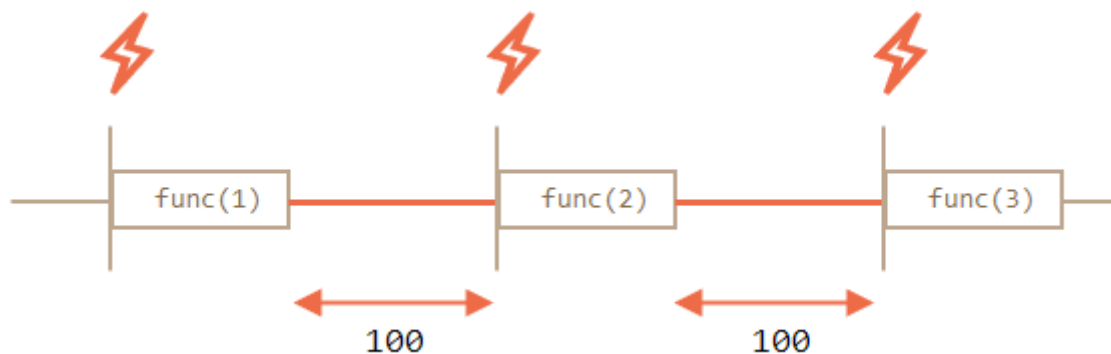
```
let i = 1;
setTimeout(function run() {
  func(i);
  setTimeout(run, 100);
}, 100);
```

Для `setInterval` внутренний планировщик выполнит `func(i)` каждые 100 мс:



Реальная задержка между `func` для `setInterval` меньше, чем видно из кода. Это нормально, потому что время, затраченное на выполнение `func`, «потребляет» часть заданного интервала времени. Вполне возможно, что выполнение `func` будет дольше, чем ожидается, и займёт более 100 мс. В данном случае движок ждёт окончания выполнения `func` и затем проверяет планировщик и, если время истекло, немедленно запускает его снова. В крайнем случае, если функция всегда выполняется дольше, чем задержка `delay`, то вызовы будут выполняться без задержек вовсе.

Ниже представлено изображение, показывающее процесс работы рекурсивного `setTimeout`:



Рекурсивный `setTimeout` гарантирует фиксированную задержку (здесь 100 мс). Это потому, что новый вызов планируется в конце предыдущего.

Когда функция передаётся в `setInterval/setTimeout`, на неё создаётся внутренняя ссылка и сохраняется в планировщике. Это предотвращает попадание функции в сборщик мусора, даже если на неё нет других ссылок.

```
// функция остаётся в памяти до тех пор, пока планировщик обращается к ней
setTimeout(function() {...}, 100);
```

Для `setInterval` функция остаётся в памяти до тех, пока не будет вызван `clearInterval`. Есть и побочный эффект. Функция ссылается на внешнее лексическое окружение, поэтому пока она существует, внешние переменные существуют тоже. Они могут занимать больше памяти, чем сама функция. Поэтому, если регулярный вызов функции больше не нужен, то лучше отменить его, даже если функция очень маленькая.

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. Так вызов функции будет запланирован сразу после выполнения текущего кода.

Например, этот код выводит «Привет» и затем сразу «Мир»:

```
setTimeout(() => alert("Мир"));

alert("Привет");
```

Первая строка «помещает вызов в календарь через 0 мс». Но планировщик «проверит календарь» после того, как текущий код завершится. Поэтому "Привет" выводится первым, а "Мир" после него.

В браузере есть ограничение, как часто внутренние счётчики могут выполняться. В стандарте HTML5 говорится: «после пяти вложенных таймеров интервал должен составлять не менее четырёх миллисекунд.»

Продемонстрируем в примере ниже, что это означает. Вызов `setTimeout` повторно вызывает себя через 0 мс. Каждый вызов запоминает реальное время

от предыдущего вызова в массиве `times`. Посмотрим какова реальная задержка:

```
let start = Date.now();
let times = [];

setTimeout(function run() {
    times.push(Date.now() - start);

    if (start + 100 < Date.now()) alert(times);
    else setTimeout(run);
});
```

Первый таймер запускается сразу (как и указано в спецификации) и затем начинается задержка и вывод 9, 15, 20, 24. Аналогичное происходит при использовании `setInterval` вместо `setTimeout`: `setInterval(f)` запускает `f` несколько раз с нулевой задержкой, а затем с задержкой 4+ мс. Это ограничение существует давно, многие скрипты полагаются на него, поэтому оно сохраняется по историческим причинам. Этого ограничения нет в серверном JavaScript. Там есть и другие способы планирования асинхронных задач. Например, `setImmediate` для Node.js. Так что это ограничение относится только к браузерам.

## 2. Прототипное наследование. Собственные и унаследованные свойства.

В программировании часто возникает необходимость что-то расширить. Например, есть объект `user` со своими свойствами и методами, надо создать объекты `admin` и `guest` как его слегка изменённые варианты. Хотелось 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.

### Свойство `[[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».

Если при чтении свойство из `object` отсутствует, JavaScript автоматически берет его из прототипа. В программировании такой механизм называется *прототипным наследованием*. Многие возможности языка и техники программирования основываются на нем.

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;

```

Свойство `__proto__` – не то же самое, что `[[Prototype]]`. Это геттер/сеттер для него. Он существует по историческим причинам, в современном языке его заменяют функции `Object.getPrototypeOf/Object.setPrototypeOf`, которые также получают/устанавливают прототип. По спецификации `__proto__` должен поддерживаться только браузерами, но по факту все среды, включая серверную, поддерживают его.

В примере ниже осуществляется поиск свойства в `rabbit`, а оно отсутствует, и JavaScript автоматически берет его из `animal`:

```

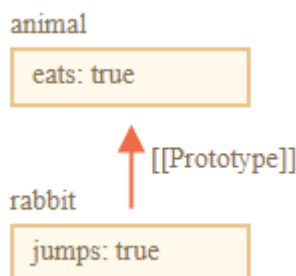
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal; // (*)

alert( rabbit.eats ); // true (**)
alert( rabbit.jumps ); // true

```

Здесь строка (\*) устанавливает `animal` как прототип для `rabbit`. Затем, когда `alert` пытается прочесть свойство `rabbit.eats` (\*\*), его нет в `rabbit`, поэтому JavaScript следует по ссылке `[[Prototype]]` и находит её в `animal` (смотрите снизу вверх):



Здесь можно сказать, что `animal` является прототипом `rabbit` или `rabbit` прототипно наследует от `animal`. Так что если у `animal` много полезных свойств и методов, то они автоматически становятся доступными у `rabbit`.

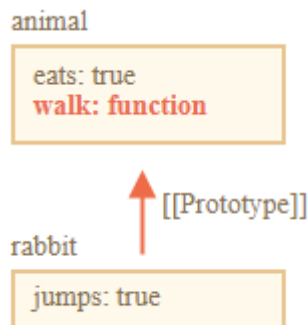
Такие свойства называются *унаследованными*. Например, есть метод в `animal`, он может быть вызван на `rabbit`:

```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

rabbit.walk(); // Animal walk
```

Метод автоматически берётся из прототипа:



Цепочка прототипов может быть длиннее:

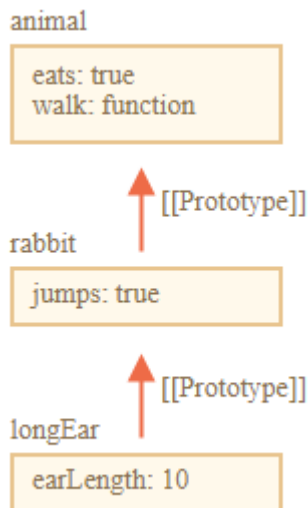
```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

let longEar = {
  earLength: 10,
  __proto__: rabbit
};

longEar.walk(); // Animal walk
alert(longEar.jumps); // true (для rabbit)
```





Есть только два ограничения:

1. Ссылки не могут идти по кругу. JavaScript выдаст ошибку, если попытаться назначить `__proto__` по кругу.
2. Значение `__proto__` может быть объектом или `null`. Другие типы игнорируются.

Это вполне очевидно, но все же: может быть только один `[[Prototype]]`. Объект не может наследовать от двух других.

Прототип используется только для чтения свойств. Операции записи/удаления работают напрямую с объектом. В приведённом ниже примере присваивается `rabbit` собственный метод `walk`:

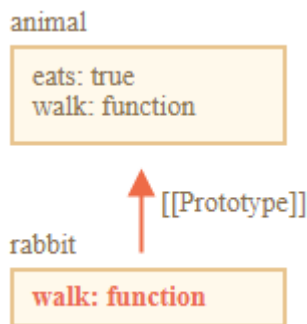
```
let animal = {
  eats: true,
  walk() {
    /* ... */
  }
};

let rabbit = {
  __proto__: animal
};

rabbit.walk = function() {
  alert("Rabbit! Bounce-bounce!");
};

rabbit.walk(); // Rabbit! Bounce-bounce!
```

Теперь вызов `rabbit.walk()` находит метод непосредственно в объекте и выполняет его, не используя прототип:



Свойства-аксессоры – исключение, так как запись в него обрабатывается функцией-сеттером. То есть, это, фактически, вызов функции. По этой причине `admin.fullName` работает корректно в приведённом ниже коде:

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

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

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

let admin = {
  __proto__: user,
  isAdmin: true
};

alert(admin.fullName); // John Smith (*)

// срабатывает сеттер!
admin.fullName = "Alice Cooper"; // (**)
```

Здесь в строке (\*) свойство `admin.fullName` имеет геттер в прототипе `user`, поэтому вызывается он. В строке (\*\*) свойство также имеет сеттер в прототипе, который и будет вызван.

### Значение «this»

Прототипы никак не влияют на `this`. Неважно, где находится метод: в объекте или его прототипе. При вызове метода `this` – всегда объект перед точкой. Таким образом, вызов сеттера `admin.fullName` в качестве `this` использует `admin`, а не `user`.

Это на самом деле очень важная деталь, потому что может быть большой объект со множеством методов, от которого можно наследовать. Затем

наследующие объекты могут вызывать его методы, но они будут изменять состояние этих объектов, а не большого. Например, здесь `animal` представляет собой «хранилище методов», и `rabbit` использует его. Вызов `rabbit.sleep()` устанавливает `this.isSleeping` для объекта `rabbit`:

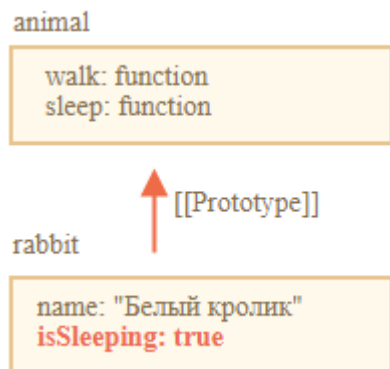
```
let animal = {
  walk() {
    if (!this.isSleeping) {
      alert(`I walk`);
    }
  },
  sleep() {
    this.isSleeping = true;
  }
};

let rabbit = {
  name: "White Rabbit",
  __proto__: animal
};

rabbit.sleep();

alert(rabbit.isSleeping); // true
alert(animal.isSleeping); // undefined (нет такого свойства в прототипе)
```

Картинка с результатом:



Если бы были другие объекты, такие как `bird`, `snake` и т.д., унаследованные от `animal`, они также получили бы доступ к методам `animal`. Но `this` при вызове каждого метода будет соответствовать объекту, на котором происходит вызов (перед точкой), а не `animal`. Поэтому, когда записываются данные в `this`, они сохраняются в этих объектах. В результате методы являются общими, а состояние объекта — нет.

## Цикл `for...in`

Цикл `for..in` проходит не только по собственным, но и по унаследованным свойствам объекта. Например:

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

let rabbit = {
  jumps: true,
  __proto__: animal
};

alert(Object.keys(rabbit)); // jumps

for(let prop in rabbit) alert(prop); // jumps, then eats
```

Если унаследованные свойства не нужны, то можно отфильтровать их при помощи встроенного метода `obj.hasOwnProperty(key)`: он возвращает `true`, если у `obj` есть собственное, не унаследованное, свойство с именем `key`. Пример такой фильтрации:

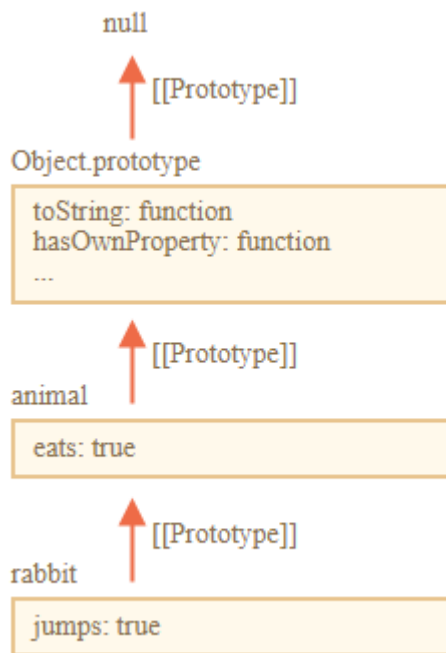
```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

for(let prop in rabbit) {
  let isOwn = rabbit.hasOwnProperty(prop);

  if (isOwn) {
    alert(`Our: ${prop}`); // Our: jumps
  } else {
    alert(`Inherited: ${prop}`); // Inherited: eats
  }
}
```

В этом примере цепочка наследования выглядит так: `rabbit` наследует от `animal`, который наследует от `Object.prototype` (так как `animal` – литеральный объект `{...}`, это по умолчанию), а затем `null` на самом верху:



Стоит отметить следующее: метод `rabbit.hasOwnProperty` явно не определен. Если посмотреть на цепочку прототипов, то видно, что он берётся из `Object.prototype.hasOwnProperty`. То есть, он унаследован, но не появляется в цикле `for..in`, в отличие от `eats` и `jumps`. Дело в том, что это свойство не перечислимо. То есть, у него внутренний флаг `enumerable` стоит `false`, как и у других свойств `Object.prototype`. Поэтому оно и не появляется в цикле.

Почти все методы, получающие ключи/значения, такие как `Object.keys`, `Object.values` и другие — игнорируют унаследованные свойства. Они учитывают только свойства самого объекта, не его прототипа.

### 3. Свойство `F.prototype`.

Как известно, новые объекты могут быть созданы с помощью функции-конструктора, `new F()`. Если в `F.prototype` содержится объект, оператор `new` устанавливает его в качестве `[[Prototype]]` для нового объекта.

JavaScript использовал прототипное наследование с момента своего появления. Это одна из основных особенностей языка.

Но раньше, прямого доступа к прототипу объекта не было. Надёжно работало только свойство `"prototype"` функции-конструктора. Поэтому оно используется во многих скриптах. Обратите внимание, что `F.prototype` означает обычное свойство с именем `"prototype"` для `F`. Это ещё не «прототип объекта», а обычное свойство `F` с таким именем. Приведём пример:

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

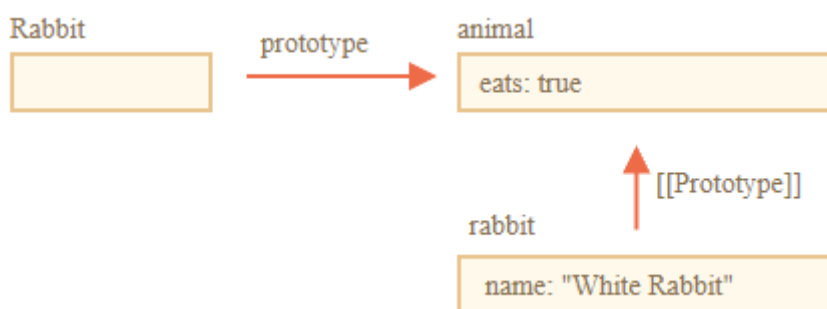
```
function Rabbit(name) {
  this.name = name;
}

Rabbit.prototype = animal;

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

alert( rabbit.eats ); // true
```

Установка `Rabbit.prototype = animal` буквально говорит интерпретатору следующее: "При создании объекта через `new Rabbit()` запиши ему `animal` в `[[Prototype]]`". Результат будет выглядеть так:



На изображении: "prototype" – горизонтальная стрелка, обозначающая обычное свойство для "F", а `[[Prototype]]` – вертикальная, обозначающая наследование `rabbit` от `animal`.

`F.prototype` используется только при вызове `new F()` и присваивается в качестве свойства `[[Prototype]]` нового объекта. После этого `F.prototype` и новый объект ничего не связывает. После создания `F.prototype` может измениться, и новые объекты, созданные с помощью `new F()`, будут иметь другой объект в качестве `[[Prototype]]`, но уже существующие объекты сохраняют старый.

У каждой функции по умолчанию уже есть свойство "prototype". По умолчанию "prototype" – объект с единственным свойством `constructor`, которое ссылается на функцию-конструктор. Вот такой:

```
function Rabbit() {}

/* прототип по умолчанию
Rabbit.prototype = { constructor: Rabbit };
*/
```



Проверим это:

```
function Rabbit() {}
// по умолчанию:
// Rabbit.prototype = { constructor: Rabbit }

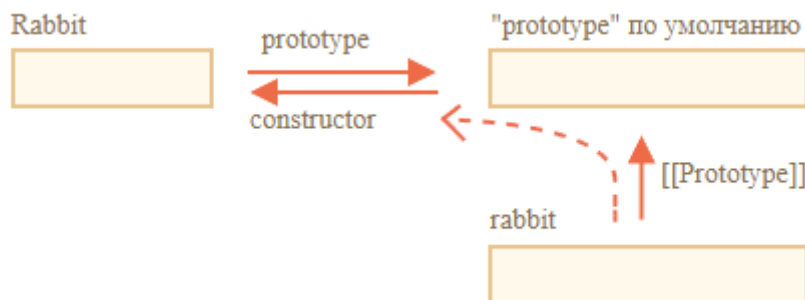
alert( Rabbit.prototype.constructor == Rabbit ); // true
```

Соответственно, если ничего не меняется, то свойство `constructor` будет доступно всем кроликам через `[[Prototype]]`:

```
function Rabbit() {}
// по умолчанию:
// Rabbit.prototype = { constructor: Rabbit }

let rabbit = new Rabbit(); // наследует от {constructor: Rabbit}

alert(rabbit.constructor == Rabbit); // true (свойство получено из
прототипа)
```



Можно использовать свойство `constructor` существующего объекта для создания нового. Пример:

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

let rabbit = new Rabbit("White Rabbit");

let rabbit2 = new rabbit.constructor("Black Rabbit");
```

Это удобно, когда есть объект, но неизвестно какой конструктор использовался для его создания (например, он был взят из сторонней библиотеки), а необходимо создать ещё один такой объект.

Самое важное о свойстве `"constructor"` это то, что JavaScript сам по себе не гарантирует правильное значение свойства `"constructor"`. Оно является свойством по умолчанию в `"prototype"` у функций, но что будет с ним позже — зависит только от разработчика. В частности, если заменить прототип по умолчанию на другой объект, свойства `"constructor"` в нём не будет. Например:

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

```
Rabbit.prototype = {
  jumps: true
};

let rabbit = new Rabbit();
alert(rabbit.constructor === Rabbit); // false
```

Таким образом, чтобы сохранить верное свойство "constructor", надо добавлять/удалять/изменять свойства у прототипа по умолчанию вместо того, чтобы перезаписывать его целиком. В примере ниже Rabbit.prototype не перезаписывается полностью, а добавляется к нему свойство. Прототип по умолчанию сохраняется, и сохраняется доступ к Rabbit.prototype.constructor.

```
function Rabbit() {}

Rabbit.prototype.jumps = true
```

Или можно заново создать свойство constructor:

```
Rabbit.prototype = {
  jumps: true,
  constructor: Rabbit
};
```

#### 4. Встроенные прототипы.

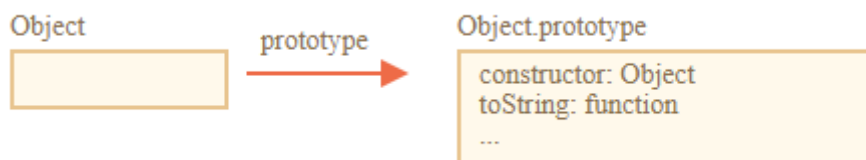
Свойство "prototype" широко используется в внутри самого языка JavaScript. Все встроенные функции-конструкторы используют его.

##### Object.prototype

Выведем пустой объект:

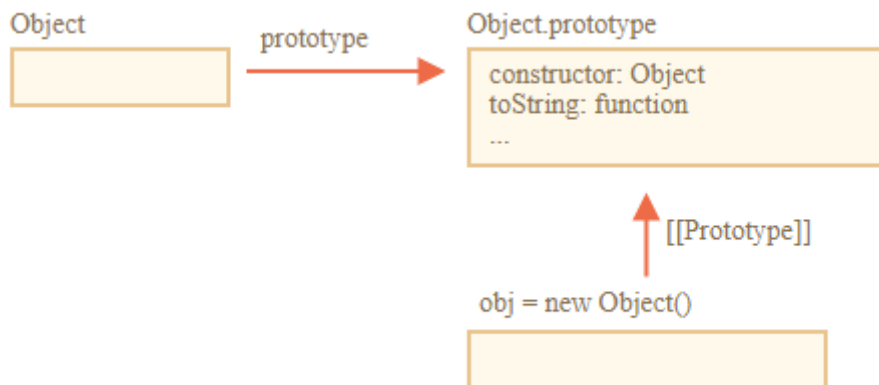
```
let obj = {};
alert( obj ); // "[object Object]" ?
```

В рассматриваемом примере нету кода, который генерирует строку "[object Object]". Понятно, что это встроенный метод toString, но явно не видно, где он объявлен, ведь obj пуст. Дело в том, что краткая нотация obj = {} это то же самое, что и obj = new Object(), где Object – встроенная функция-конструктор для объектов с собственным свойством prototype, который ссылается на огромный объект с методом toString и другими. Вот что происходит:





Когда вызывается `new Object()` (или создаётся объект с помощью литерала `{...}`), свойство `[[Prototype]]` этого объекта устанавливается на `Object.prototype` по правилам, которые рассматривались в предыдущем вопросе:



Таким образом, когда вызывается `obj.toString()`, метод берётся из `Object.prototype`. Можно проверить это так:

```
let obj = {};  
  
alert(obj.__proto__ === Object.prototype); // true  
// obj.toString === obj.__proto__.toString ==  
Object.prototype.toString
```

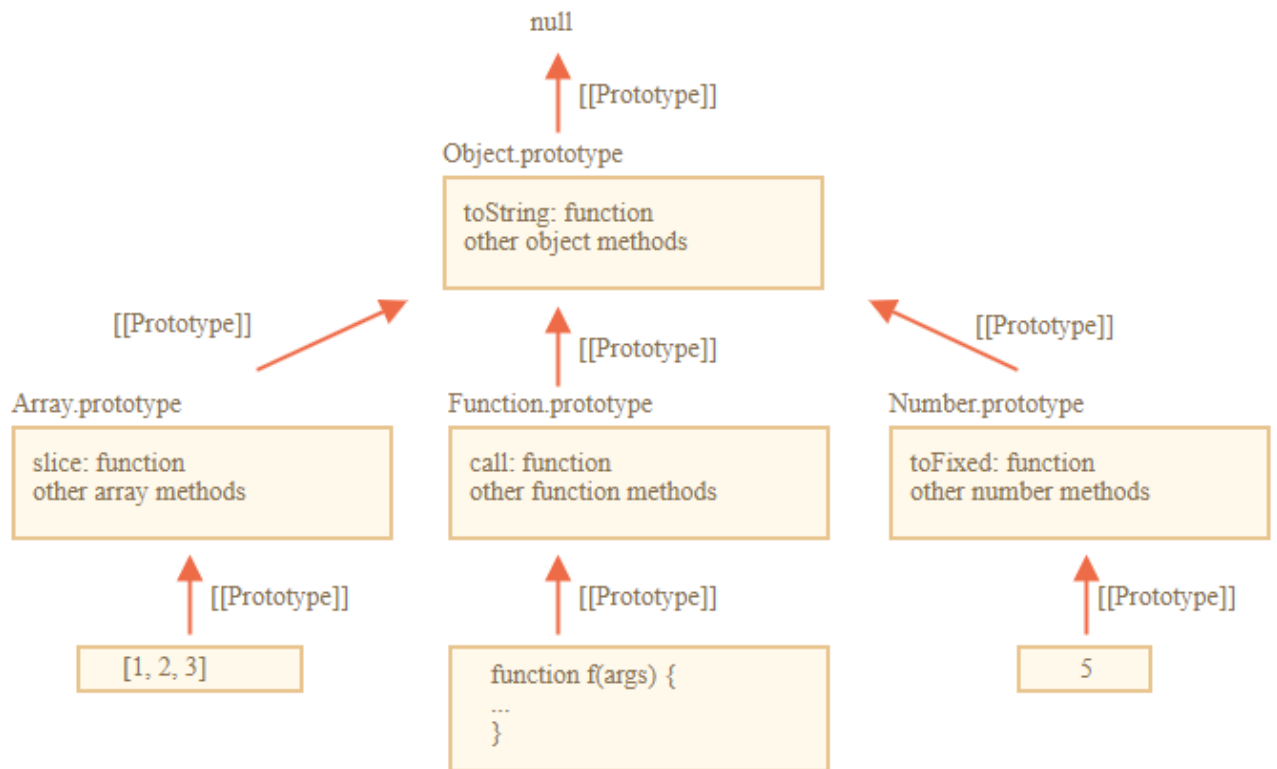
Обратим внимание, что выше `Object.prototype` по цепочке прототипов больше нет `[[Prototype]]`:

```
alert(Object.prototype.__proto__); // null
```

Другие встроенные объекты, такие как `Array`, `Date`, `Function` и другие, также хранят свои методы в прототипах. Например, при создании массива `[1, 2, 3]` внутренне используется конструктор массива `Array`. Поэтому прототипом массива становится `Array.prototype`, предоставляя ему свои методы. Это позволяет эффективно использовать память.

Согласно спецификации, наверху иерархии встроенных прототипов находится `Object.prototype`. Поэтому иногда говорят, что «всё наследует от объектов».

Вот более полная картина (для 3 встроенных объектов):



Проверим прототипы:

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

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

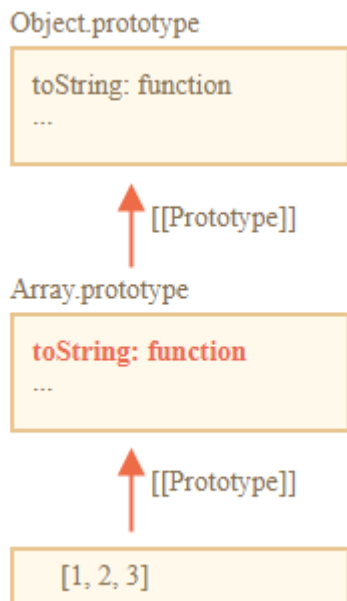
// затем от Object.prototype?
alert( arr.__proto__.__proto__ === Object.prototype ); // true

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

Некоторые методы в прототипах могут пересекаться, например, у `Array.prototype` есть свой метод `toString`, который выводит элементы массива через запятую:

```
let arr = [1, 2, 3]
alert(arr); // 1,2,3 <-- результат Array.prototype.toString
```

Как известно, у `Object.prototype` есть свой метод `toString`, но так как `Array.prototype` ближе в цепочке прототипов, то берётся именно вариант для массивов:



В браузерных инструментах, таких как консоль разработчика, можно посмотреть цепочку наследования (возможно, потребуется использовать `console.dir` для встроенных объектов):

```
> console.dir([1,2,3])
▼ Array[3] ⓘ
  0: 1
  1: 2
  2: 3
  length: 3
  ▼ __proto__: Array.prototype
    ► concat: function concat() { [native code] }
    ► ...
    ► unshift: function unshift() { [native code] }
    ▼ __proto__: Object.prototype
      ► ...
      ► constructor: function Object() { [native code] }
      ► hasOwnProperty: function hasOwnProperty() { [native code] }
      ► isPrototypeOf: function isPrototypeOf() { [native code] }
      ► ...
```

Другие встроенные объекты устроены аналогично. Даже функции – объекты встроенного конструктора `Function`, и все их методы (`call`/`apply` и другие) берутся из `Function.prototype`. Также у функций есть свой метод `toString`.

```
function f() {}
```

```
alert(f.__proto__ == Function.prototype); // true
alert(f.__proto__.__proto__ == Object.prototype); // true, наследует
от Object
```

## Примитивы

Самое сложное происходит со строками, числами и булевым типом. Как известно, они не объекты. Но если попытаться получить доступ к их свойствам, тогда будет создан временный объект-обёртка с использованием встроенных конструкторов String, Number, Boolean, который предоставит методы и после чего исчезнет. Эти объекты создаются невидимо для нас, и большая часть движков оптимизирует этот процесс, но спецификация описывает это именно таким образом. Методы этих объектов также находятся в прототипах, доступных как String.prototype, Number.prototype и Boolean.prototype.

Специальные 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. Например, если добавить метод к String.prototype, метод становится доступен для всех строк:

```
String.prototype.show = function() {  
    alert(this);  
};  
  
"BOOM!".show(); // BOOM!
```

В течение процесса разработки могут возникнуть идеи о новых встроенных методах, которые хотелось бы иметь и добавить их во встроенные прототипы. Это плохая идея. Прототипы глобальны, поэтому очень легко могут возникнуть конфликты. Если две библиотеки добавляют метод String.prototype.show, то одна из них перепишет метод другой.

В современном программировании есть только один случай, в котором одобряется изменение встроенных прототипов. Это создание полифилов. Полифил – это термин, который означает замену метода, который существует в спецификации JavaScript, но он ещё не поддерживается текущим движком JavaScript. Тогда можно реализовать его и добавить его во встроенный прототип. Например:

```
if (!String.prototype.repeat) { // Если такого метода нет  
    // добавляем его в прототип  
  
    String.prototype.repeat = function(n) {  
        // повторить строку n раз  
  
        return new Array(n + 1).join(this);  
    };  
}  
  
alert( "La".repeat(3) ); // LaLaLa
```

## Заемствование у прототипов

Ранее рассматривалось заимствование методов. Это когда метод из одного объекта копируется в другой. Некоторые методы встроенных прототипов часто одалживают. Например, если создать объект, похожий на массив (псевдомассив), можно скопировать некоторые методы из Array в этот объект. Пример:

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

obj.join = Array.prototype.join;

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

Это работает, потому что для внутреннего алгоритма встроенного метода join важна только корректность индексов и свойства length, он не проверяет является ли объект на самом деле массивом. И многие встроенные методы работают так же. Альтернативная возможность – можно унаследовать от массива, установив obj.\_\_proto\_\_ как Array.prototype, таким образом все методы Array станут автоматически доступны в obj. Но это будет невозможно, если obj уже наследует от другого объекта, ведь можно наследовать только от одного объекта одновременно.

Заемствование методов – гибкий способ, позволяющий смешивать функциональность разных объектов по необходимости.

## 5. Методы прототипов.

Ранее упоминалось, что существуют современные методы работы с прототипами. Свойство \_\_proto\_\_ считается устаревшим, и по стандарту должно поддерживаться только браузерами. Современные методы это:

- Object.create(proto, [descriptors]) – создаёт пустой объект со свойством [[Prototype]], указанным как proto, и необязательными дескрипторами свойств descriptors.
- Object.getPrototypeOf(obj) – возвращает свойство [[Prototype]] объекта obj.
- Object.setPrototypeOf(obj, proto) – устанавливает свойство [[Prototype]] объекта obj как proto.

Эти методы нужно использовать вместо \_\_proto\_\_. Например:

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

```
let rabbit = Object.create(animal);

alert(rabbit.eats); // true

alert(Object.getPrototypeOf(rabbit) === animal);

Object.setPrototypeOf(rabbit, {});
```

У `Object.create` есть необязательный второй аргумент: дескрипторы свойств. Можно добавить дополнительное свойство новому объекту таким образом:

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

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

alert(rabbit.jumps); // true
```

Также можно использовать `Object.create` для глубокого клонирования объекта, более мощного, чем копирование свойств в цикле `for..in`:

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

Такой вызов создаёт точную копию объекта `obj`, включая все свойства: перечисляемые и не перечисляемые, свойства, геттеры/сеттеры для свойств – и все это с правильным свойством `[[Prototype]]`.

Технически, можно установить/получить `[[Prototype]]` в любое время. Но обычно прототип устанавливается только раз во время создания объекта, а после не меняется: `rabbit` наследует от `animal`, и это не изменится.

JavaScript движки хорошо оптимизированы для этого. Изменение прототипа «на лету» с помощью `Object.setPrototypeOf` или `obj.__proto__ =` – очень медленная операция, которая ломает внутренние оптимизации для операций доступа к свойствам объекта. Так что лучше избегайте этого, кроме тех случаев, когда знаете, что делаете, либо скорость JavaScript для вас не имеет никакого значения.

Как известно, объекты можно использовать как ассоциативные массивы для хранения пар ключ/значение. Но если попробовать хранить созданные пользователями ключи (например, словари с пользовательским вводом), можно заметить интересный сбой: все ключи работают как ожидается, за исключением `"__proto__"`. Например:

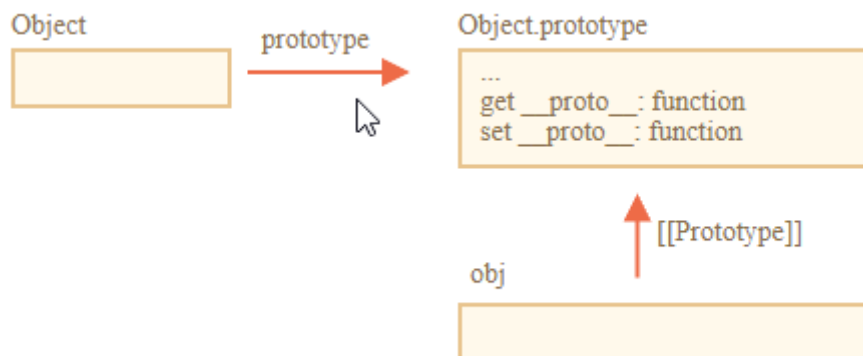
```
let obj = {};
```

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

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

Если пользователь введёт `__proto__`, присвоение проигнорируется, так как свойство `__proto__` должно быть либо объектом, либо `null`, а строка не может стать прототипом.

Свойство `__proto__` – не обычное, а аксессор, заданный в `Object.prototype`:



Так что при чтении или установке `obj.__proto__` вызывается соответствующий геттер/сеттер из прототипа `obj`, и именно он устанавливает/получает свойство `[[Prototype]]`.

Поните, что `__proto__` – это способ доступа к свойству `[[Prototype]]`, это не само свойство `[[Prototype]]`.

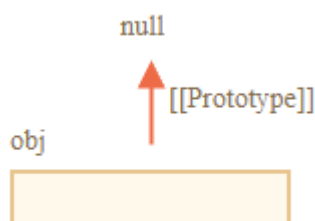
Теперь, если надо использовать объект как ассоциативный массив, можно сделать это следующим образом:

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

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

```
alert(obj[key]); // "some value"
```

`Object.create(null)` создаёт пустой объект без прототипа (`[[Prototype]]` будет `null`):



Таким образом не будет унаследованного геттера/сеттера для `__proto__`. Теперь это свойство обрабатывается как обычное свойство, и приведённый выше пример работает правильно. Можно назвать такой объект «простейшим» или «чистым словарным объектом», потому что он ещё проще чем обычный объект `{...}`. Недостаток в том, что у таких объектов не будет встроенных методов объекта, таких как `toString`:

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

alert(obj); // Error (no toString)
```

Но обычно это нормально для ассоциативных массивов. Обратите внимание, что большая часть методов, связанных с объектами, имеют вид `Object.something(...)`. К примеру, `Object.keys(obj)` не находится в прототипе, так что они продолжают работать для таких объектов:

```
let chineseDictionary = Object.create(null);
chineseDictionary.hello = "你好";
chineseDictionary.bye = "再见";

alert(Object.keys(chineseDictionary)); // hello,bye
```

## 6. Классы. Class Expression. Приватные и защищённые методы и свойства.

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

На практике часто надо создавать много объектов одного вида, например пользователей, товары или что-то еще. Как известно, 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:

```
class MyClass {
  // методы класса
  constructor() { ... }
  method1() { ... }
  method2() { ... }
  method3() { ... }
  ...
}
```

Затем используйте вызов `new MyClass()` для создания нового объекта со всеми перечисленными методами. При этом автоматически вызывается метод `constructor()`, в нём можно инициализовать объект. Например:



```
class User {

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

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

}

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`.

Затем можно вызывать методы объекта, такие как `user.sayHi()`.

Методы в классе не разделяются запятой. Это приводит к синтаксической ошибке.

В JavaScript класс – это разновидность функции. Рассмотрим пример:

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

alert(typeof User); // function
```

Вот что на самом деле делает конструкция `class User {...}`:

1. Создает функцию с именем `User`, которая становится результатом объявления класса. Код функции берется из метода `constructor` (она будет пустой, если такого метода нет).
2. Сохраняет все методы, такие как `sayHi`, в `User.prototype`.

Затем, при вызове метода на новых объектах `new User`, он возьмётся из прототипа. Таким образом, объект `new User` имеет доступ к методам класса. На картинке показан результат объявления `class User`:



Как видно из кода ниже, класс – это функция или, если точнее, это метод `constructor`, методы находятся в `User.prototype`.

```

class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

alert(typeof User); // function

alert(User === User.prototype.constructor); // true

alert(User.prototype.sayHi); // alert(this.name);

alert(Object.getOwnPropertyNames(User.prototype)); // constructor,
sayHi

```

Иногда говорят, что class — это просто «синтаксический сахар» в JavaScript (синтаксис для улучшения читаемости кода, но не делающий ничего принципиально нового), потому что можно сделать все то же самое без конструкции class. Например:

```

// перепишем класс User с помощью функций

// 1. Создаём функцию constructor
function User(name) {
  this.name = name;
}

// 2. Добавляем метод в прототип
User.prototype.sayHi = function() {
  alert(this.name);
};

let user = new User("Иван");
user.sayHi();

```

Результат этого кода очень похож на предыдущий. Поэтому, class можно считать синтаксическим сахаром для определения конструктора вместе с методами прототипа. Однако есть важные отличия:

1. Во-первых, функция, созданная с помощью class, помечена специальным внутренним свойством `[[FunctionKind]]:"classConstructor"`. Поэтому это не совсем то же самое, что создавать её вручную.

В отличие от обычных функций, конструктор класса не может быть вызван без new:

```

class User {
  constructor() {}
}

alert(typeof User); // function

```

```
User(); // Error: Class constructor User cannot be invoked without 'new'
```

Кроме того, строковое представление конструктора класса в большинстве движков JavaScript начинается с «class ...».

```
class User {  
  constructor() {}  
}  
  
alert(User); // class User { ... }
```

2. Методы класса являются перечислимыми. Определение класса устанавливает флаг enumerable в false для всех методов в "prototype".
3. Классы всегда используют use strict. Весь код внутри класса автоматически находится в строгом режиме.

### 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() {  
    alert("Привет");  
  }  
};
```

Как и Named Function Expressions, выражения классов могут иметь имя, которое видно только внутри класса. Если у Class Expression есть имя, то оно видно только внутри класса:

```
let User = class MyClass {  
  sayHi() {  
    alert(MyClass);  
  }  
};  
  
new User().sayHi(); // работает  
  
alert(MyClass); // ошибка
```

Можно динамически создавать классы «по-запросу»:

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

```

    };
  };
}

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

new User().sayHi(); // Привет

```

Как и в литеральных объектах, в классах можно объявлять генераторы, вычисляемые свойства, геттеры/сеттеры и т.д. Пример `user.name`, реализованного с использованием `get/set`:

```

class User {

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

  get name() {
    return this._name;
  }

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

let user = new User("Иван");
alert(user.name); // Иван

user = new User(""); // Имя слишком короткое.

```

При объявлении класса геттеры/сеттеры создаются на `User.prototype`:

```

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

```

Пример с вычисляемым свойством в скобках [...]:

```
class User {  
  
  ['say' + 'Hi']() {  
    alert("Привет");  
  }  
  
}  
  
new User().sayHi();
```

Для методов-генераторов добавьте перед именем \*.

### Свойства классов

Свойства классов добавлены в язык недавно. Старым браузерам может понадобиться полифил. В приведённом выше примере у класса User были только методы. Добавим свойство:

```
class User {  
  name = "Аноним";  
  
  sayHi() {  
    alert(`Привет, ${this.name}!`);  
  }  
}  
  
new User().sayHi();
```

Свойство name не устанавливается в User.prototype. Вместо этого оно создаётся оператором new перед запуском конструктора, это именно свойство объекта

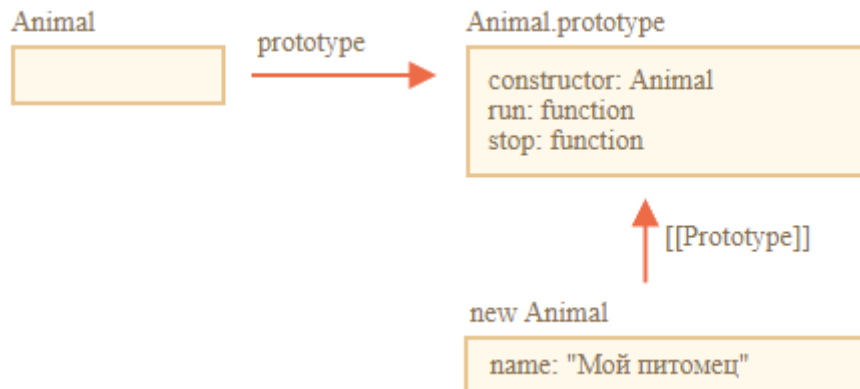
## 7. Наследование классов. Переопределение методов. Статические свойства и методы.

Допустим, у нас есть два класса. 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`);  
  }  
}
```

```
}  
}
```

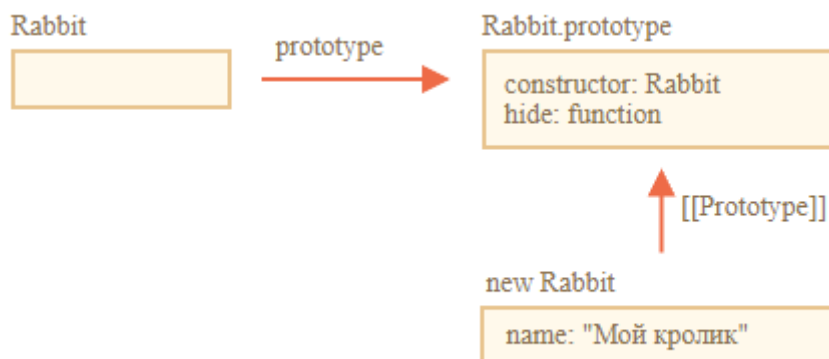
```
let animal = new Animal("Мой питомец");
```



Класс Rabbit:

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

```
let rabbit = new Rabbit("Мой кролик");
```



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

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

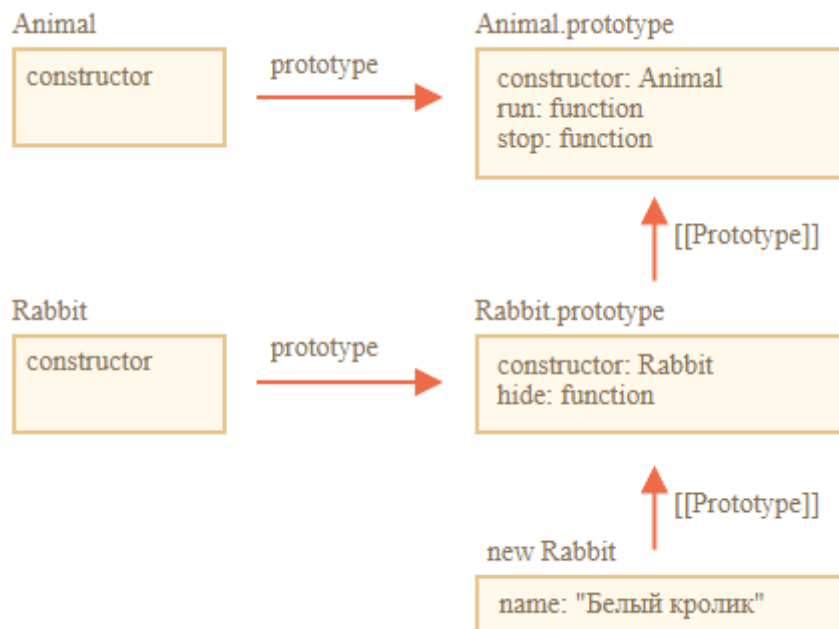
let rabbit = new Rabbit("Белый кролик");

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

```

Теперь код Rabbit стал короче, так как используется конструктор класса Animal по умолчанию и кролик может использовать метод run как и все животные.

На самом деле ключевое слово extends добавляет ссылку на `[[Prototype]]` из `Rabbit.prototype` в `Animal.prototype`:



Если метод не найден в `Rabbit.prototype`, JavaScript возьмёт его из `Animal.prototype`.

Синтаксис создания класса допускает указывать после `extends` не только класс, но 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 можно использовать функции для генерации классов в зависимости от многих условий и затем наследовать их.

### Переопределение методов

Сейчас Rabbit наследует от Animal метод stop, который устанавливает this.speed = 0. Если определить свой метод stop в классе Rabbit, то он будет использоваться взамен родительского:

```
class Rabbit extends Animal {
  stop() {
    // ...будет использован для rabbit.stop()
  }
}
```

Обычно нет необходимости полностью заменять родительский метод, а только сделать новый на его основе, изменяя или расширяя его функциональность. Для этого надо определить новый метод, добавив нужный функционал, и вызывать родительский метод до/после или в процессе.

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

- super.method(...) вызывает родительский метод.
- super(...) вызывает родительский конструктор (работает только внутри нашего конструктора).

Пусть наш кролик автоматически прячется при остановке:

```
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.`);
    }
}

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

    stop() {
        super.stop(); // вызываем родительский метод stop
        this.hide(); // и затем hide
    }
}

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

rabbit.run(5); // Белый кролик бежит со скоростью 5.
rabbit.stop(); // Белый кролик стоит. Белый кролик прячется!

```

Теперь у класса `Rabbit` есть метод `stop`, который вызывает родительский `super.stop()` в процессе выполнения.

У стрелочных функций нет `super`. При обращении к `super` стрелочной функции он берётся из внешней функции:

```

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

```

В примере `super` в стрелочной функции тот же самый, что и в `stop()`, поэтому метод отрабатывает как и ожидается. Если указать здесь «обычную» функцию, была бы ошибка:

```

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

```

Согласно спецификации, если класс расширяет другой класс и не имеет конструктора, то автоматически создаётся такой «пустой» конструктор. Например, у `Rabbit` нет своего конструктора.

```

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

```

Как видно, он просто вызывает конструктор родительского класса. Так будет происходить, пока не будет создан собственный конструктор. Добавим конструктор для Rabbit. Он будет устанавливать earLength в дополнение к name:

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  // ...
}

class Rabbit extends Animal {

  constructor(name, earLength) {
    this.speed = 0;
    this.name = name;
    this.earLength = earLength;
  }

  // ...
}

let rabbit = new Rabbit("Белый кролик", 10); // Error: this is not
defined.
```

Ошибка возникла из-за того, что в классах-потомках конструктор обязан вызывать super(...) и делать это перед использованием this. Дело в том, что в JavaScript существует различие между «функцией-конструктором наследующего класса» и всеми остальными. В наследующем классе соответствующая функция-конструктор помечена специальным внутренним свойством [[ConstructorKind]]:"derived". Разница в следующем:

- Когда выполняется обычный конструктор, он создаёт пустой объект и присваивает его this.
- Когда запускается конструктор унаследованного класса, он этого не делает. Вместо этого он ждёт, что это сделает конструктор родительского класса.

Поэтому, если создать собственный конструктор, то надо вызвать super, в противном случае объект для this не будет создан, и возникнет ошибка. Чтобы конструктор Rabbit работал, он должен вызвать super() до того, как использовать this, чтобы не было ошибки:

```
class Animal {

  constructor(name) {
    this.speed = 0;
    this.name = name;
```

```

    }

    // ...
}

class Rabbit extends Animal {

    constructor(name, earLength) {
        super(name);
        this.earLength = earLength;
    }

    // ...
}

let rabbit = new Rabbit("Белый кролик", 10);
alert(rabbit.name); // Белый кролик
alert(rabbit.earLength); // 10

```

### [[HomeObject]]

В JavaScript для функций добавлено специальное внутреннее свойство: `[[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. Посмотрим, как это работает – опять же, используя простые объекты:

```

let animal = {
    name: "Животное",
    eat() { // animal.eat.[[HomeObject]] == animal
        alert(`${this.name} eats.`);
    }
};

let rabbit = {
    __proto__: animal,
    name: "Кролик",
    eat() { // rabbit.eat.[[HomeObject]] == rabbit
        super.eat();
    }
};

let longEar = {
    __proto__: rabbit,
    name: "Длинноух",
    eat() { // longEar.eat.[[HomeObject]] == longEar
        super.eat();
    }
};

longEar.eat(); // Длинноух ест.

```

Это работает как задумано благодаря `[[HomeObject]]`. Метод, такой как `longEar.eat`, знает свой `[[HomeObject]]` и получает метод родителя из его прототипа. Вообще без использования `this`.

### Методы не «свободны»

Как известно, функции в JavaScript «свободны», не привязаны к объектам. Их можно копировать между объектами и вызывать с любым `this`. Но само 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 все ещё можно 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:

```
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");
  }
};

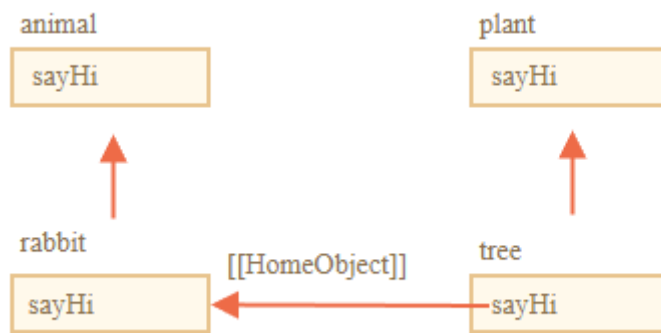
// tree наследует от plant
let tree = {
  __proto__: plant,
  sayHi: rabbit.sayHi // (*)
};

tree.sayHi(); // Я животное
```

Вызов `tree.sayHi()` показывает «Я животное». Определённо неверно. Причина проста:

- В строке `(*)`, метод `tree.sayHi` скопирован из `rabbit`.
- 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`.

Вот диаграмма происходящего:



Свойство `[[HomeObject]]` определено для методов как классов, так и обычных объектов. Но для объектов методы должны быть объявлены именно как `method()`, а не `"method: function()"`. В приведённом ниже примере используется синтаксис не метода, свойства-функции. Поэтому у него нет `[[HomeObject]]`, и наследование не работает:

```
let animal = {
  eat: function() { // должен быть короткий синтаксис: eat() {...}
    // ...
  }
};

let rabbit = {
  __proto__: animal,
  eat: function() {
    super.eat();
  }
};

rabbit.eat(); // Ошибка вызова super
```

Можно присвоить метод самой функции-классу, а не её `"prototype"`. Такие методы называются статическими. В классе такие методы обозначаются ключевым словом `static`, например:

```
class User {
  static staticMethod() {
    alert(this === User);
  }
}

User.staticMethod(); // true
```

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

```
class User() { }

User.staticMethod = function() {
    alert(this === User);
};
```

Значением `this` при вызове `User.staticMethod()` является сам конструктор класса `User` (правило «объект до точки»).

Обычно статические методы используются для реализации функций, принадлежащих классу, но не к каким-то конкретным его объектам. Например, есть объекты статей `Article`, и нужна функция для их сравнения. Естественное решение – сделать для этого метод `Article.compare`:

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

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

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 способ их сравнения. Это метод не отдельной статьи, а всего класса. Другим примером может быть так называемый «фабричный» метод. Представим, что нужно создавать статьи различными способами:

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

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

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

```

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

let article = Article.createToday();

alert( article.title ); // Сегодняшний дайджест

```

Теперь каждый раз, когда нужно создать сегодняшний дайджест, нужно вызывать `Article.createToday()`. 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:

```
Article.remove({id: 12345});
```

### Статические свойства

Эта возможность была добавлена в язык недавно. Примеры работают в последнем Chrome. Статические свойства также возможны, они выглядят как свойства класса, но с `static` в начале:

```

class Article {
    static publisher = "Иван Иванов";
}

alert( Article.publisher ); // Иван Иванов

```

Это то же самое, что и прямое присваивание `Article`:

```
Article.publisher = "Иван Иванов";
```

Статические свойства и методы наследуются. Например, метод `Animal.compare` в коде ниже наследуется и доступен как `Rabbit.compare`:

```

class Animal {

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

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

```

```

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

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

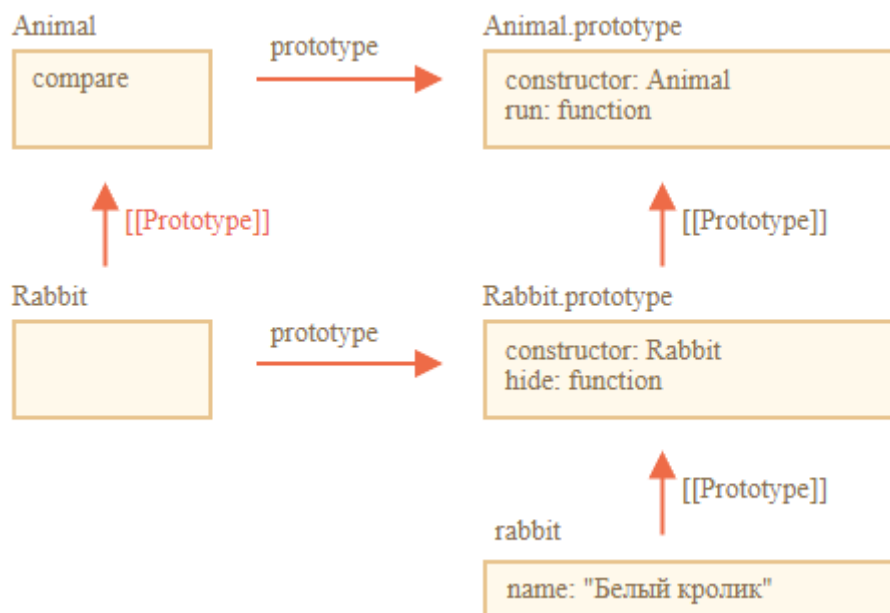
let rabbits = [
    new Rabbit("Белый кролик", 10),
    new Rabbit("Чёрный кролик", 5)
];

rabbits.sort(Rabbit.compare);

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

```

Можно вызвать `Rabbit.compare`, при этом будет вызван унаследованный `Animal.compare`. Это работает с использованием прототипов. `Extends` даёт `Rabbit` ссылку `[[Prototype]]` на `Animal`.



Так что `Rabbit extends Animal` создаёт две ссылки на прототип:

1. Функция `Rabbit` прототипно наследует от `Animal` function.
2. `Rabbit.prototype` прототипно наследует от `Animal.prototype`.

В результате наследование работает как для обычных, так и для статических методов:



```

class Animal {}
class Rabbit extends Animal {}

// для статики
alert(Rabbit.__proto__ === Animal); // true

// для обычных методов
alert(Rabbit.prototype.__proto__ === Animal.prototype);

```

## 8. Расширение встроенных классов. Оператор instanceof.

От встроенных классов, таких как Array, Map и других, тоже можно наследовать. Например, в этом примере PowerArray наследуется от встроенного Array:

```

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

let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // false

let filteredArr = arr.filter(item => item >= 10);
alert(filteredArr); // 10, 50
alert(filteredArr.isEmpty()); // false

```

Обратите внимание: встроенные методы, такие как filter, map и другие возвращают новые объекты унаследованного класса PowerArray. Их внутренняя реализация такова, что для этого они используют свойство объекта constructor.

В примере выше arr.constructor === PowerArray. Поэтому при вызове метода arr.filter() он внутри создаёт массив результатов, именно используя arr.constructor, а не обычный массив. Это удобно, поскольку можно продолжать использовать методы PowerArray далее на результатах.

Более того, можно настроить это поведение. При помощи специального статического геттера Symbol.species можно вернуть конструктор, который JavaScript будет использовать в filter, map и других методах для создания новых объектов. Если надо, чтобы методы map, filter и т. д. возвращали обычные массивы, то нужно вернуть Array в Symbol.species, вот так:

```

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

```

```

    // встроенные методы массива будут использовать этот метод как
    конструктор
    static get [Symbol.species]() {
        return Array;
    }
}

let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // false

// filter создаст новый массив, используя
arr.constructor[Symbol.species] как конструктор
let filteredArr = arr.filter(item => item >= 10);

// filteredArr не является PowerArray, это Array
alert(filteredArr.isEmpty()); // Error: filteredArr.isEmpty is not a
function

```

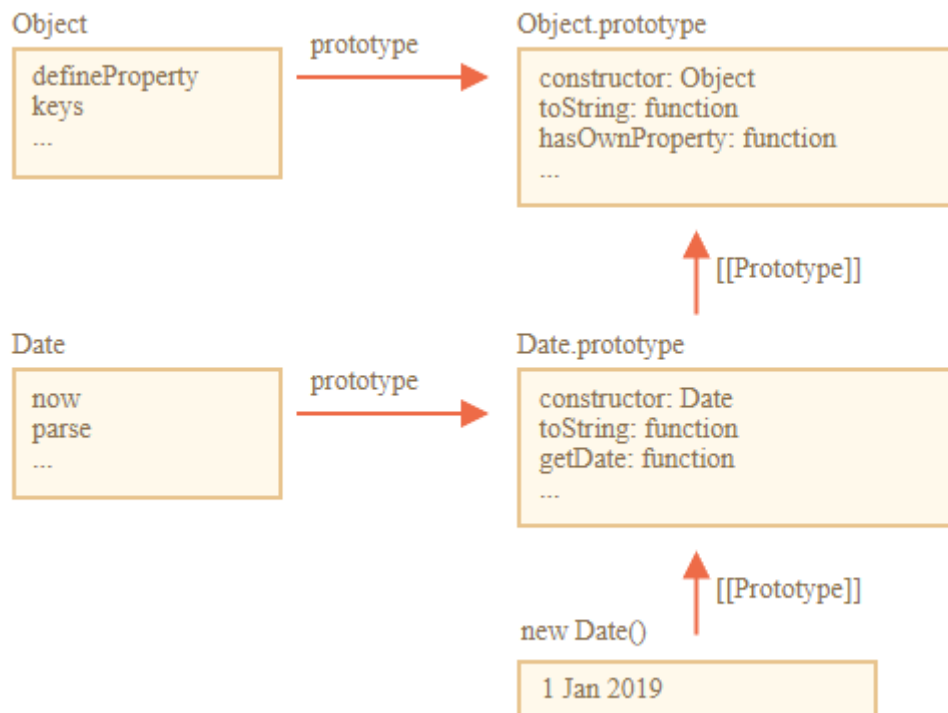
Теперь `.filter` возвращает `Array`. Расширенная функциональность не будет передаваться далее.

Другие коллекции, такие как `Map`, `Set`, работают аналогично. Они также исполуют `Symbol.species`.

### **Отсутствие статического наследования встроенных классов**

У встроенных объектов есть собственные статические методы, например, `Object.keys`, `Array.isArray` и т. д. Как известно, встроенные классы расширяют друг друга. Обычно, когда один класс наследует другому, то наследуются и статические методы. Но встроенные классы – исключение. Они не наследуют статические методы друг друга. Например, и `Array` и `Date` наследуют от `Object`, так что в их экземплярах доступны методы из `Object.prototype`. Но `Array.[[Prototype]]` не ссылается на `Object`, поэтому нет методов `Array.keys()` или `Date.keys()`.

Ниже представлена структура `Date` и `Object`:



Как видно, нет связи между `Date` и `Object`. Они независимы, только `Date.prototype` наследует от `Object.prototype`. В этом важное отличие наследования встроенных объектов от тех, которые получаются с использованием `extends`.

## 1. Проверка класса: "instanceof"

Оператор `instanceof` позволяет проверить, к какому классу принадлежит объект, с учётом наследования. Такая проверка может потребоваться во многих случаях. В рассматриваемых примерах она используется для создания полиморфной функции, которая интерпретирует аргументы по-разному в зависимости от их типа.

### Оператор `instanceof`

Синтаксис:

```
obj instanceof Class
```

Оператор вернёт `true`, если `obj` принадлежит классу `Class` или наследующему от него. Например:

```
class Rabbit {}  
let rabbit = new Rabbit();  
  
alert( rabbit instanceof Rabbit ); // true
```

Также это работает с функциями-конструкторами:

```
function Rabbit() {}

alert( new Rabbit() instanceof Rabbit ); // true
```

И для встроенных классов, таких как Array:

```
let arr = [1, 2, 3];
alert( arr instanceof Array ); // true
alert( arr instanceof Object ); // true
```

Обратите внимание, что arr также принадлежит классу Object, потому что Array наследует от Object.

Обычно оператор instanceof просматривает для проверки цепочку прототипов. Но это поведение может быть изменено при помощи статического метода Symbol.hasInstance.

Алгоритм работы obj instanceof Class работает примерно так:

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

Пример:

```
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 используется стандартная логика: проверяется, равен ли Class.prototype одному из прототипов в прототипной цепочке obj.

Другими словами, сравнивается:

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

В примере выше rabbit.\_\_proto\_\_ === Rabbit.prototype, так что результат будет получен немедленно. В случае с наследованием, совпадение будет на втором шаге:

```

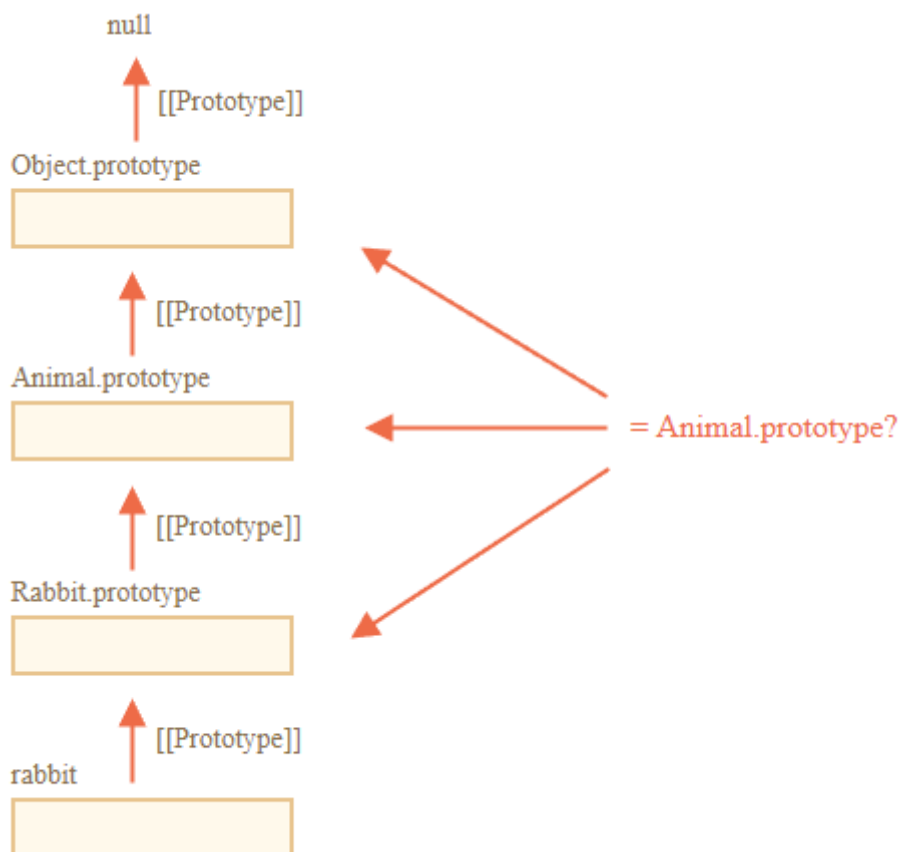
class Animal {}
class Rabbit extends Animal {}

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

// rabbit.__proto__ === Rabbit.prototype
// rabbit.__proto__.__proto__ === Animal.prototype

```

Вот иллюстрация того как `rabbit instanceof Animal` сравнивается с `Animal.prototype`:



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)`. Сам конструктор `Class` не участвует в процессе проверки. Важна только цепочка прототипов `Class.prototype`. Это может приводить к интересным последствиям при изменении свойства `prototype` после создания объекта. Как, например, тут:

```

function Rabbit() {}
let rabbit = new Rabbit();

// заменяем прототип
Rabbit.prototype = {};

```

```
alert( rabbit instanceof Rabbit ); // false
```

Известно, что обычные объекты преобразуются к строке как [object Object]:

```
let obj = {};  
  
alert(obj); // [object Object]  
alert(obj.toString()); // то же самое
```

Так работает реализация метода toString. Но у toString имеются скрытые возможности, которые делают метод гораздо более мощным. Можно 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 булевого типа это будет [object Boolean];
- для null: [object Null];
- для undefined: [object Undefined];
- для массивов: [object Array] и т.д.

Например:

```
let objectToString = Object.prototype.toString;  
  
let arr = [];  
  
alert( objectToString.call(arr) ); // [object Array]
```

В примере использовался call, чтобы выполнить функцию objectToString в контексте this=arr. Внутри, алгоритм метода toString анализирует контекст вызова this и возвращает соответствующий результат. Пример:

```
let s = Object.prototype.toString;  
  
alert( s.call(123) ); // [object Number]  
alert( s.call(null) ); // [object Null]  
alert( s.call(alert) ); // [object Function]
```

### Свойство Symbol.toStringTag

Поведение метода объектов toString можно настраивать, используя специальное свойство объекта Symbol.toStringTag. Например:

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

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

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

```
alert( window[Symbol.toStringTag]); // window
alert( XMLHttpRequest.prototype[Symbol.toStringTag] ); //
XMLHttpRequest
```

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

Как видно, результат — это значение `Symbol.toStringTag` (если он имеется) обёрнутое в `[object ...]`. В итоге получили `typeof`, 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 надо получить тип в виде строки, а не просто сделать проверку.

## 9. Модули. Основные возможности модулей.

По мере роста приложения, обычно возникает необходимость разделить его на много файлов, так называемых «модулей». Модуль обычно содержит класс или библиотеку с функциями. Долгое время в JavaScript отсутствовал синтаксис модулей на уровне языка. Это не было проблемой, потому что первые скрипты были маленькими и простыми. В модулях не было необходимости. Но со временем скрипты становились всё более и более сложными, поэтому сообщество придумало несколько вариантов организации кода в модули. Появились библиотеки для динамической подгрузки модулей. Например:

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

Теперь все они постепенно становятся частью истории, хотя их и можно найти в старых скриптах.

Система модулей на уровне языка появилась в стандарте JavaScript в 2015 году и постепенно эволюционировала. На данный момент она поддерживается большинством браузеров и `Node.js`.

Модуль — это файл с кодом. Один скрипт — это один модуль. Модули могут загружать друг друга и использовать директивы `export` и `import`, чтобы обмениваться функциональностью, вызывать функции одного модуля из другого:

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

Например, если есть файл `sayHi.js`, который экспортирует функцию:

```
//sayHi.js
export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

Тогда другой файл может импортировать её и использовать:

```
// main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // function...
sayHi('John'); // Hello, John!
```

Директива `import` загружает модуль по пути `./sayHi.js` относительно текущего файла и записывает экпортированную функцию `sayHi` в соответствующую переменную. 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">`. Вот так:

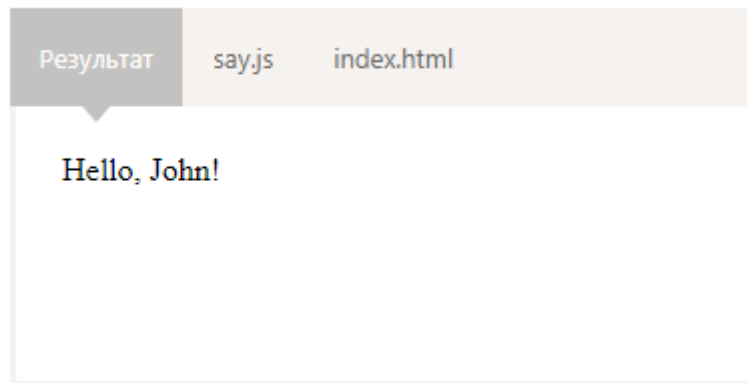
```
// say.js
export function sayHi(user) {
  return `Hello, ${user}!`;
}

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

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

Браузер автоматически загрузит и запустит импортированный модуль (и те, которые он импортирует, если надо), а затем запустит скрипт.





## Основные возможности модулей

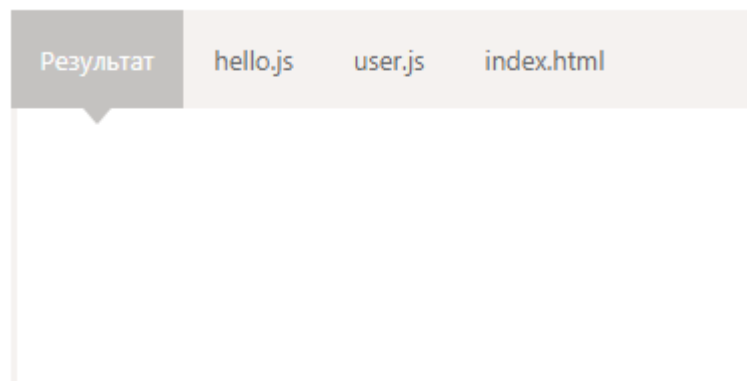
Есть основные возможности и особенности, работающие как в браузере, так и в серверном JavaScript.

В модулях всегда используется режим `use strict`. Например, присваивание к необъявленной переменной вызовет ошибку.

```
<script type="module">  
  а = 5; // ошибка  
</script>
```

Каждый модуль имеет свою собственную область видимости. Другими словами, переменные и функции, объявленные в модуле, не видны в других скриптах.

В следующем примере импортированы 2 скрипта, и `hello.js` пытается использовать переменную `user`, объявленную в `user.js`. В итоге ошибка:



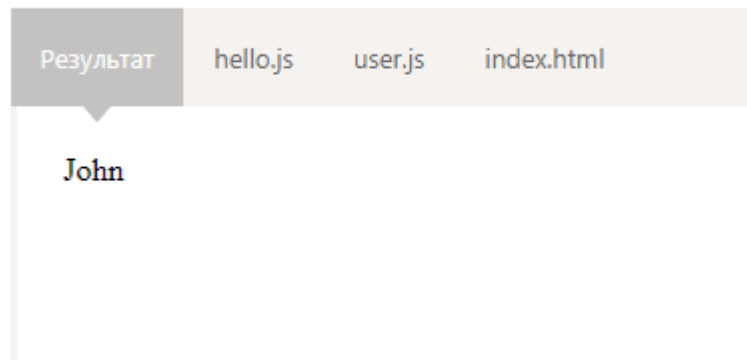
```
// hello.js  
alert(user);
```

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

```
// index.html  
<!doctype html>
```

```
<script type="module" src="user.js"></script>
<script type="module" src="hello.js"></script>
```

Модули должны экспортировать функционал, предназначенный для использования извне. А другие модули могут его импортировать. Так что надо импортировать user.js в hello.js и взять из него нужный функционал, вместо того чтобы полагаться на глобальные переменные. Правильный вариант:



```
// hello.js
import {user} from './user.js';

document.body.innerHTML = user; // John

// user.js
export let user = "John";

// index.html
import {user} from './user.js';

document.body.innerHTML = user; // John
```

В браузере также существует независимая область видимости для каждого 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 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.

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. Рассмотрим примеры.

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 надо, чтобы что-то можно было использовать много раз, то экспортируем это.

Рассмотрим более сложный пример. Представим, что модуль экспортирует объект:

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

Если модуль импортируется в нескольких файлах, то код модуля будет выполнен только один раз, объект `admin` будет создан и в дальнейшем будет передан всем импортёрам. Все импортёры получают один-единственный объект `admin`:

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

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

В примере оба файла, `1.js` и `2.js`, импортируют один и тот же объект. Изменения, сделанные в `1.js`, будут видны в `2.js`. Если что-то изменится в объекте `admin`, то другие модули тоже увидят эти изменения. Такое поведение позволяет конфигурировать модули при первом импорте. Можно 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:

```
// admin.js
export let admin = { };

export function sayHi() {
  alert(`Ready to serve, ${admin.name}!`);
}
```

В `init.js`, первом скрипте рассматриваемого приложения, установим `admin.name`. Тогда все это увидят, включая вызовы, сделанные из самого `admin.js`:

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

Другой модуль тоже увидит `admin.name`:

```
// other.js
import {admin, sayHi} from './admin.js';

alert(admin.name); // Pete

sayHi(); // Ready to serve, Pete!
```

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 встроеного
  скрипта
</script>
```

В модуле на верхнем уровне `this` не определён (`undefined`). Это незначительная особенность, но для полноты картины нужно упомянуть об этом. Сравним с немодульными скриптами, там `this` – глобальный объект:

```
<script>
  alert(this); // window
</script>

<script type="module">
  alert(this); // undefined
</script>
```

Есть и несколько других, именно браузерных особенностей скриптов с `type="module"` по сравнению с обычными скриптами.

Модули всегда выполняются в отложенном (`deferred`) режиме, точно так же, как скрипты с атрибутом `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. Например:

```
<script type="module">
  alert(typeof button); // object: скрипт может 'видеть' кнопку под
  ним
</script>
```

Сравним с обычным скриптом ниже:

```
<script>
  alert(typeof button); // Ошибка: кнопка не определена, скрипт не
  видит элементы под ним
</script>

<button id="button">Кнопка</button>
```

Второй скрипт выполнится раньше, чем первый. Поэтому сначала будет выведено `undefined`, а потом `object`. Это потому, что модули начинают выполняться после полной загрузки страницы. Обычные скрипты запускаются сразу же, поэтому сообщение из обычного скрипта видно первым.

При использовании модулей 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. Следует разместить «индикатор загрузки» или что-то ещё, чтобы не смутить этим посетителя.

Атрибут `async` работает во встроенных скриптах. Для немодульных 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.

```
<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. В примере ниже 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.

```
<script type="module" src="http://another-site.com/their.js"></script>
```

Это обеспечивает лучшую безопасность по умолчанию.

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';
```

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.

Старые браузеры не понимают атрибут `type="module"`. Скрипты с неизвестным атрибутом `type` просто игнорируются. Можно 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 как есть. Обычно, они объединяются вместе, специальными инструментами, например Webpack и после выкладывается код на рабочий сервер.

Одно из преимуществ использования сборщика – он предоставляет большой 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 необходимо 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.
4. В процессе могут происходить и другие трансформации, и оптимизации кода:
  - недоступный 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;
  - полученный файл можно минимизировать (удалить пробелы, заменить названия переменных на более короткие и т.д.).

Если используются инструменты сборки, то они объединяют модули вместе в один или несколько файлов, и заменяют `import/export` на свои вызовы. Поэтому итоговую сборку (в примере ниже – `bundle.js`) можно подключать и без атрибута `type="module"`, как обычный скрипт:

```
<script src="bundle.js"></script>
```

Хотя и «как есть» модули тоже можно использовать, а сборщик настроить позже при необходимости.

## 10. Модули: экспорт и импорт.

Директивы экспорт и импорт имеют несколько вариантов вызова.

## Экспорт до объявления

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;
  }
}
```

Обратите внимание, что `export` перед классом или функцией не делает их функциональным выражением. Это всё также объявление функции, хотя и экспортируемое. Большинство руководств по стилю кода в JavaScript не рекомендуют ставить точку с запятой после объявлений функций или классов. Поэтому в конце `export class` и `export function` не нужна точка с запятой:

```
export function sayHi(user) {
  alert(`Hello, ${user}!`);
} // без ; в конце
```

## Экспорт отдельно от объявления

Также можно написать `export` отдельно. Здесь сначала объявляются функции, а затем экспортируются:

```
// say.js
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

function sayBye(user) {
  alert(`Bye, ${user}!`);
}

export {sayHi, sayBye}; // список экспортируемых переменных
```

Технически также можно расположить `export` выше функций.

## Импорт \*

Обычно список того, что надо импортировать располагается, в фигурных скобках `import {...}`, например, вот так:



```
// main.js
import {sayHi, sayBye} from './say.js';

sayHi('John'); // Hello, John!
sayBye('John'); // Bye, John!
```

Но если импортировать нужно много чего, то можно импортировать всё сразу в виде объекта, используя `import * as <obj>`. Например:

```
// main.js
import * as say from './say.js';

say.sayHi('John');
say.sayBye('John');
```

На первый взгляд «импортировать всё» выглядит очень удобно, не надо писать лишнего. Но иногда надо явно перечислять список того, что нужно импортировать, тому есть несколько причин:

1. Современные инструменты сборки (webpack и другие) собирают модули вместе и оптимизируют их, ускоряя загрузку и удаляя неиспользуемый код.

Предположим, в проект была добавлена сторонняя библиотека `say.js` с множеством функций:

```
// say.js
export function sayHi() { ... }
export function sayBye() { ... }
export function becomeSilent() { ... }
```

Теперь, если из этой библиотеки в проекте использовать только одну функцию:

```
// main.js
import {sayHi} from './say.js';
```

тогда оптимизатор увидит, что другие функции не используются, и удалит остальные из собранного кода, тем самым делая код меньше. Это называется «tree-shaking».

2. Явно перечисляя то, что надо импортировать, можно получить более короткие имена функций: `sayHi()` вместо `say.sayHi()`.
3. Явное перечисление импортов делает код более понятным, позволяет увидеть, что именно и где используется. Это упрощает поддержку и рефакторинг кода.

## Импорт «как»

Также можно использовать `as`, чтобы импортировать под другими именами. Например, для краткости импортируем `sayHi` в локальную переменную `hi`, а `sayBye` импортируем как `bye`:

```
// main.js
import {sayHi as hi, sayBye as bye} from './say.js';

hi('John'); // Hello, John!
bye('John'); // Bye, John!
```

### Экспортировать «как»

Аналогичный синтаксис существует и для `export`. Давайте экспортируем функции, как `hi` и `bye`:

```
// say.js
export {sayHi as hi, sayBye as bye};
```

Теперь `hi` и `bye` — официальные имена для внешнего кода, их нужно использовать при импорте:

```
// main.js
import * as say from './say.js';

say.hi('John'); // Hello, John!
say.bye('John'); // Bye, John!
```

### 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`.

По большей части, удобнее второй подход, когда каждая «вещь» находится в своём собственном модуле. Естественно, требуется много файлов, если для всего делать отдельный модуль, но это не проблема. Так удобнее: 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. Указываем `export default` перед тем, что нужно экспортировать:

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

В файле может быть не более одного `export default`, и потом импорт без фигурных скобок:

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

new User('John');
```

Импорты без фигурных скобок выглядят красивее. Обычная ошибка начинающих: забывать про фигурные скобки. Запомним: фигурные скобки необходимы в случае именованных экспортов, для `export default` они не нужны.

Именованный экспорт	Экспорт по умолчанию
<code>export class User {...}</code>	<code>export default class User {...}</code>
<code>import {User} from ...</code>	<code>import User from ...</code>

Технически в одном модуле может быть, как экспорт по умолчанию, так и именованные экспорты, но на практике обычно их не смешивают. То есть, в модуле находятся либо именованные экспорты, либо один экспорт по умолчанию.

Так как в файле может быть максимум один `export default`, то экспортируемая сущность не обязана иметь имя. Например, всё это — полностью корректные экспорты по умолчанию:

```
export default class { // у класса нет имени
  constructor() { ... }
}
export default function(user) { // у функции нет имени
  alert(`Hello, ${user}!`);
}
// экспортируем значение, не создавая переменную
export default ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
```

Это нормально, потому что может быть только один `export default` в файле, так что `import` без фигурных скобок всегда знает, что импортировать. Без `default` такой экспорт выдал бы ошибку, так как необходимо имя класса, если это не экспорт по умолчанию:

```
export class { // Ошибка
  constructor() {}
}
```

### 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}!`);  
}  
  
export {sayHi as default};
```

Представим следующую ситуацию: модуль user.js экспортирует одну сущность «по умолчанию» и несколько именованных:

```
// 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');
```

Если импортируется 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');
```

Однако, у экспорта по умолчанию есть недостатки. Именованные экспорты «включают в себя» своё имя. Эта информация является частью модуля, она сообщает, что именно экспортируется. Именованные экспорты вынуждают использовать правильное имя при импорте:

```
// import {MyUser} не работает, должно быть именно имя {User}  
import {User} from './user.js';
```

В то время как для экспорта по умолчанию можно выбрать любое имя при импорте:

```
import User from './user.js'; // работает
```

```
import MyUser from './user.js'; // работает
```

Так что члены команды разработчиков 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. Вот так:

```
import User from './user.js';  
import LoginForm from './loginForm.js';  
import func from '/path/to/func.js';
```

Тем не менее, в некоторых командах это считают серьёзным доводом против экспортов по умолчанию и предпочитают использовать именованные экспорты везде. Даже если экспортируется только одна вещь, она всё равно экспортируется с именем, без использования default. Это также немного упрощает реэкспорт.

### Реэкспорт

Синтаксис «реэкспорта» `export ... from ...` позволяет импортировать что-то и тут же экспортировать, даже под другим именем:

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

Рассмотрим практический пример использования. Представим, что разрабатывается «пакет»: папка со множеством модулей, из которой часть функционала экспортируется наружу (инструменты вроде NPM позволяют публиковать и распространять такие пакеты), а многие модули – просто вспомогательные, для внутреннего использования в других модулях пакета. Структура файлов может быть такой:

```
auth/  
  index.js  
  user.js  
  helpers.js  
  tests/  
    login.js  
  providers/  
    github.js  
    facebook.js  
  ...
```

Надо сделать функционал пакета доступным через единую точку входа: «главный файл» `auth/index.js`. Чтобы можно было использовать его следующим образом:

```
import {login, logout} from 'auth/index.js'
```

Идея в том, что внешние разработчики, которые будут использовать пакет, не должны разбираться с его внутренней структурой. Всё, что нужно, надо экспортировать в `auth/index.js`, а остальное скрыть от разработчиков. Так как нужный функционал может быть разбросан по модулям пакета, то надо импортировать их в `auth/index.js` и тут же экспортировать наружу.

```
// auth/index.js
```

```
// импортировать login/logout и тут же экспортировать
import {login, logout} from './helpers.js';
export {login, logout};
```

```
// импортировать экспорт по умолчанию как User и тут же экспортировать
import User from './user.js';
export {User};
```

Теперь пользователи пакета могут писать `import {login} from "auth/index.js"`. Запись `export ... from ...` — это просто более короткий вариант такого импорта-экспорта:

```
// auth/index.js
```

```
// импортировать login/logout и тут же экспортировать
export {login, logout} from './helpers.js';
```

```
// импортировать экспорт по умолчанию как User и тут же экспортировать
export {default as User} from './user.js';
```

### Реэкспорт экспорта по умолчанию

При реэкспорте экспорт по умолчанию нужно обрабатывать особым образом. Например, есть `user.js`, из которого надо реэкспортировать класс `User`:

```
// user.js
export default class User {
  // ...
}
```

1. `export User from './user.js'` не будет работать, возникнет синтаксическая ошибка.

Чтобы реэкспортировать экспорт по умолчанию, надо написать `export {default as User}`, как в примере выше.

2. `export * from './user.js'` реэкспортирует только именованные экспорты, исключая экспорт по умолчанию.

Если надо реэкспортировать и именованные экспорты, и экспорт по умолчанию, то понадобятся две инструкции:

```
export * from './user.js'; // для реэкспорта именованных экспортов
export {default} from './user.js'; // для реэкспорта по умолчанию
```

Такое особое поведение реэкспорта с экспортом по умолчанию – одна из причин того, почему их неудобно использовать.

Инструкции экспорта и импорта, которые рассматривались в предыдущем вопросе, называются «статическими». Синтаксис у них весьма простой и строгий.

Во-первых, нельзя динамически задавать никакие из параметров `import`. Путь к модулю должен быть строковым примитивом и не может быть вызовом функции. Вот так работать не будет:

```
import ... from getModuleName(); // Ошибка, должна быть строка
```

Во-вторых, нельзя делать импорт в зависимости от условий или в процессе выполнения.

```
if(...) {
  import ...; // Ошибка, запрещено
}

{
  import ...; // Ошибка, нельзя ставить импорт в блок
}
```

Всё это следствие того, что цель директив `import/export` – задать костяк структуры кода. Благодаря им она может быть проанализирована, модули могут быть собраны в один файл специальными инструментами, а неиспользуемые экспорты удалены. Это возможно только благодаря тому, что всё статично.

### Выражение `import()`

Выражение `import(module)` загружает модуль и возвращает промис, результатом которого становится объект модуля, содержащий все его экспорты. Использовать его можно динамически в любом месте кода, например, так:

```
let modulePath = prompt("Какой модуль загружать?");

import(modulePath)
  .then(obj => <объект модуля>)
  .catch(err => <ошибка загрузки, например если нет такого модуля>)
```

Или если внутри асинхронной функции, то можно `let module = await import(modulePath)`. Например, если у нас есть такой модуль `say.js`:

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

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

То динамический импорт может выглядеть так:

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

hi();
bye();
```

А если в say.js указан экспорт по умолчанию:

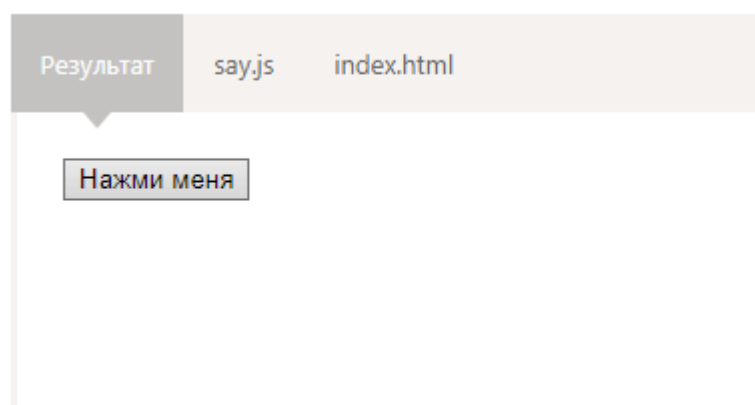
```
// say.js
export default function() {
  alert("Module loaded (export default)!");
}
```

То для доступа к нему следует взять свойство default объекта модуля:

```
let obj = await import('./say.js');
let say = obj.default;
// или, одной строкой: let {default: say} = await import('./say.js');

say();
```

Вот полный пример:



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

export function bye() {
```



```

    alert(`Пока`);
}

export default function() {
    alert("Модуль загружен (экспорт по умолчанию)!");
}

// index.html
<!doctype html>
<script>
    async function load() {
        let say = await import('./say.js');
        say.hi(); // Привет!
        say.bye(); // Пока!
        say.default(); // Модуль загружен (экспорт по умолчанию)!
    }
</script>
<button onclick="load()">Нажми меня</button>

```

Динамический импорт работает в обычных скриптах, он не требует указания `script type="module"`.

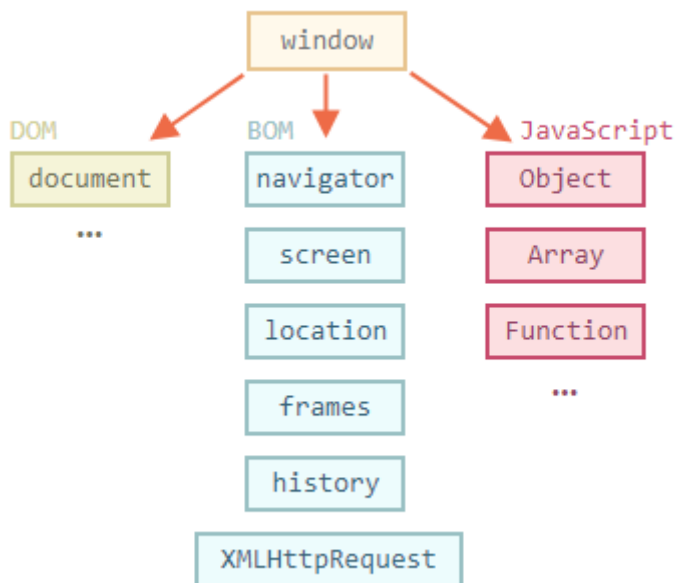
Хотя `import()` и выглядит похоже на вызов функции, на самом деле это специальный синтаксис, так же, как, например, `super()`.

Так что нельзя копировать `import` в другую переменную или вызвать при помощи `.call/apply`. Это не функция.

## 11.Окружение: DOM, BOM. Дерево DOM.

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, даже в кофеварке. Каждая среда предоставляет свой функционал, который 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 изображении ниже в общих чертах показано, что доступно для JavaScript в браузерном окружении:



Как видно, 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 используется `window` как глобальный объект:

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

```
window.sayHi();
```

А здесь используется `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. Они будут рассмотрены позднее.

## DOM (Document Object Model)

*Document Object Model*, сокращенно DOM – объектная модель документа, которая представляет все содержимое страницы в виде объектов, которые можно менять.

*Объект document* – основная «входная точка». С его помощью можно 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);
```

В примере использован 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>.

Спецификация DOM описывает структуру документа и предоставляет объекты для манипуляций со страницей. Существует 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.

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. Стили из JavaScript редко добавляются/удаляются, 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 можно 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.

BOM является частью общей спецификации HTML. Спецификация HTML по адресу <https://html.spec.whatwg.org> не только про «язык HTML»

(теги, атрибуты), она также покрывает целое множество объектов, методов и специфичных для каждого браузера расширений DOM. Это всё «HTML в широком смысле». Для некоторых вещей есть отдельные спецификации, перечисленные на <https://spec.whatwg.org>.

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, можно 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
```

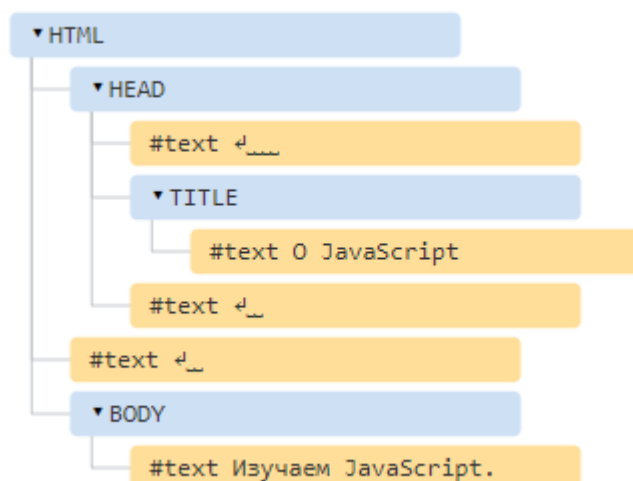
Это был лишь небольшой пример того, что может DOM.

## Структура DOM

Разберемся со структурой DOM. Рассмотрим простой документ:

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>0 лосях</title>
  </head>
  <body>
    Правда о лосях.
  </body>
</html>
```

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



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. Например, в теге `<title>` есть текстовый узел "О лосях".

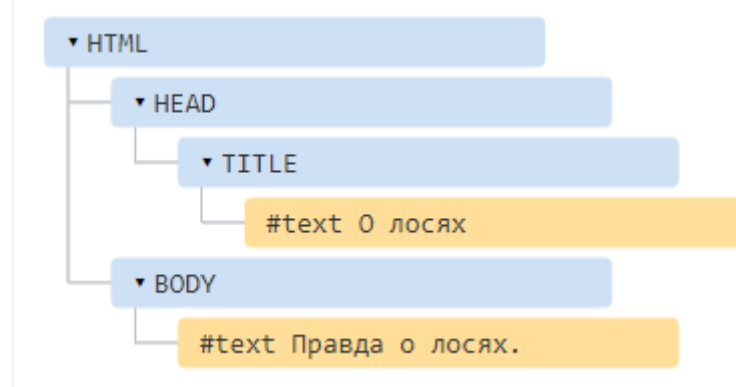
Обратите внимание на специальные символы в текстовых узлах: перевод строки: `\n` (в JavaScript он обозначается как `\n`) и пробел: . Пробелы и переводы строки – это полноправные символы, как буквы и цифры. Они образуют текстовые узлы и становятся частью дерева DOM. Так, в примере выше в теге `<head>` 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 записать что-либо после закрывающего тега `</body>`, браузер автоматически перемещает эту запись в конец `body`, поскольку спецификация HTML требует чтобы все содержимое было внутри `<body>`. Поэтому после закрывающего тега `</body>` не может быть никаких пробелов.

В остальных если в документе есть пробелы (или любые другие символы), они становятся текстовыми узлами дерева DOM, и если их надо удалить, то в DOM их тоже не будет. Здесь пробельных текстовых узлов нет:

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



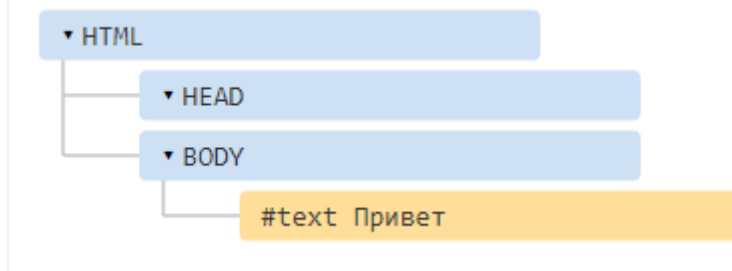
Пробелы по краям строк и пробельные текстовые узлы скрыты в инструментах разработки и обычно не отображаются. Таким образом инструменты разработки экономят место на экране.

В дальнейших иллюстрациях DOM также будут для краткости пропущены пробельные текстовые узлы там, где они не имеют значения. Обычно они не влияют на то, как отображается документ.

## 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>`.

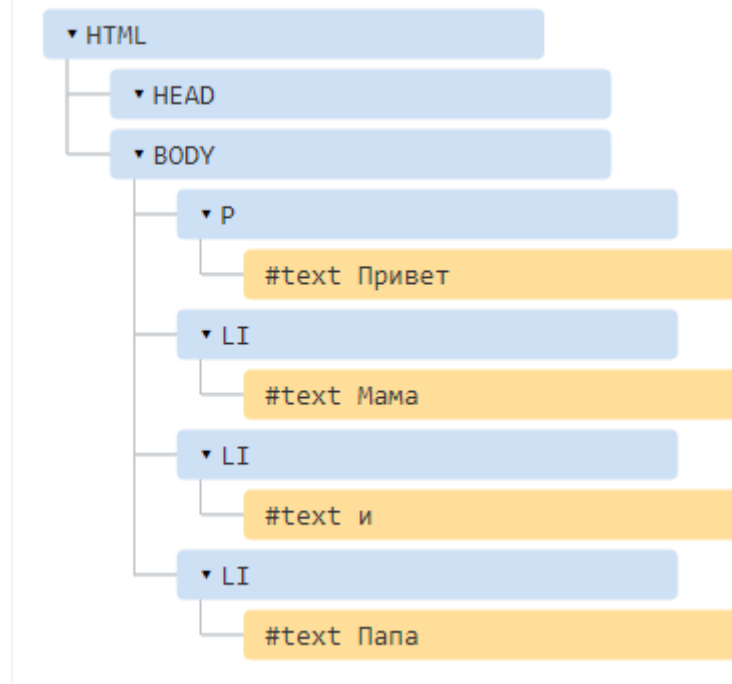
Например, если HTML-файл состоит из единственного слова "Привет", браузер обернёт его в теги `<html>` и `<body>`, добавит необходимый тег `<head>`, и DOM будет выглядеть так:



При генерации DOM браузер самостоятельно обрабатывает ошибки в документе, закрывает теги и так далее. Такой документ с незакрытыми тегами:

```
<p>Привет
<li>Мама
<li>и
<li>Папа
```

Но DOM будет нормальным, потому что браузер сам закрывает теги и восстановит отсутствующие детали:

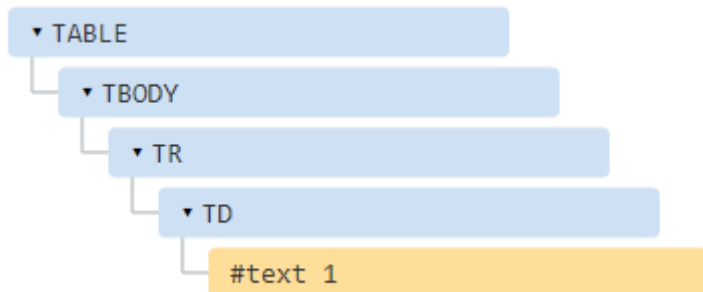


Важный «особый случай» – работа с таблицами. По стандарту DOM у них должен быть `<tbody>`, но в HTML их можно написать (официально) без

него. В этом случае браузер добавляет `<tbody>` в DOM самостоятельно. Для такого HTML:

```
<table id="table"><tr><td>1</td></tr></table>
```

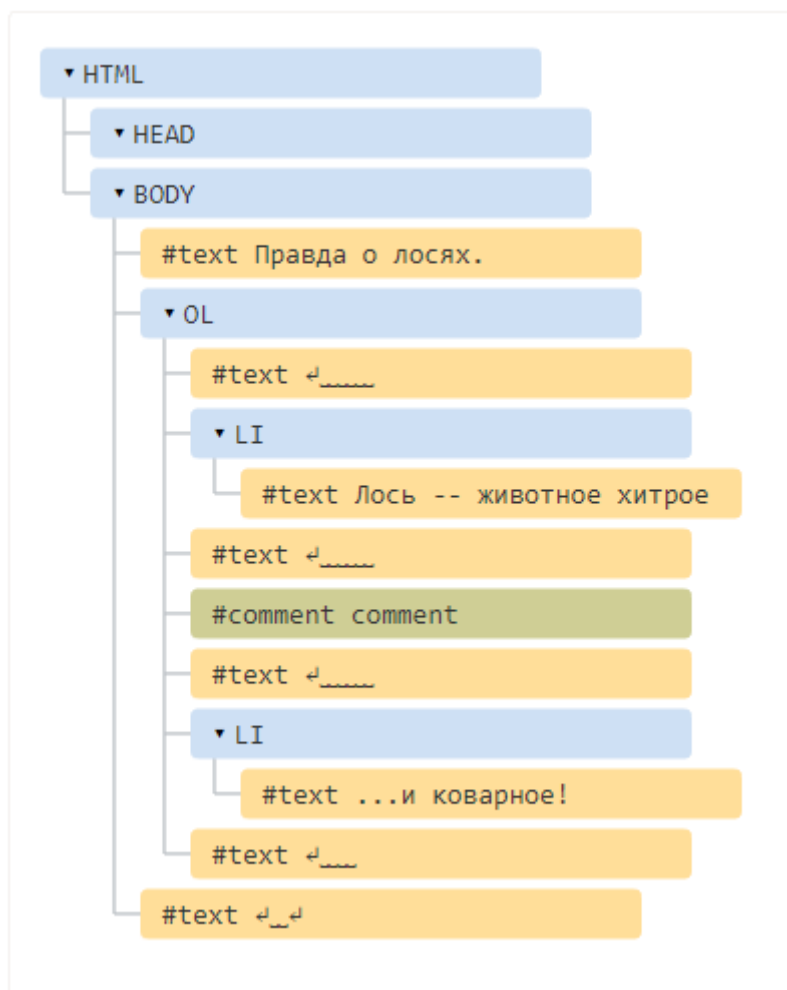
DOM-структура будет такой:



### Другие типы узлов

Есть и некоторые другие типы узлов, кроме элементов и текстовых узлов. Например, узел-комментарий:

```
<!DOCTYPE HTML>
<html>
<body>
  Правда о лосях.
  <ol>
    <li>Лось -- животное хитрое</li>
    <!-- комментарий -->
    <li>...и коварное!</li>
  </ol>
</body>
</html>
```



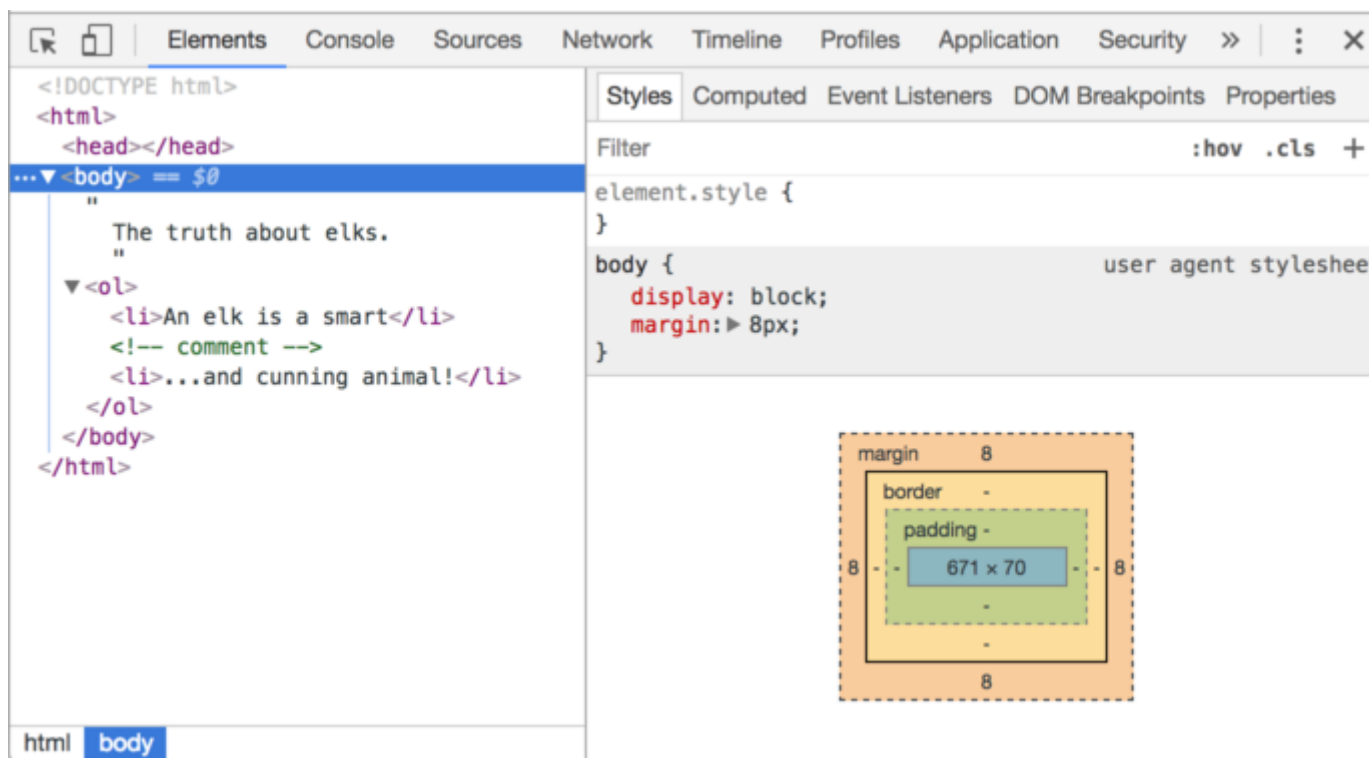
Здесь видно, что новый узел нового типа – комментарий, обозначенный как `#comment`, между двумя текстовыми узлами. Есть важное правило: если что-то есть в HTML, то оно должно быть в DOM-дереве. Поэтому все, что есть в HTML, даже комментарии, является частью DOM. Даже директива `<!DOCTYPE...>`, которая располагается в начале HTML, тоже является DOM-узлом. Она находится в дереве DOM прямо перед `<html>`. Даже объект `document`, представляющий весь документ, формально, является DOM узлом.

Существует 12 типов узлов. Но на практике в основном работают с четырьмя 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.

Чтобы посмотреть структуру DOM в реальном времени, можно использовать инструменты разработчика браузера. Для этого откройте страницу `*.html`, включите инструменты разработчика и перейдите на вкладку `Elements`. Выглядит примерно так:

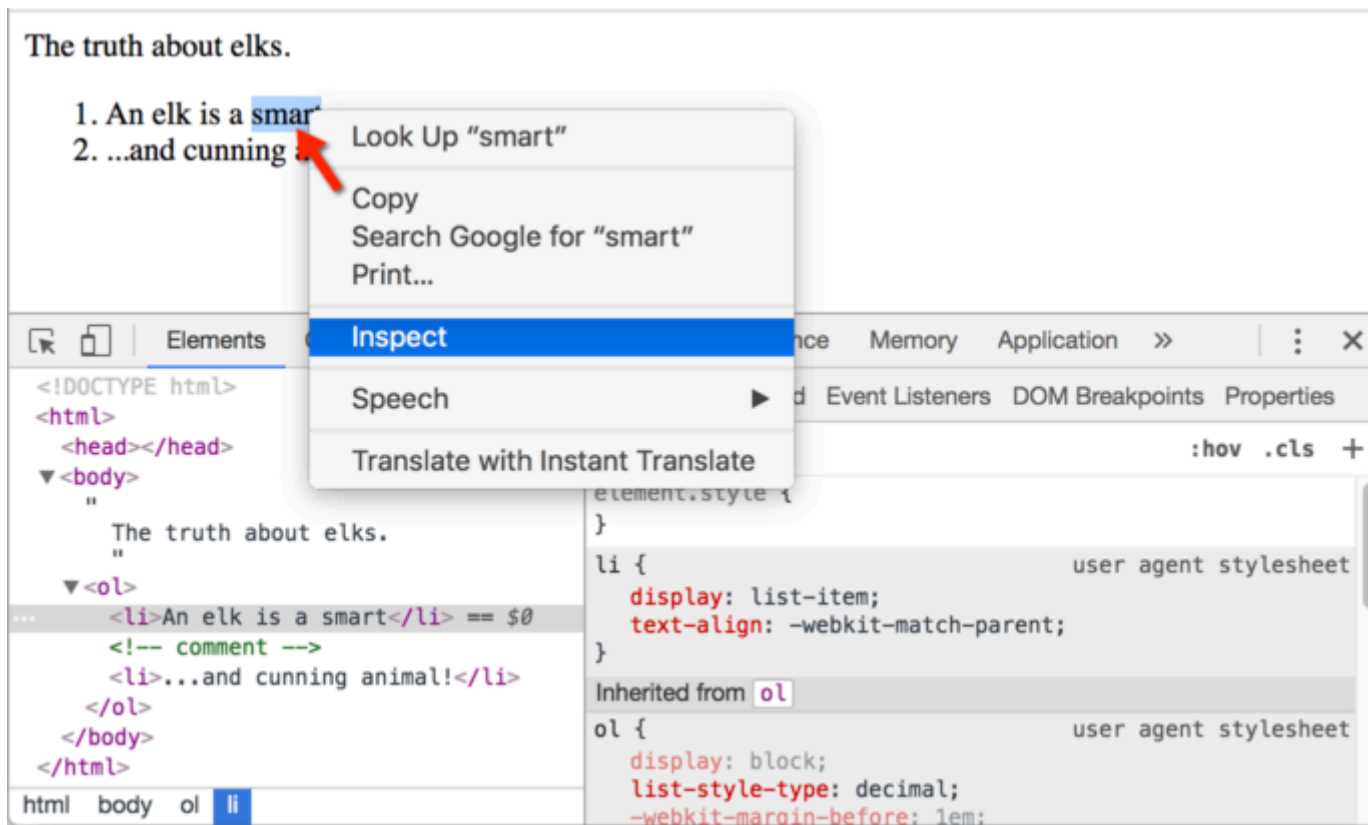




Можно увидеть DOM, раскрыть/свернуть элементы, детально рассмотреть их и так далее. Обратите внимание, что структура DOM в инструментах разработчика отображается в упрощённом виде. Текстовые узлы показаны как простой текст. И кроме пробелов нет никаких «пустых» текстовых узлов, что очень удобно.

Клик по этой  кнопке в левом верхнем углу инспектора позволяет при помощи мыши (или другого устройства ввода) выбрать элемент на веб-странице и «проинспектировать» его (браузер сам найдёт и отметит его во вкладке Elements). Этот способ отлично подходит, когда есть огромная HTML-страница (и соответствующий ей огромный DOM), и надо увидеть, где находится интересующий нас элемент.

Есть и другой способ сделать это, можно кликнуть на странице по элементу правой кнопкой мыши и в контекстном меню выбрать «Inspect».



В правой части инструментов разработчика находятся следующие подразделы:

- Styles – здесь показан CSS, применённый к текущему элементу: правило за правилом, включая встроенные стили (выделены серым). Почти все можно отредактировать на месте, включая размеры/внешние и внутренние отступы.
- Computed – здесь видны итоговые CSS-свойства элемента, которые он приобрёл в результате применения всего каскада стилей (в том числе унаследованные свойства и т.д.).
- Event Listeners – в этом разделе видны обработчики событий, привязанные к DOM-элементам.

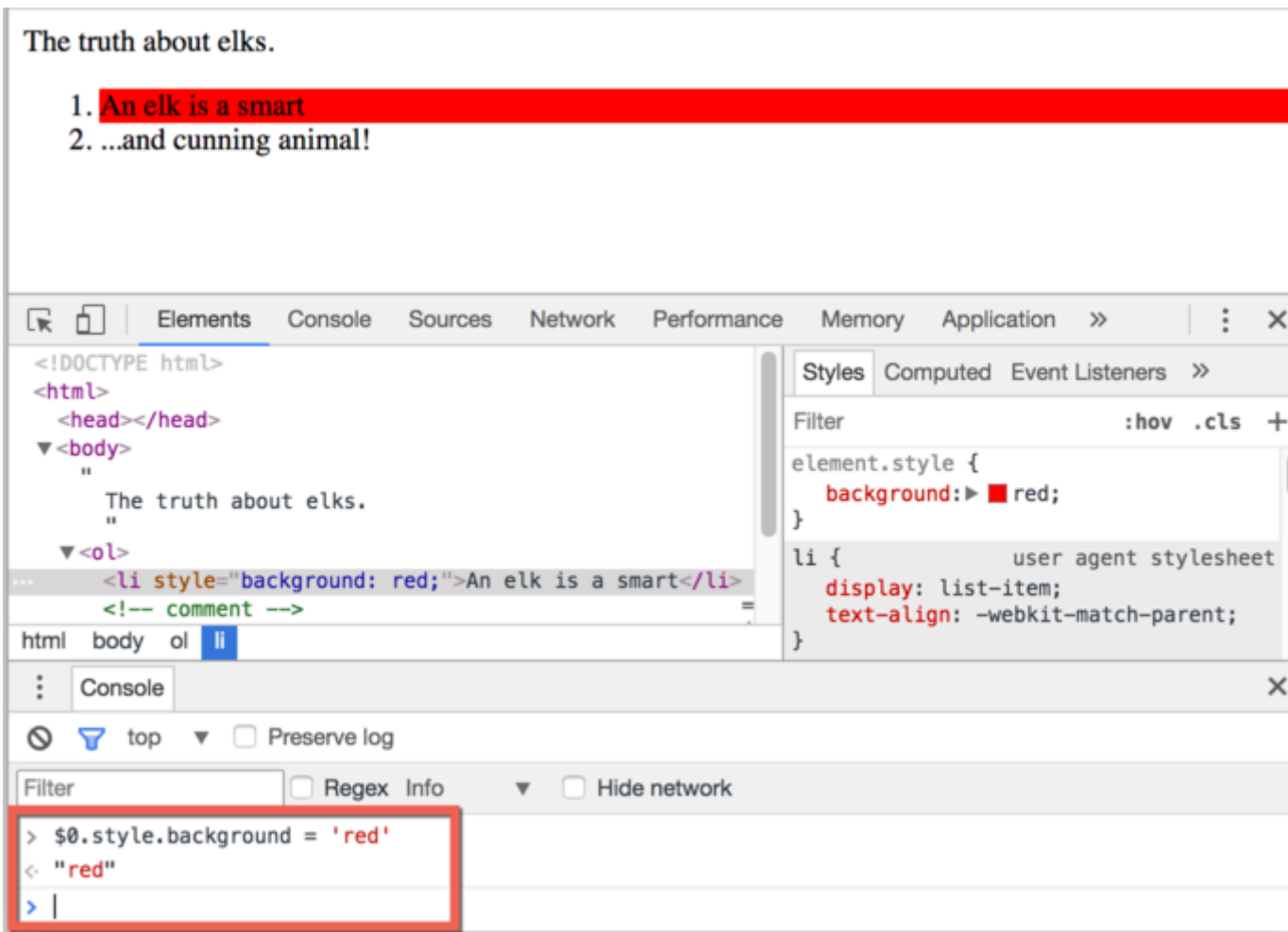
### Взаимодействие с консолью

При работе с DOM, часто требуется применить к нему JavaScript. Например: получить узел и запустить какой-нибудь код для его изменения, чтобы посмотреть результат. Вот несколько подсказок по тому, как перемещаться между вкладками Elements и Console. Для начала:

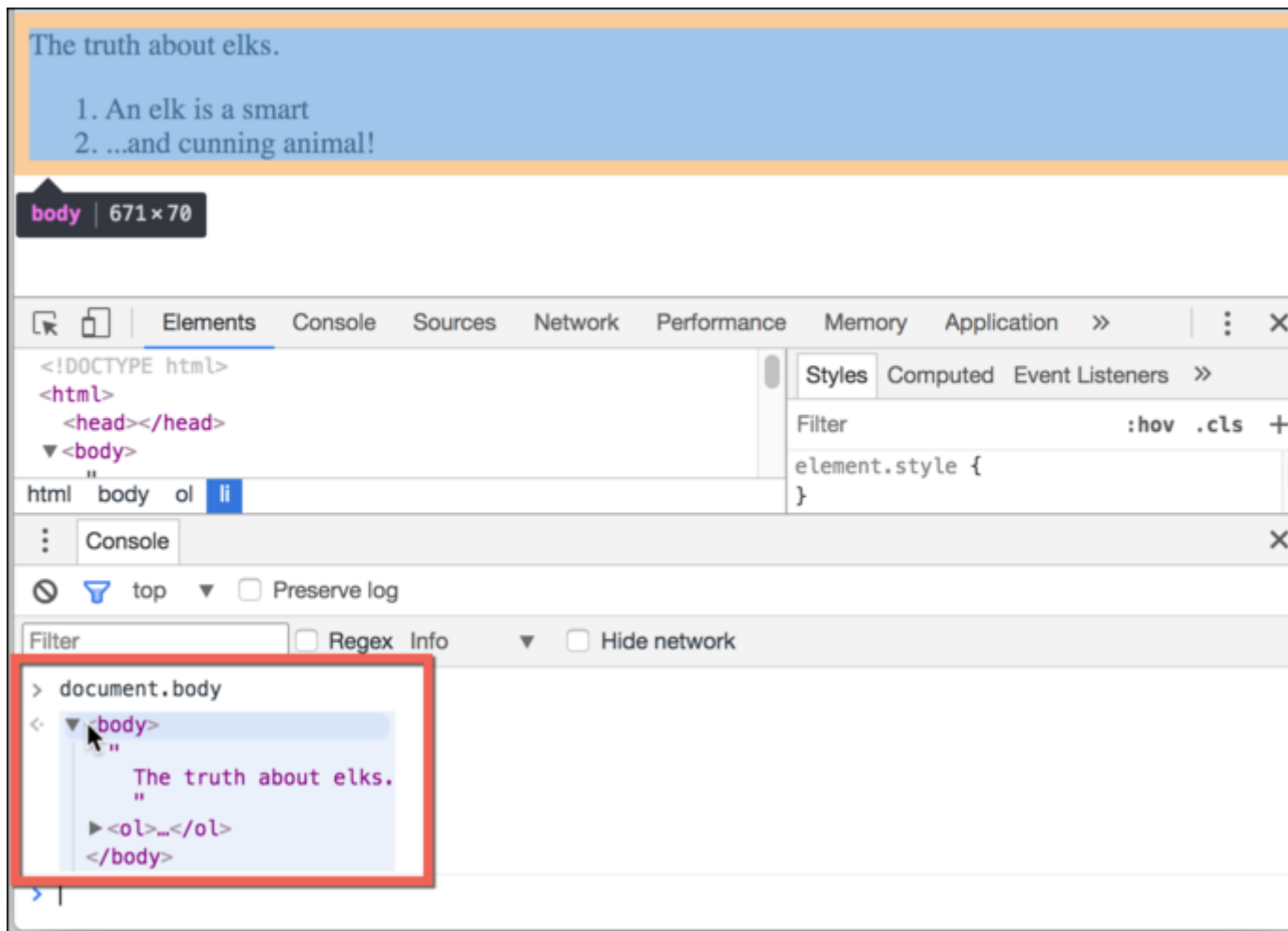
- На вкладке Elements выберите первый элемент `<li>`.
- Нажмите Esc – прямо под вкладкой Elements откроется Console.

Последний элемент, выбранный во вкладке Elements, доступен в консоли как `$0`, предыдущий, выбранный до него, как `$1` и т.д.

Теперь можно запускать на них команды. Например `$0.style.background = 'red'` сделает выбранный элемент красным, как здесь:



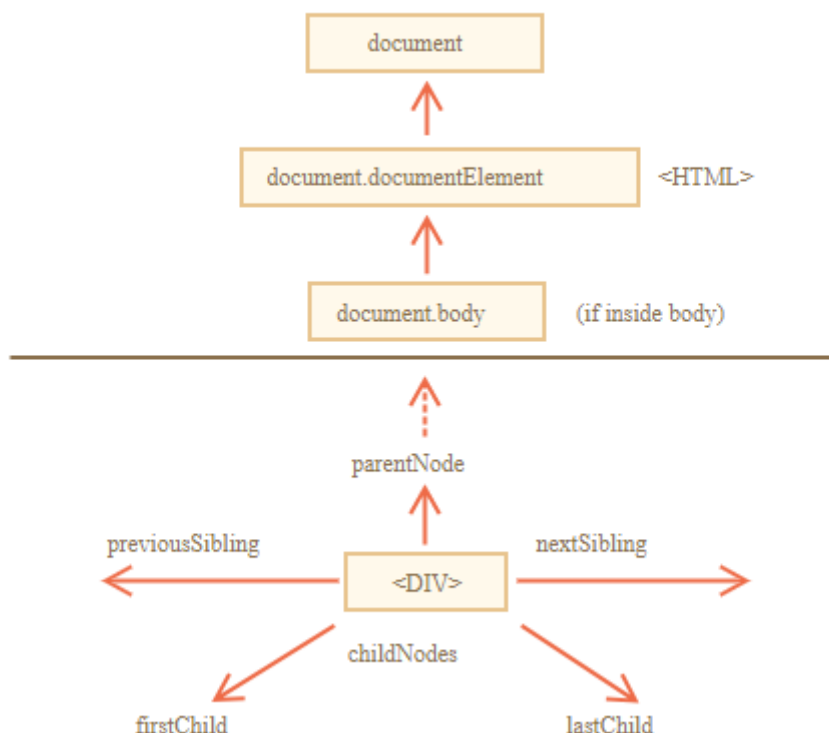
Так можно получить узел из Elements в Console. Есть и обратный путь: если есть переменная `node`, ссылающаяся на DOM-узел, можно использовать в консоли команду `inspect(node)`, чтобы увидеть этот элемент во вкладке Elements. Или можно просто вывести DOM-узел в консоль и исследовать «на месте», как `document.body` ниже:



Это может быть полезно для отладки. Инструменты разработчика браузера отлично помогают в разработке: можно исследовать DOM, пробовать с ним что-то делать и смотреть, что идёт не так.

## 12. Навигация и методы поиска DOM-элементов.

DOM позволяет 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 можно получить доступ к любому узлу. Так выглядят основные ссылки, по которым можно переходить между узлами DOM:



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 еще не существует в момент выполнения скрипта. В частности, если скрипт находится в `<head>`, `document.body` в нём недоступен, потому что браузер его еще не прочитал. Поэтому, в примере ниже первый `alert` выведет `null`:

```
<html>

<head>
  <script>
    alert( "Из HEAD: " + document.body ); // null, <body> еще нет
  </script>
</head>

<body>

  <script>
    alert( "Из BODY: " + document.body ); // HTMLBodyElement, теперь
    он есть
  </script>

</body>
```

```
</html>
```

В DOM значение null значит «не существует» или «нет такого узла».

### Дети: `childNodes`, `firstChild`, `lastChild`

Здесь и далее будут использоваться два принципиально разных термина:

- 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.

В примере ниже детьми тега `<body>` являются теги `<div>` и `<ul>` (и несколько пустых текстовых узлов):

```
<html>
<body>
  <div>Начало</div>

  <ul>
    <li>
      <b>Информация</b>
    </li>
  </ul>
</body>
</html>
```

А потомки `<body>` – это и прямые дети `<div>`, `<ul>` и вложенные в них: `<li>` (потомок `<ul>`) и `<b>` (потомок `<li>`) – в общем, все элементы поддерева.

Коллекция `childNodes` содержит список всех детей, включая текстовые узлы. Пример ниже последовательно выведет детей `document.body`:

```
<html>
<body>
  <div>Начало</div>

  <ul>
    <li>Информация</li>
  </ul>

  <div>Конец</div>

  <script>
    for (let i = 0; i < document.body.childNodes.length; i++) {
      alert( document.body.childNodes[i] ); // Text, DIV, Text, UL,
..., SCRIPT
    }
  </script>
```

```
</script>
...какой-то HTML-код...
</body>
</html>
```

Обратим внимание на одну особенность. Если запустить пример выше, то последним будет выведен элемент `<script>`. На самом деле, в документе есть ещё «какой-то HTML-код», но на момент выполнения скрипта браузер ещё до него не дошёл, поэтому скрипт не видит его.

Свойства `firstChild` и `lastChild` обеспечивают быстрый доступ к первому и последнему дочернему элементу. Они, по сути, являются всего лишь сокращениями. Если у тега есть дочерние узлы, условие ниже всегда верно:

```
elem.childNodes[0] === elem.firstChild
elem.childNodes[elem.childNodes.length - 1] === elem.lastChild
```

Для проверки наличия дочерних узлов существует также специальная функция `elem.hasChildNodes()`.

### DOM-коллекции

`childNodes` это не массив, а коллекция – особый перебираемый объект-псевдомассив. И есть два важных следствия из этого:

1. Для перебора коллекции можно 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. Если хочется использовать именно методы массива, то можно создать настоящий массив из коллекции, используя `Array.from`:

```
alert( Array.from(document.body.childNodes).filter ); // сделали
массив
```

DOM-коллекции и все навигационные свойства, рассматриваемые в этой теме, доступны только для чтения. Нельзя заменить один дочерний узел

на другой, просто написав `childNodes[i] = ...`. Для изменения DOM требуются другие методы. Изучим их позже.

Почти все DOM-коллекции, за небольшим исключением, отражают текущее состояние DOM. Если сохранить ссылку на `elem.childNodes` и добавить/удалить узлы в DOM, то они появятся в сохраненной коллекции автоматически.

Коллекции перебираются циклом `for..of`. Не стоит использовать для этого цикл `for..in`. Цикл `for..in` перебирает все перечисляемые свойства. А у коллекций есть некоторые «лишние», редко используемые свойства, которые обычно не нужны:

```
<body>
<script>
  // выводит 0, 1, length, item, values и другие свойства.
  for (let prop in document.body.childNodes) alert(prop);
</script>
</body>
```

### Соседи и родитель

Соседи – это узлы, у которых один и тот же родитель. Например, здесь `<head>` и `<body>` соседи:

```
<html>
  <head>...</head><body>...</body>
</html>
```

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

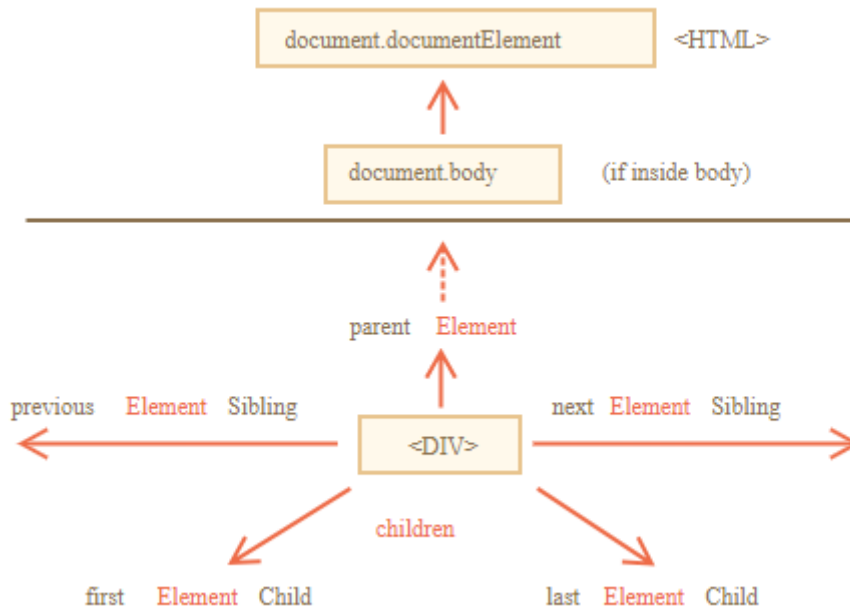
// после <head> идет <body>
alert( document.head.nextSibling ); // HTMLBodyElement

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

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 не нужны. Как правило, надо манипулировать узлами-элементами, которые представляют собой теги и



формируют структуру страницы. Поэтому рассмотрим дополнительный набор ссылок, которые учитывают только узлы-элементы:



Эти ссылки похожи на те, что рассматривались 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.

Свойство `parentElement` возвращает родитель-элемент, а `parentNode` возвращает «любого родителя». Обычно эти свойства одинаковы: они оба получают родителя. За исключением `document.documentElement`:

```
alert( document.documentElement.parentNode ); // выведет document
alert( document.documentElement.parentElement ); // выведет null
```

Причина в том, что родителем корневого узла `document.documentElement` (`<html>`) является `document`. Но `document` – это не узел-элемент, так что `parentNode` вернет его, а `parentElement` нет. Эта деталь может быть полезна, если надо пройти вверх по цепочке родителей от произвольного элемента `elem` к `<html>`, но не до `document`:

```
while(elem = elem.parentElement) { // идти наверх до <html>
  alert( elem );
}
```

Изменим один из примеров выше: заменим `childNodes` на `children`. Теперь цикл выводит только элементы:

```

<html>
<body>
  <div>Начало</div>

  <ul>
    <li>Информация</li>
  </ul>

  <div>Конец</div>

  <script>
    for (let elem of document.body.children) {
      alert(elem); // DIV, UL, DIV, SCRIPT
    }
  </script>
  ...
</body>
</html>

```

До сих пор рассматривались основные навигационные ссылки. Некоторые типы DOM-элементов предоставляют для удобства дополнительные свойства, специфичные для их типа. Таблицы – отличный пример таких элементов. Элемент `<table>`, в дополнение к свойствам, о которых речь шла выше, поддерживает следующие:

- `table.rows` – коллекция строк `<tr>` таблицы;
- `table.caption/tHead/tFoot` – ссылки на элементы таблицы `<caption>`, `<thead>`, `<tfoot>`;
- `table.tBodies` – коллекция элементов таблицы `<tbody>` (по спецификации их может быть больше одного);
- `<thead>`, `<tfoot>`, `<tbody>` предоставляют свойство `rows`: `tbody.rows` – коллекция строк `<tr>` секции;
- `<tr>`:
  - `tr.cells` – коллекция `<td>` и `<th>` ячеек, находящихся внутри строки `<tr>`;
  - `tr.sectionRowIndex` – номер строки `<tr>` в текущей секции `<thead>/<tbody>/<tfoot>`;
  - `tr.rowIndex` – номер строки `<tr>` в таблице (включая все строки таблицы);
- `<td>` and `<th>`: `td.cellIndex` – номер ячейки в строке `<tr>`.

Пример использования:

```

<table id="table">
  <tr>
    <td>один</td><td>два</td>
  </tr>
  <tr>

```

```

    <td>три</td><td>четыре</td>
  </tr>
</table>

<script>
  // выводит содержимое первой строки, второй ячейки
  alert( table.rows[0].cells[1].innerHTML ) // "два"
</script>

```

Существуют также дополнительные навигационные ссылки для HTML-форм. Они будут рассматриваться позже, при работе с формами.

Свойства навигации по DOM удобны, когда элементы расположены рядом. Чтобы получить произвольный элемент страницы, можно использовать дополнительные методы поиска.

### **document.getElementById или просто id**

Когда у элемента есть атрибут `id`, значение атрибута используется в качестве имени глобальной переменной. С её помощью можно обратиться к элементу напрямую (если в имени переменной используется дефис '-', то она доступна через квадратные скобки [...]):

```

<div id="elem">
  <div id="elem-content">Элемент</div>
</div>

<script>
  alert(elem); // DOM-элемент с id="elem"
  alert(window.elem); // доступ к глобальной переменной тоже работает

  alert(window['elem-content']);
</script>

```

Это поведение соответствует стандарту, но поддерживается в основном для совместимости. Браузер пытается помочь, смешивая пространства имён JS и DOM. Это подходит для простого кода, но возможны конфликты. Если осуществляется работа с JS-кодом, не видя HTML, не очевидно, откуда возьмётся переменная. Если объявить переменную с тем же именем, она будет иметь приоритет:

```

<div id="elem"></div>

<script>
  let elem = 5;

  alert(elem); // 5
</script>

```

Лучшая альтернатива — использовать специальный метод `document.getElementById(id)`. Например:

```
<div id="elem">
  <div id="elem-content">Элемент</div>
</div>

<script>
  let elem = document.getElementById('elem');

  elem.style.background = 'red';
</script>
```

Далее в примерах часто будет использоваться прямое обращение через `id`, но это только для краткости. В реальных проектах предпочтителен метод `document.getElementById`.

Значение `id` должно быть уникальным. В документе может быть только один элемент с данным `id`. Если в документе есть несколько элементов с одинаковым значением `id`, то поведение методов поиска непредсказуемо. Браузер может вернуть любой из них случайным образом. Поэтому придерживайтесь правила сохранения уникальности `id`.

Метод `getElementById` можно вызвать только для объекта `document`. Он осуществляет поиск по `id` по всему документу.

### Методы `querySelectorAll` и `querySelector`

Самый универсальный метод поиска — это `elem.querySelectorAll(css)`, он возвращает все элементы внутри `elem`, удовлетворяющие данному CSS-селектору. Следующий запрос получает все элементы `<li>`, которые являются последними потомками в `<ul>`:

```
<ul>
  <li>Этот</li>
  <li>тест</li>
</ul>
<ul>
  <li>полностью</li>
  <li>пройден</li>
</ul>
<script>
  let elements = document.querySelectorAll('ul > li:last-child');

  for (let elem of elements) {
    alert(elem.innerHTML); // "тест", "пройден"
  }
</script>
```

С этим методом можно использовать любой CSS-селектор. Псевдоклассы в CSS-селекторе, в частности `:hover` и `:active`, также поддерживаются. Например, `document.querySelectorAll(':hover')` вернёт коллекцию (в порядке вложенности: от внешнего к внутреннему) из текущих элементов под курсором мыши.

Метод `elem.querySelector(css)` возвращает первый элемент, соответствующий данному CSS-селектору. Иначе говоря, результат такой же, как при вызове `elem.querySelectorAll(css)[0]`, но он сначала найдёт все элементы, а потом возьмёт первый, в то время как `elem.querySelector` найдёт только первый и остановится. Это быстрее, кроме того, его короче писать.

### Метод `matches`

Предыдущие методы искали по DOM. Метод `elem.matches(css)` ничего не ищет, а проверяет, удовлетворяет ли `elem` CSS-селектору, и возвращает `true` или `false`. Этот метод удобен, когда надо перебрать элементы (например, в массиве или в чём-то подобном) и выбрать те из них, которые интересуют. Например:

```
<a href="http://example.com/file.zip">...</a>
<a href="http://ya.ru">...</a>

<script>
  // может быть любая коллекция вместо document.body.children
  for (let elem of document.body.children) {
    if (elem.matches('a[href$="zip"]')) {
      alert("Ссылка на архив: " + elem.href );
    }
  }
</script>
```

### Метод `closest`

Предки элемента — родитель, родитель родителя, его родитель и так далее. Вместе они образуют цепочку иерархии от элемента до вершины. Метод `elem.closest(css)` ищет ближайшего предка, который соответствует CSS-селектору. Сам элемент также включается в поиск. Другими словами, метод `closest` поднимается вверх от элемента и проверяет каждого из родителей. Если он соответствует селектору, поиск прекращается. Метод возвращает либо предка, либо `null`, если такой элемент не найден. Например:

```
<h1>Содержание</h1>

<div class="contents">
  <ul class="book">
    <li class="chapter">Глава 1</li>
    <li class="chapter">Глава 2</li>
  </ul>
```

```

</div>

<script>
  let chapter = document.querySelector('.chapter'); // LI

  alert(chapter.closest('.book')); // UL
  alert(chapter.closest('.contents')); // DIV

  alert(chapter.closest('h1')); // null (потому что h1 - не предок)
</script>

```

### Методы getElementsBy\*

Существуют также другие методы поиска элементов по тегу, классу и так далее. На данный момент, они скорее исторические, так как `querySelector` более чем эффективен. Здесь рассмотрим их для полноты картины, также можно встретить их в старом коде.

- `elem.getElementsByTagName(tag)` ищет элементы с данным тегом и возвращает их коллекцию. Передав "\*" вместо тега, можно получить всех потомков.
- `elem.getElementsByClassName(className)` возвращает элементы, которые имеют данный CSS-класс.
- `document.getElementsByName(name)` возвращает элементы с заданным атрибутом `name`. Очень редко используется.

Например, получить все элементы `div` в документе:

```
let divs = document.getElementsByTagName('div');
```

Найдём все `input` в таблице:

```

<table id="table">
  <tr>
    <td>Ваш возраст:</td>

    <td>
      <label>
        <input type="radio" name="age" value="young" checked> младше
18
      </label>
      <label>
        <input type="radio" name="age" value="mature"> от 18 до 50
      </label>
      <label>
        <input type="radio" name="age" value="senior"> старше 60
      </label>
    </td>
  </tr>
</table>

<script>

```

```

let inputs = table.getElementsByTagName('input');

for (let input of inputs) {
    alert( input.value + ': ' + input.checked );
}
</script>

```

Одна из самых частых ошибок – это забыть про букву "s". То есть пробовать вызывать метод `getElementByTagName` вместо `getElementsByTagName`. Буква "s" отсутствует в названии метода `getElementById`, так как в данном случае возвращает один элемент. Но `getElementsByTagName` вернёт список элементов, поэтому "s" обязательна.

Метод возвращает коллекцию, а не элемент. Поэтому, другая распространённая ошибка – написать:

```

// не работает
document.getElementsByTagName('input').value = 5;

```

Попытка присвоить значение коллекции, а не элементам внутри неё, не сработает. Нужно перебрать коллекцию в цикле или получить элемент по номеру и уже ему присваивать значение, например, так:

```

// работает (если есть input)
document.getElementsByTagName('input')[0].value = 5;

```

Ищем элементы с классом `.article`:

```

<form name="my-form">
  <div class="article">Article</div>
  <div class="long article">Long article</div>
</form>

<script>
  // поиск по имени атрибута
  let form = document.getElementsByName('my-form')[0];

  // поиск по классу внутри form
  let articles = form.getElementsByClassName('article');
  alert(articles.length); // 2, найдены два элемента с классом article
</script>

```

Все методы `"getElementsByTagName"` возвращают живую коллекцию. Такие коллекции всегда отражают текущее состояние документа и автоматически обновляются при его изменении. В приведённом ниже примере есть два скрипта.

1. Первый создаёт ссылку на коллекцию `<div>`. На этот момент её длина равна 1.
2. Второй скрипт запускается после того, как браузер встречает ещё один `<div>`, теперь её длина – 2.

```
<div>First div</div>

<script>
  let divs = document.getElementsByTagName('div');
  alert(divs.length); // 1
</script>

<div>Second div</div>

<script>
  alert(divs.length); // 2
</script>
```

Напротив, `querySelectorAll` возвращает статическую коллекцию. Это похоже на фиксированный массив элементов. Если использовать его в примере выше, то оба скрипта вернут длину коллекции, равную 1:

```
<div>First div</div>

<script>
  let divs = document.querySelectorAll('div');
  alert(divs.length); // 1
</script>

<div>Second div</div>

<script>
  alert(divs.length); // 1
</script>
```

Теперь видна разница. Длина статической коллекции не изменилась после появления нового `div` в документе.

### 13. Свойства узлов: тип, тег и содержимое.

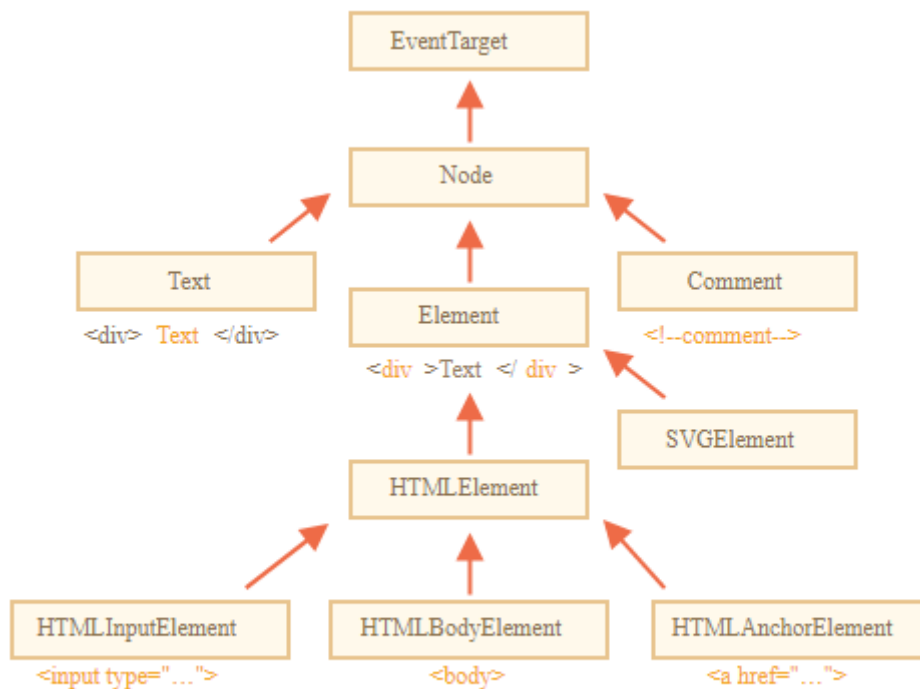
Разберём подробнее, что собой представляют DOM-узлы и изучим их основные свойства.

#### Классы DOM-узлов

У разных DOM-узлов могут быть разные свойства. Например, у узла, соответствующего тегу `<a>`, есть свойства, связанные со ссылками, а у соответствующего тегу `<input>` – свойства, связанные с полем ввода и т.д. Текстовые узлы отличаются от узлов-элементов. Но у них есть общие свойства



и методы, потому что все классы DOM-узлов образуют единую иерархию. Каждый DOM-узел принадлежит соответствующему встроенному классу. Корнем иерархии является EventTarget, от него наследует Node и остальные DOM-узлы. На рисунке ниже изображены основные классы:



Существуют следующие классы:

- EventTarget – это корневой «абстрактный» класс. Объекты этого класса никогда не создаются. Он служит основой, благодаря которой все DOM-узлы поддерживают так называемые «события», о которых рассмотрим позже.
- Node – также является «абстрактным» классом, и служит основой для DOM-узлов. Он обеспечивает базовую функциональность: parentNode, nextSibling, childNodes и т.д. (это геттеры). Объекты класса Node никогда не создаются. Но есть определенные классы узлов, которые наследуют от него: Text – для текстовых узлов, Element – для узлов-элементов и более экзотический Comment – для узлов-комментариев.
- Element – это базовый класс для DOM-элементов. Он обеспечивает навигацию на уровне элементов: nextElementSibling, children и методы поиска: getElementsByTagName, querySelector. Браузер поддерживает не только HTML, но также XML и SVG. Класс Element служит базой для следующих классов: SVGElement, XMLElement и HTMLElement.
- HTMLElement – является базовым классом для всех остальных HTML-элементов. От него наследуют конкретные элементы:
  - HTMLInputElement – класс для тега <input>,
  - HTMLBodyElement – класс для тега <body>,

- `HTMLAnchorElement` – класс для тега `<a>`,
- ...и т.д, каждому тегу соответствует свой класс, который предоставляет определенные свойства и методы.

Таким образом, полный набор свойств и методов данного узла собирается в результате наследования. Рассмотрим DOM-объект для тега `<input>`. Он принадлежит классу `HTMLInputElement`. Он получает свойства и методы из (в порядке наследования):

- `HTMLInputElement` – этот класс предоставляет специфичные для элементов формы свойства;
- `HTMLElement` – предоставляет общие для HTML-элементов методы (и геттеры/сеттеры);
- `Element` – предоставляет типовые методы элемента;
- `Node` – предоставляет общие свойства DOM-узлов;
- `EventTarget` – обеспечивает поддержку событий (поговорим о них дальше);
- `Object` – доступны методы «обычного объекта», такие как `hasOwnProperty`.

Для того, чтобы узнать имя класса DOM-узла, вспомним, что обычно у объекта есть свойство `constructor`. Оно ссылается на конструктор класса, и в свойстве `constructor.name` содержится его имя:

```
alert( document.body.constructor.name ); // HTMLBodyElement
```

Также можно просто привести его к строке:

```
alert( document.body ); // [object HTMLBodyElement]
```

Проверить наследование можно также при помощи `instanceof`:

```
alert( document.body instanceof HTMLBodyElement ); // true
alert( document.body instanceof HTMLElement ); // true
alert( document.body instanceof Element ); // true
alert( document.body instanceof Node ); // true
alert( document.body instanceof EventTarget ); // true
```

Как видно, DOM-узлы – это обычные JavaScript объекты. Для наследования они используют классы, основанные на прототипах. В этом легко убедиться, если вывести в консоли браузера любой элемент через `console.dir(elem)`. Или даже напрямую обратиться к методам, которые хранятся в `HTMLElement.prototype`, `Element.prototype` и т.д.

### **`console.dir(elem)` и `console.log(elem)`**

Большинство браузеров поддерживают в инструментах разработчика две команды: `console.log` и `console.dir`. Они выводят свои аргументы в консоль. Для JavaScript-объектов эти команды обычно выводят одно и то же. Но для

DOM-элементов они работают по-разному: `console.log(elem)` выводит элемент в виде DOM-дерева; `console.dir(elem)` выводит элемент в виде DOM-объекта, что удобно для анализа его свойств.

Чтобы узнать тип узла, можно использовать метод `instanceof` и другие способы проверить класс, но иногда `nodeType` проще использовать. Нельзя изменить значение `nodeType`, только прочитать его.

### Свойства `nodeName` и `tagName`

Получив DOM-узел, можно узнать имя его тега из свойств `nodeName` и `tagName`. Например:

```
alert( document.body.nodeName ); // BODY
alert( document.body.tagName ); // BODY
```

Разница между `tagName` и `nodeName` заключается в следующем:

- свойство `tagName` есть только у элементов `Element`;
- свойство `nodeName` определено для любых узлов `Node`:
  - для элементов оно равно `tagName`;
  - для остальных типов узлов (текст, комментариев и т.д.) оно содержит строку с типом узла.

Другими словами, свойство `tagName` есть только у узлов-элементов (поскольку они происходят от класса `Element`), а `nodeName` может что-то сказать о других типах узлов. Например, сравним `tagName` и `nodeName` на примере объекта `document` и узла-комментария:

```
<body><!-- комментарий -->

<script>
  // для комментария
  alert( document.body.firstChild.tagName ); // undefined (не
элемент)
  alert( document.body.firstChild.nodeName ); // #comment

  // for document
  alert( document.tagName ); // undefined (не элемент)
  alert( document.nodeName ); // #document
</script>
</body>
```

Если нужны только элементами, то можно использовать `tagName` или `nodeName`, нет разницы.

В браузере существуют два режима обработки документа: HTML и XML. HTML-режим обычно используется для веб-страниц. XML-режим включается, если браузер получает XML-документ с заголовком: `Content-Type: application/xml+xhtml`.

В HTML-режиме значения tagName/nodeName всегда записаны в верхнем регистре. Будет выведено BODY вне зависимости от того, как записан тег в HTML <body> или <BoDy>.

В XML-режиме регистр сохраняется «как есть». В настоящее время XML-режим применяется редко.

### Свойство innerHTML

Свойство innerHTML позволяет получить HTML-содержимое элемента в виде строки. Также можно изменять его. Это один из самых мощных способов менять содержимое на странице. Пример ниже показывает содержимое document.body, а затем полностью заменяет его:

```
<body>
  <p>Параграф</p>
  <div>DIV</div>

  <script>
    alert( document.body.innerHTML ); // читаем текущее содержимое
    document.body.innerHTML = 'Новый BODY!'; // заменяем содержимое
  </script>
</body>
```

Можно вставить некорректный HTML, браузер исправит ошибки:

```
<body>

  <script>
    document.body.innerHTML = '<b>тест'; // не закрыт тег
    alert( document.body.innerHTML ); // <b>тест</b> (исправлено)
  </script>

</body>
```

Если innerHTML вставляет в документ тег <script> – он становится частью HTML, но не запускается.

Можно добавить HTML к элементу, используя elem.innerHTML+="ещё html":

```
chatDiv.innerHTML += "<div>Привет<img src='smile.gif'/> !</div>";
chatDiv.innerHTML += "Как дела?";
```

На практике этим следует пользоваться с большой осторожностью, так как фактически происходит не добавление, а перезапись. Технически эти две строки делают одно и то же:

```
elem.innerHTML += "...";
```

```
// это более короткая запись для:  
elem.innerHTML = elem.innerHTML + "..."
```

Другими словами, `innerHTML+=` делает следующее:

1. Старое содержимое удаляется.
2. На его место становится новое значение `innerHTML` (с добавленной строкой).

Так как содержимое «обнуляется» и переписывается заново, все изображения и другие ресурсы будут перезагружены. В примере `chatDiv` выше строка `chatDiv.innerHTML+="Как дела?"` заново создаёт содержимое HTML и перезагружает `smile.gif`. Если в `chatDiv` много текста и изображений, то эта перезагрузка будет очень заметна.

Есть и другие побочные эффекты. Например, если существующий текст выделен мышкой, то при переписывании `innerHTML` большинство браузеров снимут выделение. А если это поле ввода `<input>` с текстом, введенным пользователем, то текст будет удалён.

### Свойство `outerHTML`

Свойство `outerHTML` содержит HTML элемента целиком. Это как `innerHTML` плюс сам элемент. Рассмотрим пример:

```
<div id="elem">Привет <b>Мир</b></div>  
  
<script>  
  alert(elem.outerHTML); // <div id="elem">Привет <b>Мир</b></div>  
</script>
```

Будьте осторожны: в отличие от `innerHTML`, запись в `outerHTML` не изменяет элемент. Вместо этого элемент заменяется целиком во внешнем контексте. Рассмотрим пример:

```
<div>Привет, мир!</div>  
  
<script>  
  let div = document.querySelector('div');  
  
  div.outerHTML = '<p>Новый элемент</p>'; // (*)  
  
  alert(div.outerHTML); // <div>Привет, мир!</div> (**)  
</script>
```

В строке (\*) `div` заменен на `<p>Новый элемент</p>`. Во внешнем документе располагается новое содержимое вместо `<div>`. Но, как видно в строке (\*\*), старая переменная `div` осталась прежней. Это потому, что использование `outerHTML` не изменяет DOM-элемент, а удаляет его из

внешнего контекста и вставляет вместо него новый HTML-код. То есть, при `div.outerHTML=...` произошло следующее:

- `div` был удалён из документа;
- вместо него был вставлен другой HTML `<p>A new element</p>`;
- в `div` осталось старое значение. Новый HTML не сохранён ни в какой переменной.

Здесь легко сделать ошибку: заменить `div.outerHTML`, а потом продолжить работать с `div`, как будто там новое содержимое. Но это не так. Подобное верно для `innerHTML`, но не для `outerHTML`.

Можно писать в `elem.outerHTML`, но это не меняет элемент, в который пишем. Вместо этого создается новый HTML на его месте. Можно получить ссылки на новые элементы, обратившись к DOM.

### Свойства `nodeValue` и `data`

Свойство `innerHTML` есть только у узлов-элементов. У других типов узлов, в частности, у текстовых, есть свои аналоги: свойства `nodeValue` и `data`. Эти свойства очень похожи при использовании, есть лишь небольшие различия в спецификации. Будем использовать `data`, потому что оно короче. Прочитаем содержимое текстового узла и комментария:

```
<body>
Привет
<!-- Комментарий -->
<script>
  let text = document.body.firstChild;
  alert(text.data); // Привет

  let comment = text.nextSibling;
  alert(comment.data); // Комментарий
</script>
</body>
```

Иногда комментарии используют для вставки информации и инструкций шаблонизатора в HTML, как в примере ниже:

```
<!-- if isAdmin -->
  <div>Добро пожаловать, Admin!</div>
<!-- /if -->
```

Затем JavaScript может прочитать это из свойства `data` и обработать инструкции.

### Свойство `textContent`

Свойство `textContent` предоставляет доступ к тексту внутри элемента за вычетом всех `<тегов>`. Например:

```

<div id="news">
  <h1>Срочно в номер!</h1>
  <p>Марсиане атаковали человечество!</p>
</div>

<script>
  // Срочно в номер! Марсиане атаковали человечество!
  alert(news.textContent);
</script>

```

Возвращается только текст, как если бы все <теги> были вырезаны, но текст в них остался. На практике редко появляется необходимость читать текст таким образом. Намного полезнее возможность записывать текст в свойство `textContent`, т.к. оно позволяет писать текст «безопасным способом».

Допустим есть произвольная строка, введенная пользователем, и надо показать её. С `innerHTML` вставка происходит «как HTML», со всеми HTML-тегами. С `textContent` вставка получается «как текст», все символы трактуются буквально. Сравним два тега `div`:

```

<div id="elem1"></div>
<div id="elem2"></div>

<script>
  let name = prompt("Введите ваше имя?", "<b>Винни-пух!</b>");

  elem1.innerHTML = name;
  elem2.textContent = name;
</script>

```

1. В первый `<div>` имя приходит «как HTML»: все теги стали именно тегами, поэтому отображается имя, выделенное жирным шрифтом.
2. Во второй `<div>` имя приходит «как текст», поэтому отображается `<b>Винни-пух!</b>`.

В большинстве случаев надо получить от пользователя текст и чтобы он интерпретировался как текст. Не надо, чтобы на сайте появлялся произвольный HTML-код. Присваивание через `textContent` – один из способов от этого защититься.

### Свойство «hidden»

Атрибут и DOM-свойство «hidden» указывает на то, виден ли элемент или нет. Можно использовать его в HTML или назначать при помощи JavaScript, как в примере ниже:

```

<div>Оба тега DIV внизу невидимы</div>

<div hidden>С атрибутом "hidden"</div>

```

```
<div id="elem">С назначенным JavaScript свойством "hidden"</div>

<script>
  elem.hidden = true;
</script>
```

Технически, hidden работает так же, как style="display:none". Но его применение проще. Мигающий элемент:

```
<div id="elem">Мигающий элемент</div>

<script>
  setInterval(() => elem.hidden = !elem.hidden, 1000);
</script>
```

### Другие свойства

У DOM-элементов есть дополнительные свойства, в частности, зависящие от класса:

- value – значение для <input>, <select> и <textarea> (HTMLInputElement, HTMLSelectElement и др.);
- href – адрес ссылки «href» для <a href="..."> (HTMLAnchorElement);
- id – значение атрибута «id» для всех элементов (HTMLElement) и многие другие.

Например:

```
<input type="text" id="elem" value="значение">

<script>
  alert(elem.type); // "text"
  alert(elem.id); // "elem"
  alert(elem.value); // значение
</script>
```

Большинство стандартных HTML-атрибутов имеют соответствующее DOM-свойство, и можно получить к нему доступ. Если надо узнать полный список поддерживаемых свойств для данного класса, можно найти их в спецификации. Например, класс HTMLInputElement описывается здесь: <https://html.spec.whatwg.org/#htmlinputelement>. Также можно вывести элемент в консоль, используя console.dir(elem), и прочитать все свойства. Или исследовать «свойства DOM» во вкладке Elements браузерных инструментов разработчика.

## 14. Атрибуты и DOM-свойства.

Когда браузер загружает страницу, он «читает» («парсит») 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.

### DOM-свойства

Ранее рассматривались встроенные DOM-свойства. Но можно добавить своё собственное свойство. DOM-узлы – это обычные объекты JavaScript. Можно их изменять. Например, создадим новое свойство для `document.body`:

```
document.body.myData = {  
  name: 'Caesar',  
  title: 'Imperator'  
};  
  
alert(document.body.myData.title); // Imperator
```

Можно добавить и метод:

```
document.body.sayTagName = function() {  
  alert(this.tagName);  
};  
  
document.body.sayTagName(); // BODY (this = document.body)
```

Также можно изменять встроенные прототипы, такие как `Element.prototype` и добавлять новые методы ко всем элементам:

```
Element.prototype.sayHi = function() {  
  alert(`Hello, I'm ${this.tagName}`);  
};  
  
document.documentElement.sayHi(); // Hello, I'm HTML  
document.body.sayHi(); // Hello, I'm BODY
```

Итак, DOM-свойства и методы ведут себя так же, как и обычные объекты JavaScript: им можно присвоить любое значение; они регистрозависимы (нужно писать `elem.nodeType`, не `elem.NoDeType`).

### HTML-атрибуты

В HTML у тегов могут быть атрибуты. Когда браузер парсит HTML, чтобы создать DOM-объекты для тегов, он распознаёт стандартные атрибуты и создаёт DOM-свойства для них. Таким образом, когда у элемента есть `id` или другой стандартный атрибут, создаётся соответствующее свойство. Но этого не происходит, если атрибут нестандартный. Например:

```
<body id="test" something="non-standard">  
  <script>
```

```
    alert(document.body.id); // test

    alert(document.body.something); // undefined
</script>
</body>
```

Пожалуйста, учтите, что стандартный атрибут для одного тега может быть нестандартным для другого. Например, атрибут "type" является стандартным для элемента <input> (HTMLInputElement), но не является стандартным для <body> (HTMLBodyElement). Стандартные атрибуты описаны в спецификации для соответствующего класса элемента. Можно увидеть это на примере ниже:

```
<body id="body" type="...">
  <input id="input" type="text">
  <script>
    alert(input.type); // text
    alert(body.type); // undefined
  </script>
</body>
```

Таким образом, для нестандартных атрибутов не будет соответствующих DOM-свойств. Все атрибуты доступны с помощью следующих методов:

- elem.hasAttribute(name) – проверяет наличие атрибута;
- elem.getAttribute(name) – получает значение атрибута;
- elem.setAttribute(name, value) – устанавливает значение атрибута;
- elem.removeAttribute(name) – удаляет атрибут.

Этим методы работают именно с тем, что написано в HTML.

Кроме этого, получить все атрибуты элемента можно с помощью свойства elem.attributes: коллекция объектов, которая принадлежит ко встроенному классу Attr со свойствами name и value. Вот демонстрация чтения нестандартного свойства:

```
<body something="non-standard">
  <script>
    alert(document.body.getAttribute('something')); // non-standard
  </script>
</body>
```

У HTML-атрибутов есть следующие особенности: их имена регистронезависимы (id то же самое, что и ID); их значения всегда являются строками. Расширенная демонстрация работы с атрибутами:

```
<body>
  <div id="elem" about="Elephant"></div>
```

```

<script>
  alert( elem.getAttribute('About') ); // (1) 'Elephant', чтение

  elem.setAttribute('Test', 123); // (2), запись

  alert( elem.outerHTML ); // (3), посмотрим, есть ли атрибут в HTML
(да)

  for (let attr of elem.attributes) { // (4) весь список
    alert( `${attr.name} = ${attr.value}` );
  }
</script>
</body>

```

Обратите внимание:

1. `getAttribute('About')` – здесь первая буква заглавная, а в HTML – строчная. Но это не важно: имена атрибутов регистронезависимы.
2. Можно присвоить что угодно атрибуту, но это станет строкой. Поэтому в этой строчке получаем значение "123".
3. Все атрибуты, в том числе те, которые были установлены, видны в `outerHTML`.
4. Коллекция `attributes` является перебираемой. В ней есть все атрибуты элемента (стандартные и нестандартные) в виде объектов со свойствами `name` и `value`.

### Синхронизация между атрибутами и свойствами

Когда стандартный атрибут изменяется, соответствующее свойство автоматически обновляется. Это работает и в обратную сторону (за некоторыми исключениями). В примере ниже `id` модифицируется как атрибут, и можно увидеть, что свойство также изменено. То же самое работает и в обратную сторону:

```

<input>

<script>
  let input = document.querySelector('input');

  // атрибут => свойство
  input.setAttribute('id', 'id');
  alert(input.id); // id (обновлено)

  // свойство => атрибут
  input.id = 'newId';
  alert(input.getAttribute('id')); // newId (обновлено)
</script>

```

Но есть и исключения, например, `input.value` синхронизируется только в одну сторону – атрибут → значение, но не в обратную:

```
<input>

<script>
  let input = document.querySelector('input');

  // атрибут => значение
  input.setAttribute('value', 'text');
  alert(input.value); // text

  // свойство => атрибут
  input.value = 'newValue';
  alert(input.getAttribute('value')); // text
</script>
```

В примере выше изменение атрибута `value` обновило свойство, но изменение свойства не повлияло на атрибут. Иногда эта «особенность» может пригодиться, потому что действия пользователя могут приводить к изменениям `value`, и если после этого надо восстановить «оригинальное» значение из HTML, оно будет в атрибуте.

DOM-свойства не всегда являются строками. Например, свойство `input.checked` (для чекбоксов) имеет логический тип:

```
<input id="input" type="checkbox" checked> checkbox

<script>
  alert(input.getAttribute('checked')); // значение атрибута: пустая
  строка
  alert(input.checked); // значение свойства: true
</script>
```

Есть и другие примеры. Атрибут `style` – строка, но свойство `style` является объектом:

```
<div id="div" style="color:red;font-size:120%">Hello</div>

<script>
  // строка
  alert(div.getAttribute('style')); // color:red;font-size:120%

  // объект
  alert(div.style); // [object CSSStyleDeclaration]
  alert(div.style.color); // red
</script>
```

Большинство свойств, всё же, строки, но могут отличаться от атрибутов. Например, DOM-свойство href всегда содержит полный URL, даже если атрибут содержит относительный URL или просто #hash. Ниже пример:

```
<a id="a" href="#hello">link</a>
<script>
  // атрибут
  alert(a.getAttribute('href')); // #hello

  // свойство
  alert(a.href ); // полный URL в виде http://site.com/page#hello
</script>
```

Если же нужно значение href или любого другого атрибута в точности, как оно записано в HTML, можно воспользоваться `getAttribute`.

### Нестандартные атрибуты, dataset

Нестандартные атрибуты используются для передачи пользовательских данных из HTML в JavaScript, или чтобы «помечать» HTML-элементы для JavaScript. Пример:

```
<!-- пометить div, чтобы показать здесь поле "name" -->
<div show-info="name"></div>
<!-- а здесь возраст "age" -->
<div show-info="age"></div>

<script>
  // код находит элемент с пометкой и показывает запрошенную
  информацию
  let user = {
    name: "Pete",
    age: 25
  };

  for(let div of document.querySelectorAll('[show-info]')) {
    // вставить соответствующую информацию в поле
    let field = div.getAttribute('show-info');
    div.innerHTML = user[field]; // сначала Pete в name, потом 25 в
age
  }
</script>
```

Также они могут быть использованы, чтобы стилизовать элементы. Например, здесь для состояния заказа используется атрибут `order-state`:

```
<style>
  /* стили зависят от пользовательского атрибута "order-state" */
  .order[order-state="new"] {
    color: green;
  }
</style>
```

```

    }

    .order[order-state="pending"] {
        color: blue;
    }

    .order[order-state="canceled"] {
        color: red;
    }
}
</style>

<div class="order" order-state="new">
    A new order.
</div>

<div class="order" order-state="pending">
    A pending order.
</div>

<div class="order" order-state="canceled">
    A canceled order.
</div>

```

Атрибут могут быть предпочтительнее таких классов, как `.order-state-new`, `.order-state-pending`, `order-state-canceled`, потому, что атрибутом удобнее управлять. Состояние может быть изменено достаточно просто (не надо удалять старый/добавлять новый класса):

```
div.setAttribute('order-state', 'canceled');
```

Но с пользовательскими атрибутами могут возникнуть проблемы, например, если был использован нестандартный атрибут, а позже он появится в стандарте и будет выполнять какую-то функцию.

Чтобы избежать конфликтов, существуют атрибуты вида `data-*`. Все атрибуты, начинающиеся с префикса «`data-`», зарезервированы для использования программистами. Они доступны в свойстве `dataset`. Например, если у `elem` есть атрибут `"data-about"`, то обратиться к нему можно как `elem.dataset.about`. Например:

```

<body data-about="Elephants">
<script>
    alert(document.body.dataset.about); // Elephants
</script>

```

Атрибуты, состоящие из нескольких слов, к примеру `data-order-state`, становятся свойствами, записанными с помощью верблюжьей нотации: `dataset.orderState`. Вот переписанный пример «состояния заказа»:

```

<style>
  .order[data-order-state="new"] {
    color: green;
  }

  .order[data-order-state="pending"] {
    color: blue;
  }

  .order[data-order-state="canceled"] {
    color: red;
  }
</style>

<div id="order" class="order" data-order-state="new">
  A new order.
</div>

<script>
  // чтение
  alert(order.dataset.orderState); // new

  // изменение
  order.dataset.orderState = "pending"; // (*)
</script>

```

Использование data-\* атрибутов – валидный, безопасный способ передачи пользовательских данных. Можно не только читать, но и изменять data-атрибуты. Тогда CSS обновит представление соответствующим образом: в примере выше последняя строка (\*) меняет цвет на синий.

## 15. Добавление и удаление DOM-узлов.

Рассмотрим пример, добавим на страницу сообщение:

```

<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 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');
```

В рассматриваемом примере нужно создать 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.";
```

Элемент создан, но пока он только в переменной. Нельзя пока увидеть его на странице, поскольку он не является частью документа.

## Методы вставки

Чтобы `div` появился на странице, 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>
```



Метод для различных вариантов вставки:

- `node.append(...узлы или строки)` – 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.

Вот пример использования этих методов, чтобы добавить новые элементы в список и текст до/после него:

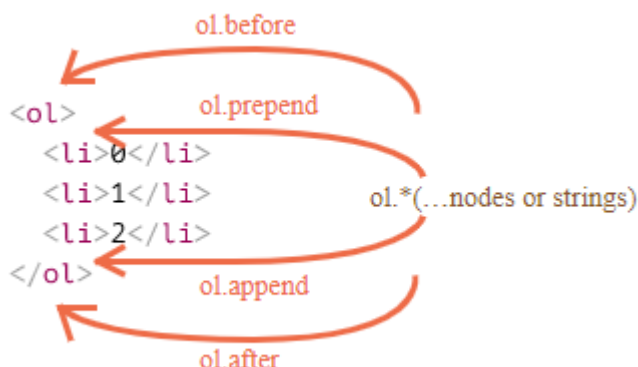
```
<ol id="ol">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>

<script>
  ol.before('before'); // вставить строку "before" перед <ol>
  ol.after('after'); // вставить строку "after" после <ol>

  let liFirst = document.createElement('li');
  liFirst.innerHTML = 'prepend';
  ol.prepend(liFirst); // вставить liFirst в начало <ol>

  let liLast = document.createElement('li');
  liLast.innerHTML = 'append';
  ol.append(liLast); // вставить liLast в конец <ol>
</script>
```

Наглядная иллюстрация того, куда эти методы вставляют:



Итоговый список будет таким:

```
before
<ol id="ol">
  <li>prepend</li>
```

```
<li>0</li>
<li>1</li>
<li>2</li>
<li>append</li>
</ol>
after
```

Эти методы могут вставлять несколько узлов и текстовых фрагментов за один вызов. Например, здесь вставляется строка и элемент:

```
<div id="div"></div>
<script>
  div.before('<p>Привет</p>', document.createElement('hr'));
</script>
```

Весь текст вставляется как текст. Поэтому финальный HTML будет таким:

```
&lt;p&gt;Привет&lt;/p&gt;
<hr>
<div id="div"></div>
```

Другими словами, строки вставляются безопасным способом, как делает это `elem.textContent`. Поэтому эти методы могут использоваться только для вставки DOM-узлов или текстовых фрагментов.

Если надо вставить HTML именно «как html», со всеми тегами и прочим, как делает это `elem.innerHTML`, тогда надо использовать другой универсальный метод: `elem.insertAdjacentHTML(where, html)`.

Первый параметр – это специальное слово, указывающее, куда по отношению к `elem` производить вставку. Значение должно быть одним из следующих:

- "beforebegin" – вставить html непосредственно перед `elem`,
- "afterbegin" – вставить html в начало `elem`,
- "beforeend" – вставить html в конец `elem`,
- "afterend" – вставить html непосредственно после `elem`.

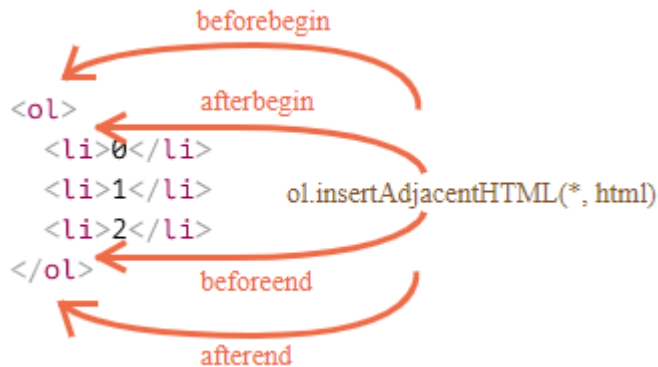
Второй параметр – это HTML-строка, которая будет вставлена именно «как HTML». Например:

```
<div id="div"></div>
<script>
  div.insertAdjacentHTML('beforebegin', '<p>Привет</p>');
  div.insertAdjacentHTML('afterend', '<p>Пока</p>');
</script>
```

Приведёт к:

```
<p>Привет</p>
<div id="div"></div>
<p>Пока</p>
```

Так можно добавлять произвольный HTML на страницу. Варианты вставки:



Есть еще два схожих метода:

- `elem.insertAdjacentText(where, text)` – такой же синтаксис, но строка `text` вставляется «как текст», вместо HTML,
- `elem.insertAdjacentElement(where, elem)` – такой же синтаксис, но вставляет элемент `elem`.

На практике часто используется только `insertAdjacentHTML`. Потому что для элементов и текста есть методы `append/prepend/before/after` – их быстрее написать, и они могут вставлять как узлы, так и текст. Альтернативный вариант показа сообщения:

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

<script>
  document.body.insertAdjacentHTML("afterbegin", `<div class="alert">
    <strong>Всем привет!</strong> Вы прочитали важное сообщение.
  </div>`);
</script>
```

### Удаление узлов

Для удаления узла есть методы `node.remove()`. Например, сделаем так, чтобы 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 нужно переместить элемент в другое место – нет необходимости удалять его со старого. Все методы вставки автоматически удаляют узлы со старых мест. Например, поменяем местами элементы:

```

<div id="first">Первый</div>
<div id="second">Второй</div>
<script>
  second.after(first); // вставляет #first после #second
</script>

```

### Клонирование узлов: cloneNode

Вызов `elem.cloneNode(true)` создаёт клон элемента со всеми атрибутами и дочерними элементами. Если вызвать `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>

```

## Узел DocumentFragment

DocumentFragment является специальным DOM-узлом, который служит обёрткой для передачи списков узлов. Можно добавить к нему другие узлы, но при вставке он «исчезает», вместо него вставляется его содержимое. Например, `getListContent` ниже генерирует фрагмент с элементами `<li>`, которые позже вставляются в `<ul>`:

```

<ul id="ul"></ul>

<script>
function getListContent() {
  let fragment = new DocumentFragment();

  for(let i=1; i<=3; i++) {
    let li = document.createElement('li');
    li.append(i);
    fragment.append(li);
  }

  return fragment;
}

ul.append(getListContent()); // (*)
</script>

```

Обратите внимание, что на последней строке с (\*) добавляется DocumentFragment, но он «исчезает», поэтому структура будет:

```

<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>

```

DocumentFragment редко используется. Не смысла добавлять элементы в специальный вид узла, если вместо этого можно вернуть массив узлов. Переписанный пример:

```

<ul id="ul"></ul>

```

```

<script>
function getListContent() {
    let result = [];

    for(let i=1; i<=3; i++) {
        let li = document.createElement('li');
        li.append(i);
        result.push(li);
    }

    return result;
}

ul.append(...getListContent());
</script>

```

DocumentFragment используется в некоторых других областях, например, для элемента template.

## 16.Стили DOM-узлов.

Как правило, существует два способа задания стилей для элемента:

1. Создать класс в CSS и использовать его: <div class="...">.
2. Писать стили непосредственно в атрибуте style: <div style="...">.

JavaScript может менять и классы и свойство style. Классы – всегда предпочтительный вариант по сравнению со style. Свойством style стоит манипулировать только в том случае, если классы «не могут справиться». Например, использование style является приемлемым, если вычисляются координаты элемента динамически и хотим установить их из JavaScript:

```

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

elem.style.left = left;
elem.style.top = top;

```

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

### className и classList

Изменение класса является одним из наиболее часто используемых действий в скриптах. Свойство "className": elem.className соответствует атрибуту "class". Например:

```

<body class="main page">
<script>

```

```

    alert(document.body.className); // main page
  </script>
</body>

```

Если присвоить что-то `elem.className`, то это заменяет всю строку с классами. Иногда это то, что нужно, но часто надо добавить/удалить один класс. Для этого есть другое свойство: `elem.classList`.

`elem.classList` — это специальный объект с методами для добавления/удаления одного класса. Например:

```

<body class="main page">
  <script>
    document.body.classList.add('article');

    alert(document.body.className); // main page article
  </script>
</body>

```

Так что можно работать как со строкой полного класса, используя `className`, так и с отдельными классами, используя `classList`.

Методы `classList`:

- `elem.classList.add/remove("class")` — добавить/удалить класс.
- `elem.classList.toggle("class")` — добавить класс, если его нет, иначе удалить.
- `elem.classList.contains("class")` — проверка наличия класса, возвращает `true/false`.

Кроме того, `classList` является перебираемым, поэтому можно перечислить все классы при помощи `for..of`:

```

<body class="main page">
  <script>
    for (let name of document.body.classList) {
      alert(name); // main, затем page
    }
  </script>
</body>

```

## Свойство `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');
```

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

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

Иногда нужно добавить свойство стиля, а потом, позже, убрать его. Например, чтобы скрыть элемент, можно задать `elem.style.display = "none"`. Затем можно удалить свойство `style.display`, чтобы вернуться к первоначальному состоянию. Вместо `delete elem.style.display` надо присвоить ему пустую строку: `elem.style.display = ""`.

```
// <body> "мигнёт"  
document.body.style.display = "none"; // скрыть  
  
setTimeout(() => document.body.style.display = "", 1000); // возврат к  
нормальному состоянию
```

Если установить в `style.display` пустую строку, то браузер применит CSS-классы и встроенные стили, как если бы такого свойства `style.display` вообще не было.

Обычно используется `style.*` для присвоения индивидуальных свойств стиля. Нельзя установить список стилей как, например, `div.style="color: red; width: 100px"`, потому что `div.style` – это объект, и он доступен только для чтения. Для задания нескольких стилей в одной строке используется специальное свойство `style.cssText`:

```
<div id="div">Button</div>  
  
<script>  
  div.style.cssText=`color: red !important;  
    background-color: yellow;  
    width: 100px;  
    text-align: center;  
  `;  
  
  alert(div.style.cssText);  
</script>
```



Это свойство редко используется, потому что такое присваивание удаляет все существующие стили: оно не добавляет, а заменяет их. Можно случайно удалить что-то нужное. Но его можно использовать, к примеру, для новых элементов, когда точно известно, что не удалится существующий стиль. То же самое можно сделать установкой атрибута: `div.setAttribute('style', 'color: red...')`.

Не забывайте добавлять к значениям единицы измерения. Например, надо устанавливать `10px`, а не просто `10` в свойство `elem.style.top`, иначе это не сработает:

```
<body>
  <script>
    // не работает
    document.body.style.margin = 20;
    alert(document.body.style.margin); // ''

    document.body.style.margin = '20px';
    alert(document.body.style.margin); // 20px

    alert(document.body.style.marginTop); // 20px
    alert(document.body.style.marginLeft); // 20px
  </script>
</body>
```

Обратите внимание, браузер «распаковывает» свойство `style.margin` в последних строках и выводит `style.marginLeft` и `style.marginTop` из него.

### Вычисленные стили: `getComputedStyle`

Метод `getComputedStyle` позволяет получить текущее значение свойств элемента. Синтаксис:

```
getComputedStyle(element, [pseudo])
```

`element` — элемент, значения для которого нужно получить,  
`pseudo` — указывается, если нужен стиль псевдоэлемента, например `::before`. Пустая строка или отсутствие аргумента означают сам элемент.

Результат вызова — объект со стилями, похожий на `elem.style`, но с учётом всех CSS-классов. Например:

```
<head>
  <style> body { color: red; margin: 5px } </style>
</head>
<body>

  <script>
```

```

let computedStyle = getComputedStyle(document.body);

alert( computedStyle.marginTop ); // 5px
alert( computedStyle.color ); // rgb(255, 0, 0)
</script>

</body>

```

Есть две концепции в CSS:

1. Вычисленное (computed) значение – это то, которое получено после применения всех CSS-правил и CSS-наследования. Например, height:1em или font-size:125%.
2. Окончательное (resolved) значение – непосредственно применяемое к элементу. Значения 1em или 125% являются относительными. Браузер берёт вычисленное значение и делает все единицы измерения фиксированными и абсолютными, например, height:20px or font-size:16px. Для геометрических свойств разрешенные значения могут иметь плавающую точку, например, width:50.5px.

Изначально getComputedStyle был создан для получения вычисленных значений, но окончательные значения гораздо удобнее, и стандарт изменился. Так что, в настоящее время getComputedStyle фактически возвращает окончательное значение свойства, для геометрии оно обычно в пикселях.

Для правильного получения значения нужно указать точное свойство. Например: paddingLeft, marginTop, borderTopWidth. При обращении к сокращенному: padding, margin, border – правильный результат не гарантируется.

Некоторые браузеры (Chrome) отображают 10px в документе ниже, а некоторые (Firefox) – нет:

```

<style>
  body {
    margin: 10px;
  }
</style>
<script>
  let style = getComputedStyle(document.body);
  alert(style.margin); // пустая строка в Firefox
</script>

```

Посещенные ссылки могут быть окрашены с помощью псевдокласса :visited. Но getComputedStyle не дает доступ к этой информации, чтобы произвольная страница не могла определить, посещал ли пользователь ту или иную ссылку, проверив стили.

JavaScript не видит стили, применяемые с помощью :visited. Кроме того, в CSS есть ограничение, которое запрещает в целях безопасности применять к :visited CSS-стили, изменяющие геометрию элемента. Это гарантирует, что

нет обходного пути для вредоносной страницы проверить, была ли ссылка посещена и, следовательно, нарушить конфиденциальность.

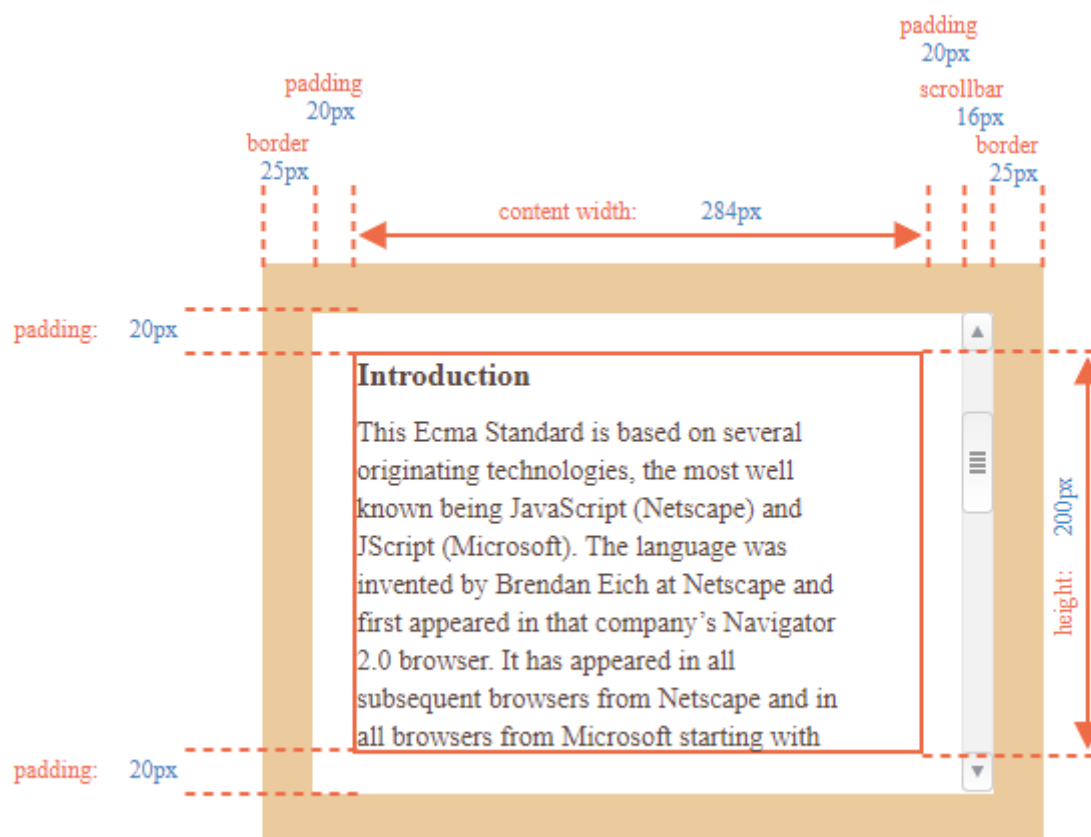
## 17.Размеры и прокрутка элементов и страницы.

Существует множество JavaScript-свойств, которые позволяют считывать информацию об элементе: ширину, высоту и другие геометрические характеристики. Будем называть их «метрики». Они часто требуются, когда нужно передвигать или позиционировать элементы с помощью JavaScript.

В качестве простого примера демонстрации свойств будем использовать следующий элемент:

```
<div id="example">
  ...Текст...
</div>
<style>
  #example {
    width: 300px;
    height: 200px;
    border: 25px solid #E8C48F;
    padding: 20px;
    overflow: auto;
  }
</style>
```

У элемента есть рамка (border), внутренний отступ (padding) и прокрутка. Полный набор характеристик. Обратите внимание, тут нет внешних отступов (margin), потому что они не являются частью элемента, для них нет особых JavaScript-свойств. Результат выглядит так:

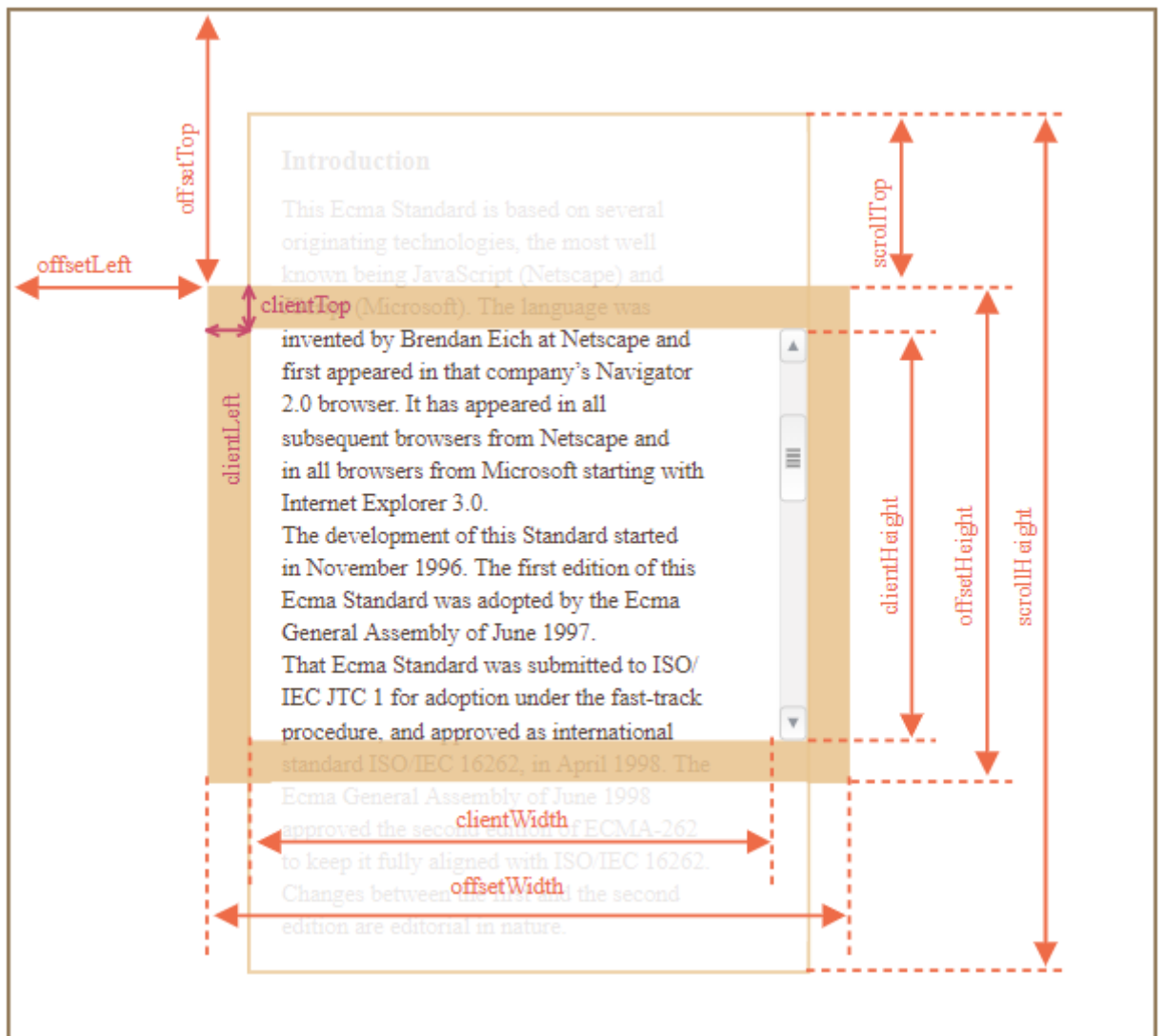


В иллюстрации выше намеренно продемонстрирован самый сложный и полный случай, когда у элемента есть ещё и полоса прокрутки. Некоторые браузеры (не все) отбирают место для неё, забирая его у области, отведённой для содержимого (помечена как «content width» выше).

Таким образом, без учёта полосы прокрутки ширина области содержимого (content width) будет 300px, но если предположить, что ширина полосы прокрутки равна 16px (её точное значение зависит от устройства и браузера), тогда остаётся только  $300 - 16 = 284$ px, и надо это учитывать.

Нижние внутренние отступы padding-bottom изображены на иллюстрациях пустыми, но если элемент содержит много текста, то он будет перекрывать padding-bottom, это нормально.

Вот общая картина с геометрическими свойствами:



Значениями свойств являются числа, подразумевается, что они в пикселях.

Свойства `offsetParent`, `offsetLeft/Top` редко используются, они являются «самыми внешними» метриками.

В свойстве `offsetParent` находится предок элемента, который используется внутри браузера для вычисления координат при рендеринге. То есть, ближайший предок, который удовлетворяет следующим условиям:

- является CSS-позиционированным (CSS-свойство `position` равно `absolute`, `relative`, `fixed` или `sticky`);
- или `<td>`, `<th>`, `<table>`;
- или `<body>`.

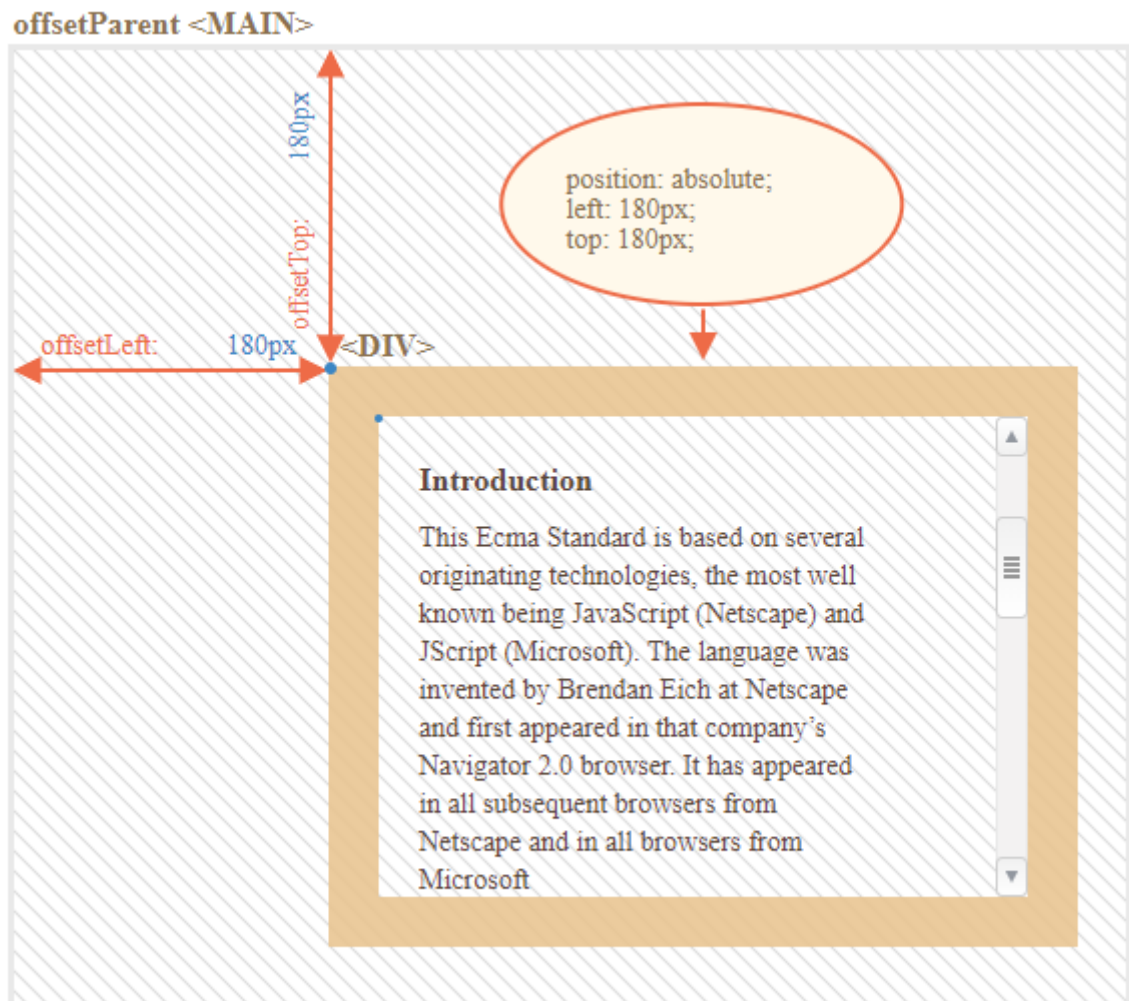
Свойства `offsetLeft/offsetTop` содержат координаты `x/y` относительно верхнего левого угла `offsetParent`. В примере ниже внутренний `<div>` имеет элемент `<main>` в качестве `offsetParent`, а свойства `offsetLeft/offsetTop` являются сдвигами относительно верхнего левого угла (180):

```
<main style="position: relative" id="main">
```

```

<article>
  <div id="example" style="position: absolute; left: 180px; top:
180px">...</div>
</article>
</main>
<script>
  alert(example.offsetParent.id); // main
  alert(example.offsetLeft); // 180 (число, а не строка "180px")
  alert(example.offsetTop); // 180
</script>

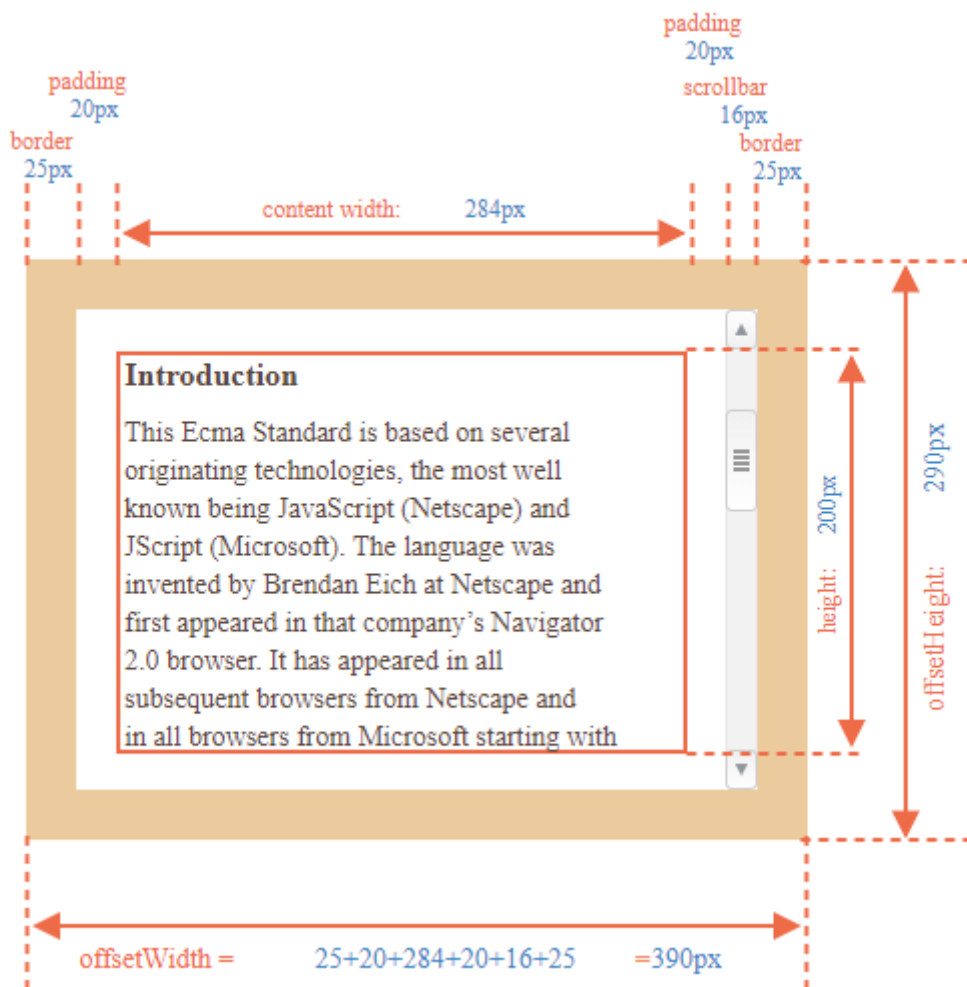
```



Существует несколько ситуаций, когда `offsetParent` равно `null`:

1. Для скрытых элементов (с CSS-свойством `display:none` или когда его нет в документе).
2. Для элементов `<body>` и `<html>`.
3. Для элементов с `position:fixed`.

Свойства `offsetWidth/Height` самые простые. Они содержат «внешнюю» ширину/высоту элемента, то есть его полный размер, включая рамки.



Для рассматриваемого элемента:

- `offsetWidth` = 390 – внешняя ширина блока, её можно получить сложением CSS-ширины (300px), внутренних отступов ( $2 * 20\text{px}$ ) и рамок ( $2 * 25\text{px}$ ).
- `offsetHeight` = 290 – внешняя высота блока.

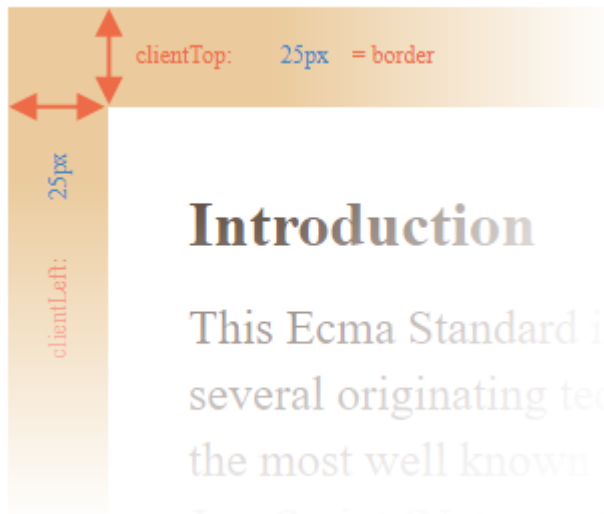
Метрики для не показываемых элементов равны нулю. Координаты и размеры в JavaScript устанавливаются только для видимых элементов. Если элемент (или любой его родитель) имеет `display:none` или отсутствует в документе, то все его метрики равны нулю (или `null`, если это `offsetParent`). Например, свойство `offsetParent` равно `null`, а `offsetWidth` и `offsetHeight` равны 0, когда элемент создан, но ещё не вставлен в документ, или если у элемента (или у его родителя) `display:none`. Это можно использовать, чтобы делать проверку на видимость:

```
function isHidden(elem) {
    return !elem.offsetWidth && !elem.offsetHeight;
}
```

Функция `isHidden` также вернёт `true` для элементов, которые в принципе показываются, но их размеры равны нулю (например, пустые `<div>`).

Внутри элемента есть рамки (border). Для них есть свойства-метрики clientTop и clientLeft. В примере:

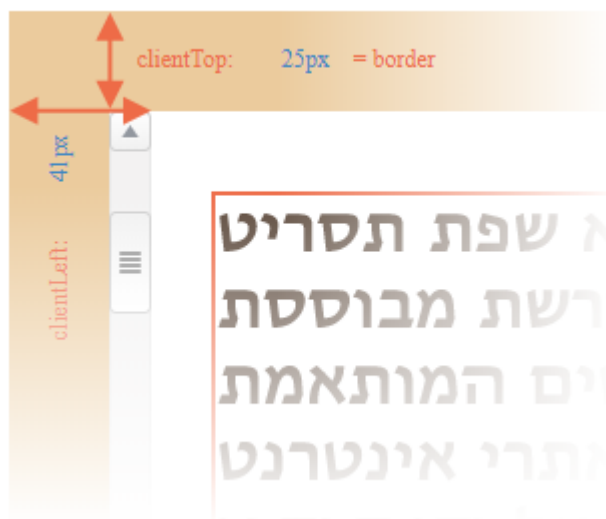
- clientLeft = 25 – ширина левой рамки,
- clientTop = 25 – ширина верхней рамки.



Но на самом деле эти свойства – вовсе не ширины рамок, а отступы внутренней части элемента от внешней. Разница в том, что когда документ располагается справа налево (операционная система на арабском языке или иврите). Полоса прокрутки в этом случае находится слева, и тогда свойство clientLeft включает в себя ещё и ширину полосы прокрутки. В

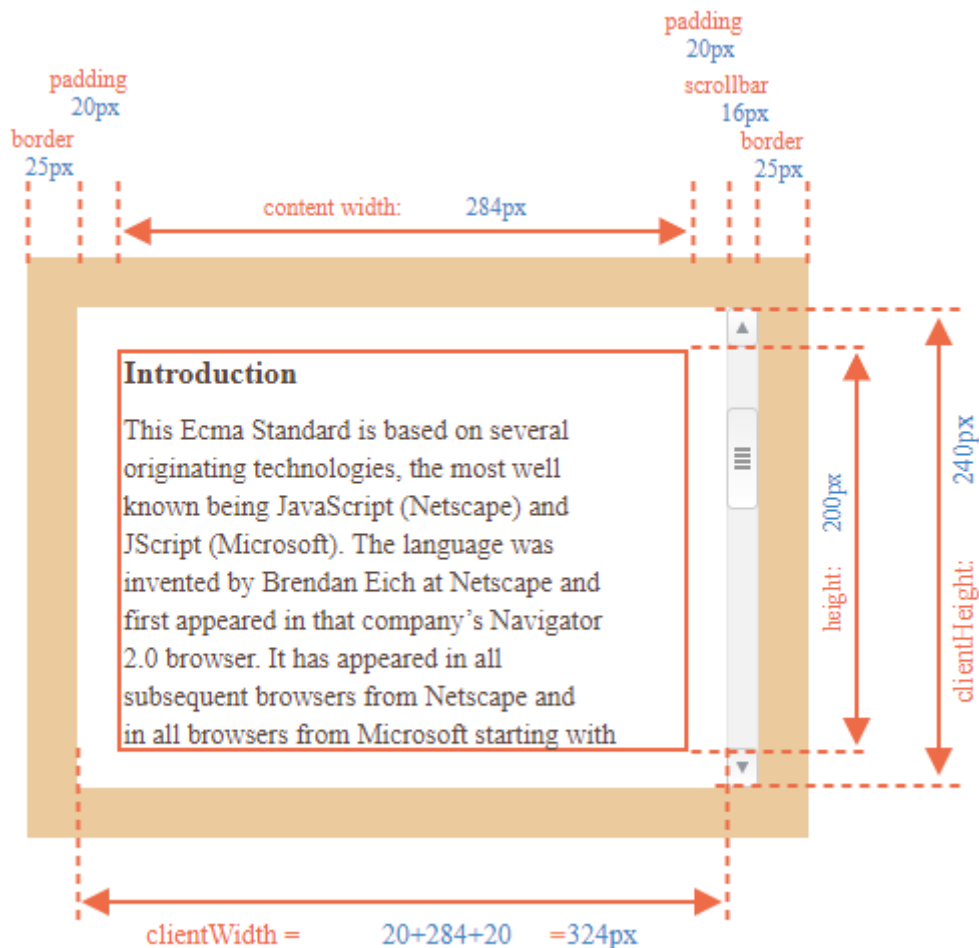
этом случае clientLeft будет равно 25, но с прокруткой –  $25 + 16 = 41$ .

Вот соответствующий пример на иврите:



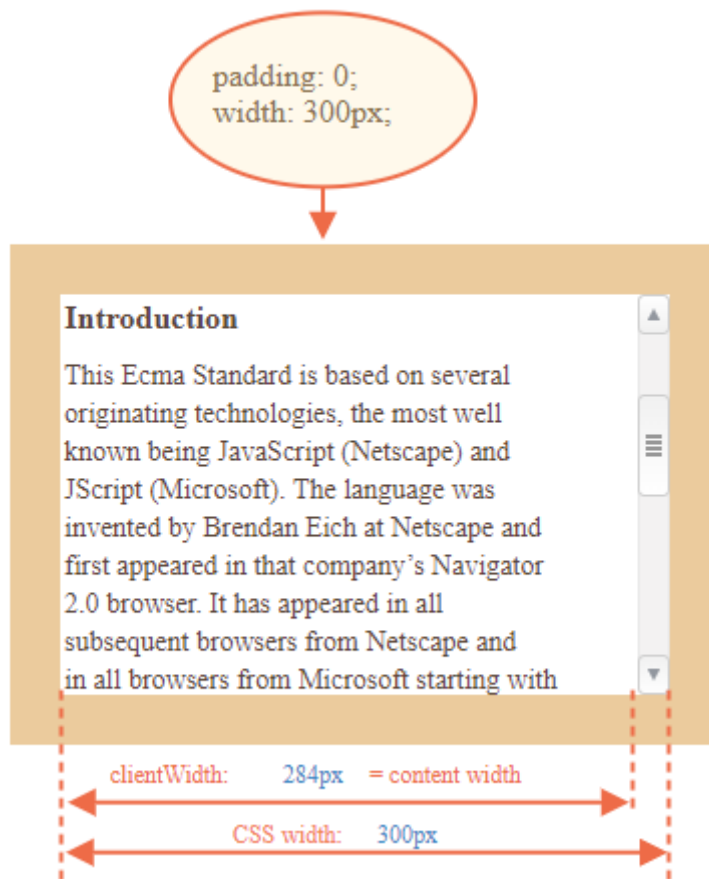
Свойства clientWidth/Height – это размер области внутри рамок элемента. Они включают в себя ширину области содержимого вместе с внутренними отступами padding, но без прокрутки:





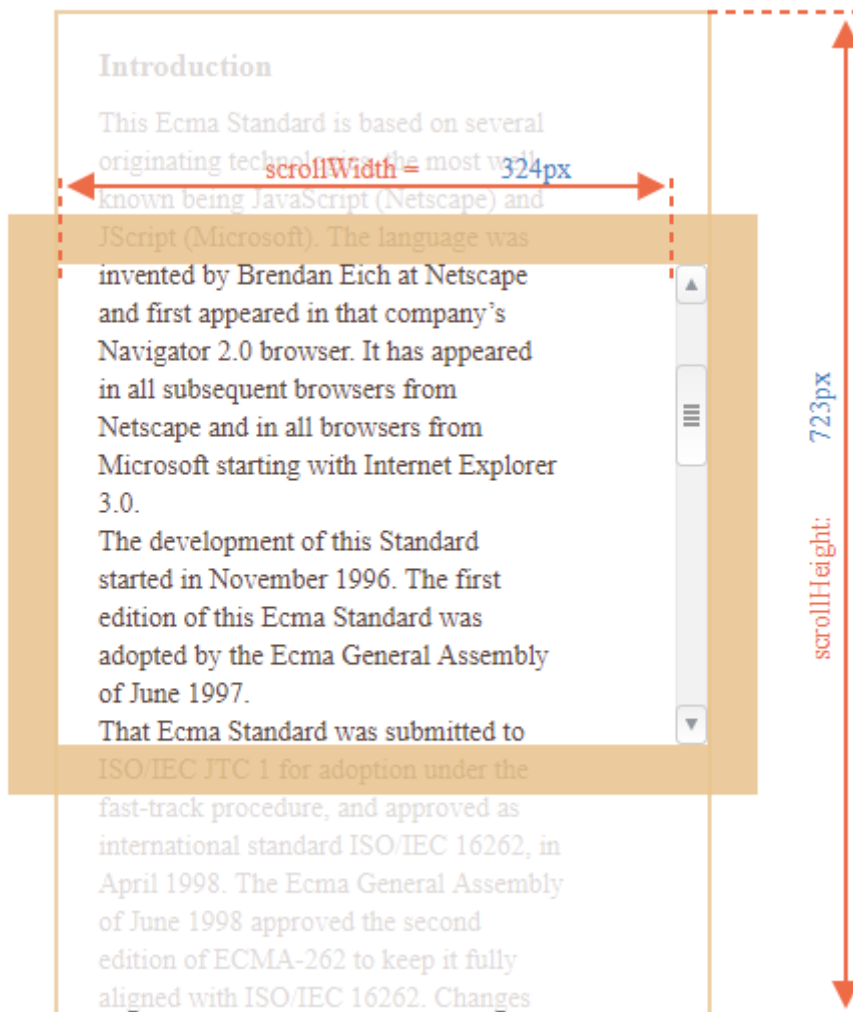
На рисунке выше горизонтальной прокрутки нет, так что высота `clientHeight` в точности то, что внутри рамок: CSS-высота 200px плюс верхние и нижние внутренние отступы ( $2 * 20\text{px}$ ), итого 240px. `clientWidth` – ширина содержимого здесь равна не 300px, а 284px, т.к. 16px отведено для полосы прокрутки. Таким образом: 284px плюс левый и правый отступы – всего 324px.

Если нет внутренних отступов `padding`, то `clientWidth/Height` в точности равны размеру области содержимого внутри рамок и полосы прокрутки (если она есть).



Поэтому в тех случаях, когда точно известно, что отступов нет, можно использовать `clientWidth/clientHeight` для получения размеров внутренней области содержимого.

Свойства `scrollWidth/Height` — как `clientWidth/clientHeight`, но также включают в себя прокрученную (которую не видно) часть элемента.



На рисунке выше:

- `scrollHeight` = 723 – полная внутренняя высота, включая прокрученную область;
- `scrollWidth` = 324 – полная внутренняя ширина, в данном случае прокрутки нет, поэтому она равна `clientWidth`.

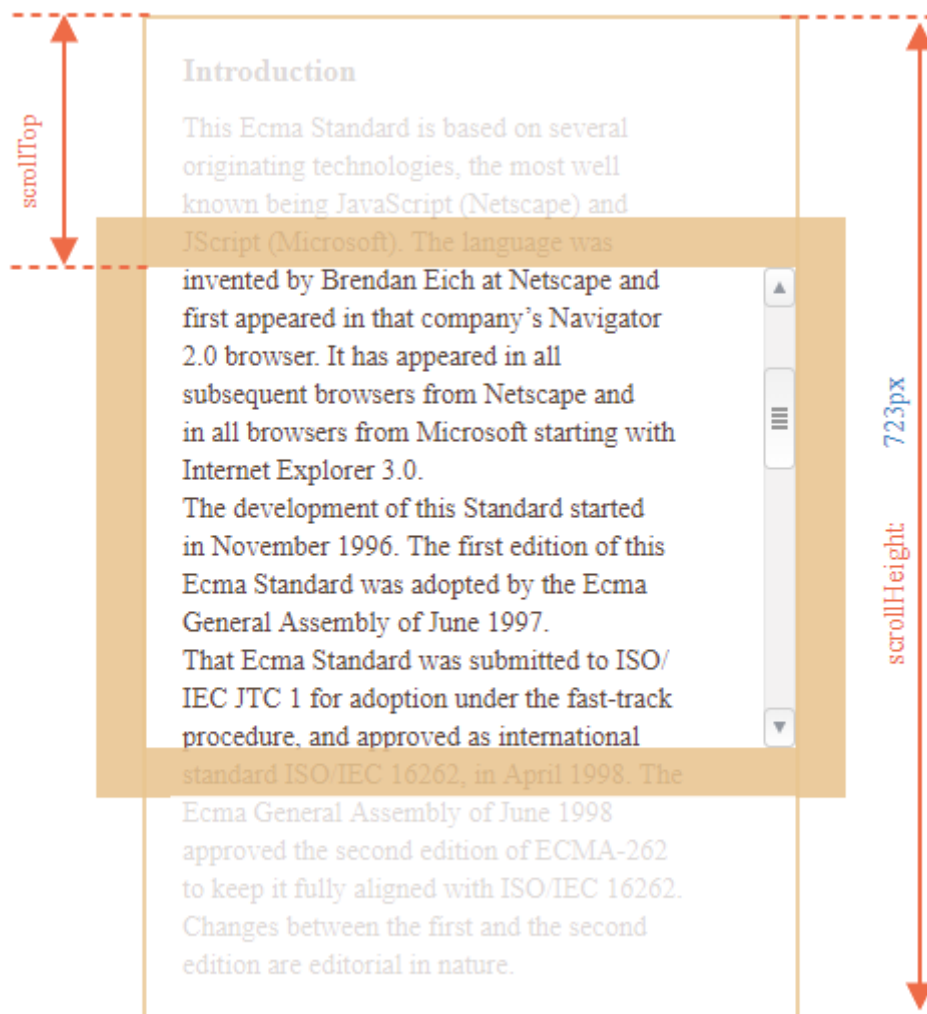
Эти свойства можно использовать, чтобы «распахнуть» элемент на всю ширину/высоту, например при нажатии на кнопку:

```
element.style.height = `${element.scrollHeight}px`;
```



Свойства `scrollLeft/scrollTop` – ширина/высота невидимой, прокрученной в данный момент, части элемента слева и сверху.

Следующая иллюстрация показывает значения `scrollHeight` и `scrollTop` для блока с вертикальной прокруткой.



Другими словами, свойство `scrollTop` – это «сколько уже прокручено вверх».

В отличие от большинства свойств, которые доступны только для чтения, значения `scrollLeft/scrollTop` можно изменять, и браузер выполнит прокрутку элемента. Установка значения `scrollTop` на 0 или `Infinity` прокрутит элемент в самый верх/низ соответственно.

Выше рассматривались метрики, которые есть у DOM-элементов, и которые можно использовать для получения различных высот, ширин и прочих расстояний. Но как известно, CSS-высоту и ширину можно извлечь, используя `getComputedStyle`:

```
let elem = document.body;

alert( getComputedStyle(elem).width );
```

Тем не менее стоит использовать свойства-метрики по следующим причинам:

1. Во-первых, CSS-свойства `width/height` зависят от другого свойства – `box-sizing`, которое определяет, «что такое», собственно, эти CSS-ширина и высота. Получается, что изменение `box-sizing`, к примеру, для более удобной вёрстки, ломает такой JavaScript.
2. Во-вторых, в CSS свойства `width/height` могут быть равны `auto`, например, для инлайнового элемента:

```
<span id="elem">Привет!</span>

<script>
  alert( getComputedStyle(elem).width ); // auto
</script>
```

Конечно, с точки зрения CSS `width:auto` – совершенно нормально, но в JavaScript нужен конкретный размер в `px`, который можно использовать для вычислений. Получается, что в данном случае ширина из CSS вообще бесполезна.

Есть и ещё одна причина: полоса прокрутки. Бывает, без полосы прокрутки код работает прекрасно, но стоит ей появиться, как начинают проявляться баги. Так происходит потому, что полоса прокрутки «забирает» место от области внутреннего содержимого в некоторых браузерах. Таким образом, реальная ширина содержимого меньше CSS-ширины. Как раз это и учитывают свойства `clientWidth/clientHeight`.

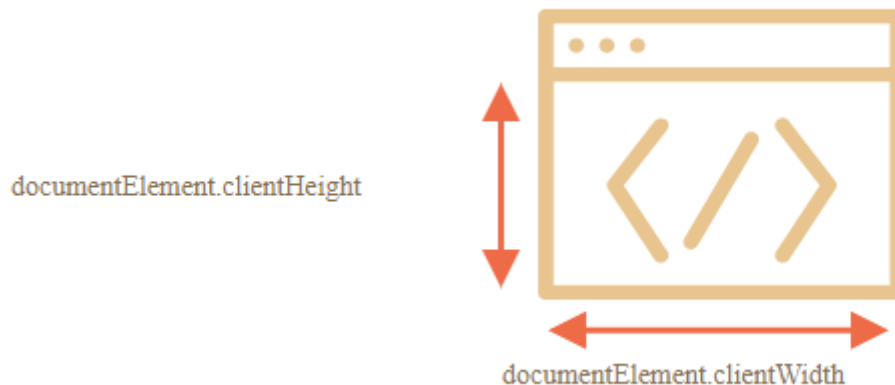
Но с `getComputedStyle(elem).width` ситуация иная. Некоторые браузеры (например, Chrome) возвращают реальную внутреннюю ширину с вычетом ширины полосы прокрутки, а некоторые (например, Firefox) – именно CSS-свойство (игнорируя полосу прокрутки). Эти кроссбраузерные отличия – ещё один повод не использовать `getComputedStyle`, а использовать свойства-метрики.

Описанные различия касаются только чтения свойства `getComputedStyle(...).width` из JavaScript, визуальное отображение корректно в обоих случаях.

## 18. Размеры и прокрутка окна. Координаты.

### Ширина/высота окна

Чтобы получить ширину/высоту окна, можно взять свойства `clientWidth/clientHeight` из `document.documentElement`:



Например, получить высоту окна можно так:  
`document.documentElement.clientHeight`.

Браузеры также поддерживают свойства `window.innerWidth/innerHeight`. Но их не стоит использовать, так как если есть полоса прокрутки, и она занимает какое-то место, то свойства `clientWidth/clientHeight` указывают на ширину/высоту документа без неё (за её вычетом). Иными словами, они возвращают высоту/ширину видимой части документа, доступной для содержимого. А `window.innerWidth/innerHeight` включают в себя полосу прокрутки. Если полоса прокрутки занимает некоторое место, то эти две строки выведут разные значения:

```
alert( window.innerWidth ); // полная ширина окна
alert( document.documentElement.clientWidth ); // ширина окна за
вычетом полосы прокрутки
```

В большинстве случаев нужна доступная ширина окна: для рисования или позиционирования. Полоса прокрутки «забирает» её часть. Поэтому следует использовать `documentElement.clientHeight/Width`.

Обратите внимание, что геометрические свойства верхнего уровня могут работать немного иначе, если в HTML нет `<!DOCTYPE HTML>`. Возможны странности. Поэтому в современном HTML всегда надо указывать DOCTYPE.

### Ширина/высота документа

Теоретически, т.к. корневым элементом документа является `documentElement.clientWidth/Height`, и он включает в себя всё содержимое, то можно получить полный размер документа как `documentElement.scrollWidth/scrollHeight`.

Но именно на этом элементе, для страницы в целом, эти свойства работают не так, как предполагается. В Chrome/Safari/Opera, если нет прокрутки, то `documentElement.scrollHeight` может быть даже меньше, чем `documentElement.clientHeight`. С точки зрения элемента это невозможная ситуация. Чтобы надёжно получить полную высоту документа, следует взять максимальное из этих свойств:

```
let scrollHeight = Math.max(
    document.body.scrollHeight, document.documentElement.scrollHeight,
    document.body.offsetHeight, document.documentElement.offsetHeight,
    document.body.clientHeight, document.documentElement.clientHeight
);

alert('Полная высота документа с прокручиваемой частью: ' +
scrollHeight);
```

Обычные элементы хранят текущее состояние прокрутки в `elem.scrollLeft/scrollTop`. Чтобы получить текущее состояние прокрутки страницы в большинстве браузеров можно обратиться к `documentElement.scrollLeft/Top`, за исключением основанных на старом WebKit (Safari), где есть ошибка (5991), и там нужно использовать `document.body` вместо `document.documentElement`.

Текущую прокрутку можно прочитать из свойств `window.pageXOffset/pageYOffset`:

```
alert('Текущая прокрутка сверху: ' + window.pageYOffset);
alert('Текущая прокрутка слева: ' + window.pageXOffset);
```

Эти свойства доступны только для чтения.

Для прокрутки страницы из JavaScript её DOM должен быть полностью построен. Например, если попытаться прокрутить страницу из скрипта в `<head>`, это не работает. Обычные элементы можно прокручивать, изменяя `scrollTop/scrollLeft`. Можно сделать то же самое для страницы в целом, используя `document.documentElement.scrollTop/Left` (кроме основанных на старом WebKit (Safari), где, как сказано выше, `document.body.scrollTop/Left`).

Есть и другие способы, в которых подобных несовместимостей нет: специальные методы `window.scrollBy(x,y)` и `window.scrollTo(pageX,pageY)`.

Метод `scrollBy(x,y)` прокручивает страницу относительно её текущего положения. Например, `scrollBy(0,10)` прокручивает страницу на 10px вниз.

Метод `scrollTo(pageX,pageY)` прокручивает страницу на абсолютные координаты (`pageX,pageY`). То есть, чтобы левый-верхний угол видимой части

страницы имел данные координаты относительно левого верхнего угла документа. Это всё равно, что поставить `scrollLeft/scrollTop`. Для прокрутки в самое начало можно использовать `scrollTo(0,0)`.

Эти методы одинаково работают для всех браузеров.

Рассмотрим ещё один метод: `elem.scrollToView(top)`. Вызов `elem.scrollToView(top)` прокручивает страницу, чтобы `elem` оказался вверху. У него есть один аргумент:

- если `top=true` (по умолчанию), то страница будет прокручена, чтобы `elem` появился в верхней части окна. Верхний край элемента совмещён с верхней частью окна.
- если `top=false`, то страница будет прокручена, чтобы `elem` появился внизу. Нижний край элемента будет совмещён с нижним краем окна.

Иногда нужно сделать документ непрокручиваемым. Например, при показе большого диалогового окна над документом – чтобы посетитель мог прокручивать это окно, но не документ. Чтобы запретить прокрутку страницы, достаточно установить `document.body.style.overflow = "hidden"`:

```
document.body.style.overflow = 'hidden'
```

Возобновить прокрутку:

```
document.body.style.overflow = ''
```

Аналогичным образом можно запретить прокрутку для других элементов, а не только для `document.body`.

Недостатком этого способа является то, что сама полоса прокрутки исчезает. Если она занимала некоторую ширину, то теперь эта ширина освободится, и содержимое страницы расширится, текст «прыгнет», заняв освободившееся место.

Это выглядит немного странно, но это можно обойти, если сравнить `clientWidth` до и после остановки, и если `clientWidth` увеличится (значит полоса прокрутки исчезла), то добавить `padding` в `document.body` вместо полосы прокрутки, чтобы оставить ширину содержимого прежней.

Большинство соответствующих методов JavaScript работают в одной из двух указанных ниже систем координат:

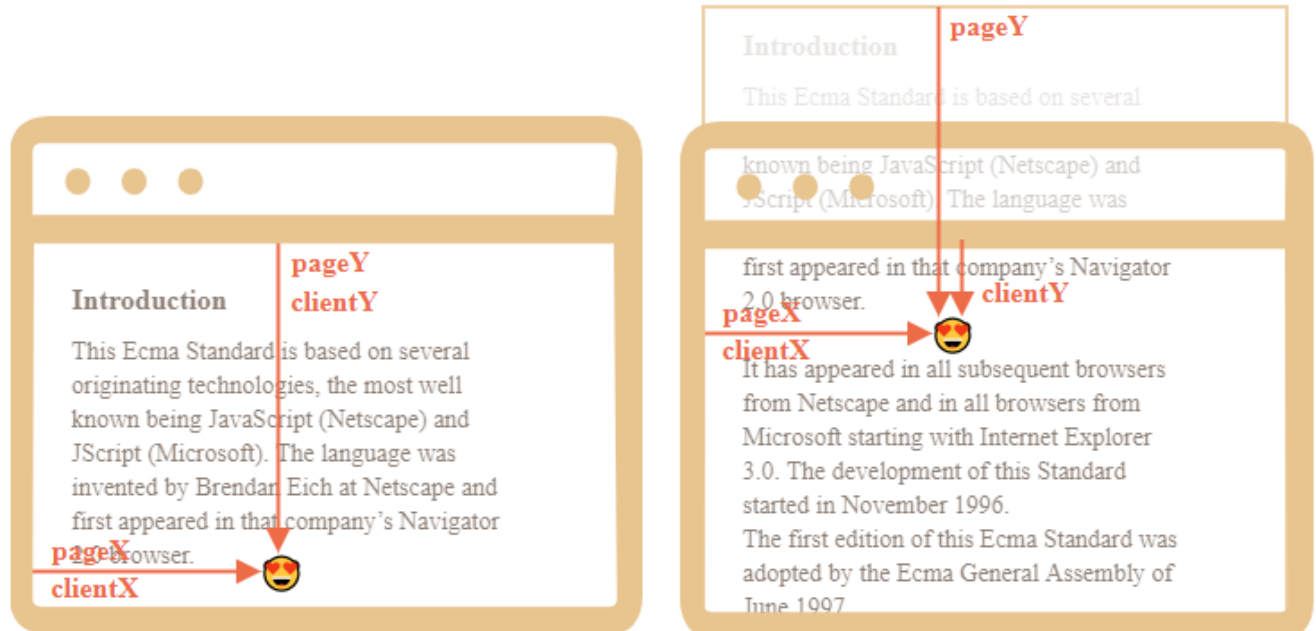
1. Относительно окна браузера – как `position:fixed`, отсчёт идёт от верхнего левого угла окна. Далее будем обозначать эти координаты как `clientX/clientY`.
2. Относительно документа – как `position:absolute` на уровне документа, отсчёт идёт от верхнего левого угла документа. Далее будем обозначать эти координаты как `pageX/pageY`.

Когда страница полностью прокручена в самое начало, то верхний левый угол окна совпадает с левым верхним углом документа, при этом обе этих системы координат тоже совпадают. Но если происходит прокрутка, то



координаты элементов в контексте окна меняются, так как они двигаются, но в то же время их координаты относительно документа остаются такими же.

На картинке ниже показаны координат точки до прокрутки (слева) и после (справа):



При прокрутке документа:

- pageY – координата точки относительно документа осталась без изменений, так как отсчёт по-прежнему ведётся от верхней границы документа (сейчас она прокручена наверх).
- clientY – координата точки относительно окна изменилась (стрелка на рисунке стала короче), так как точка стала ближе к верхней границе окна.

### Координаты относительно окна: `getBoundingClientRect`

Метод `elem.getBoundingClientRect()` возвращает координаты в контексте окна для минимального по размеру прямоугольника, который включает в себе элемент `elem`, в виде объекта встроенного класса `DOMRect`.

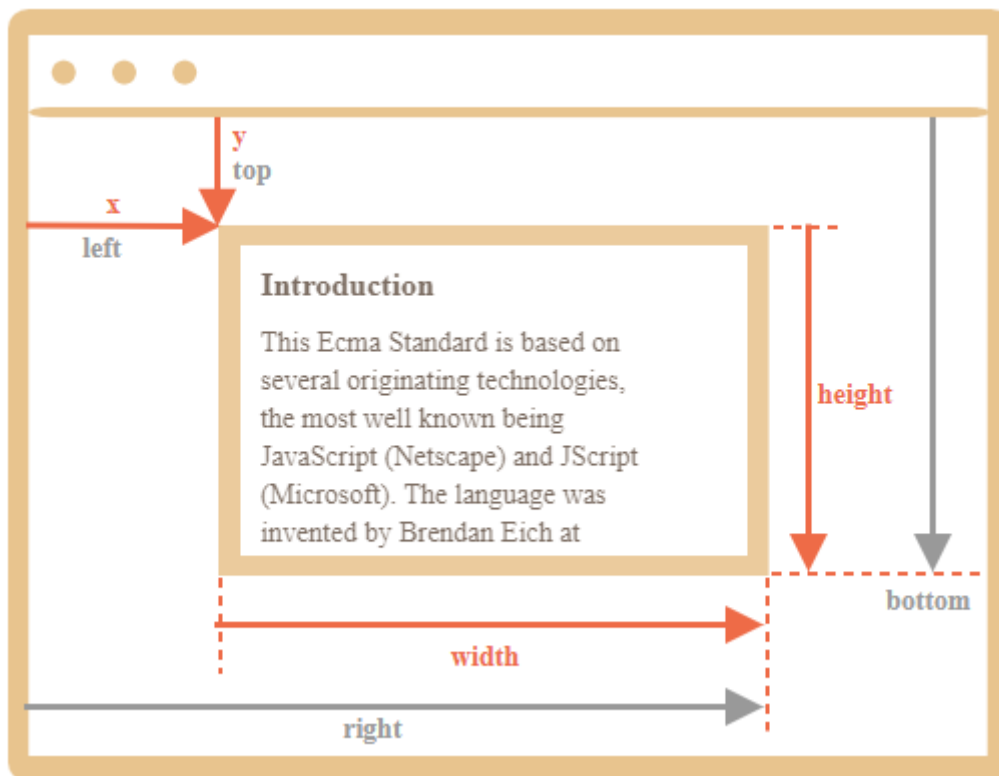
Основные свойства объекта типа `DOMRect`:

- x/y – X/Y-координаты начала прямоугольника относительно окна,
- width/height – ширина/высота прямоугольника (могут быть отрицательными).

Дополнительные, «зависимые», свойства:

- top/bottom – Y-координата верхней/нижней границы прямоугольника,
- left/right – X-координата левой/правой границы прямоугольника.

Картинка с результатами вызова `elem.getBoundingClientRect()`:



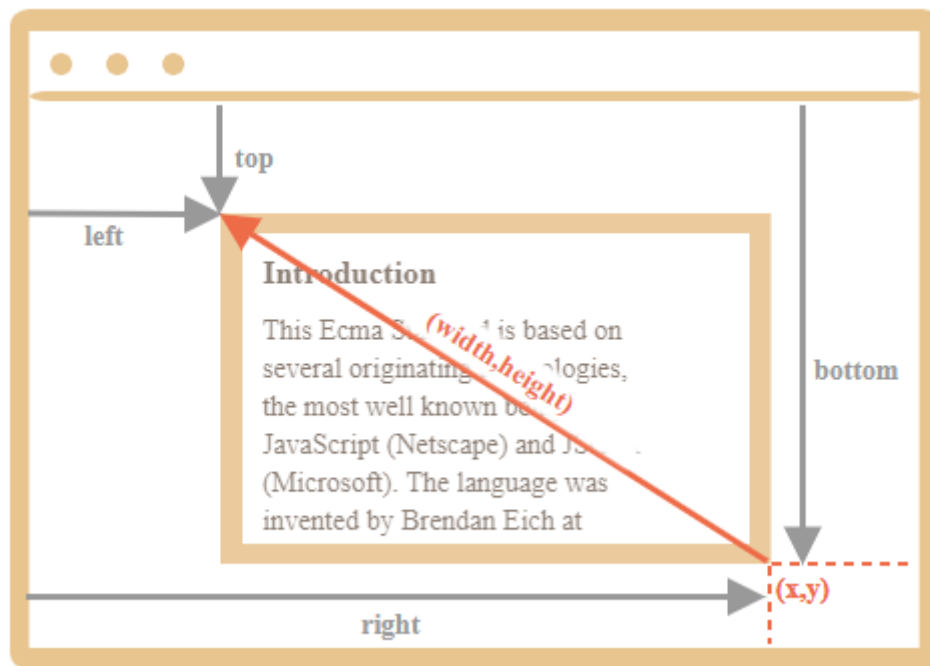
Как видно, `x/y` и `width/height` уже точно задают прямоугольник. Остальные свойства могут быть легко вычислены на их основе: `left = x`, `top = y`, `right = x + width`, `bottom = y + height`.

Заметим:

- Координаты могут считаться с десятичной частью, например, 10.5. Это нормально, ведь браузер использует дроби в своих внутренних вычислениях.
- Координаты могут быть отрицательными. Например, если страница прокручена так, что элемент `elem` ушёл вверх за пределы окна, то вызов `elem.getBoundingClientRect().top` вернёт отрицательное значение.

С математической точки зрения, прямоугольник однозначно задаётся начальной точкой (`x,y`) и вектором направления (`width, height`). Так что дополнительные зависимые свойства существуют лишь для удобства. Технически, значения `width/height` могут быть отрицательными, это позволяет задать «направленный» прямоугольник, например, для выделения мышью с отмеченным началом и концом.

Пример с  
прямоугольника  
отрицательными `width` и `height` (например, `width=-200`, `height=-100`):



Он начинается в правом-нижнем углу, и затем «растёт» влево-вверх, так как отрицательные width/height ведут его назад по координатам. Как видно, свойства left/top здесь не равны x/y. То есть они не дублируют друг друга. Формулы выше могут быть исправлены с учётом возможных отрицательных значений width/height. Это достаточно просто сделать, но редко требуется, так как результат вызова `elem.getBoundingClientRect()` всегда возвращает положительные значения для ширины/высоты.

Internet Explorer и Edge не поддерживают свойства x/y. Таким образом, можно либо сделать полифил (добавив соответствующие геттеры в `DomRect.prototype`), либо использовать top/left, так как это всегда одно и то же при положительных width/height, в частности — в результате вызова `elem.getBoundingClientRect()`.

Есть очевидное сходство между координатами относительно окна и CSS `position:fixed`. Но в CSS свойство `right` означает расстояние от правого края, и свойство `bottom` означает расстояние от нижнего края окна браузера. Если взглянуть на картинку выше, то видно, что в JavaScript это не так. Все координаты в контексте окна считаются от верхнего левого угла, включая `right/bottom`.

Вызов `document.elementFromPoint(x, y)` возвращает самый глубоко вложенный элемент в окне, находящийся по координатам (x, y). Синтаксис:

```
let elem = document.elementFromPoint(x, y);
```

Например, код ниже выделяет с помощью стилей и выводит имя тега элемента, который сейчас в центре окна браузера:

```
let centerX = document.documentElement.clientWidth / 2;  
let centerY = document.documentElement.clientHeight / 2;
```

```
let elem = document.elementFromPoint(centerX, centerY);

elem.style.background = "red";
alert(elem.tagName);
```

Поскольку используются координаты в контексте окна, то элемент может быть разным, в зависимости от того, какая сейчас прокрутка.

Метод `document.elementFromPoint(x,y)` работает, только если координаты (x,y) относятся к видимой части содержимого окна. Если любая из координат представляет собой отрицательное число или превышает размеры окна, то возвращается `null`. Вот типичная ошибка, которая может произойти, если в коде нет соответствующей проверки:

```
let elem = document.elementFromPoint(x, y);
elem.style.background = ''; // ошибка
```

Чаще всего нужны координаты для позиционирования чего-либо. Чтобы показать что-то около нужного элемента, можно вызвать `getBoundingClientRect`, чтобы получить его координаты элемента, а затем использовать CSS-свойство `position` вместе с `left/top` (или `right/bottom`). Например, функция `createMessageUnder(elem, html)` ниже показывает сообщение под элементом `elem`:

```
let elem = document.getElementById("coords-show-mark");

function createMessageUnder(elem, html) {
  let message = document.createElement('div');
  message.style.cssText = "position:fixed; color: red";

  let coords = elem.getBoundingClientRect();

  message.style.left = coords.left + "px";
  message.style.top = coords.bottom + "px";

  message.innerHTML = html;

  return message;
}

let message = createMessageUnder(elem, 'Hello, world!');
document.body.append(message);
setTimeout(() => message.remove(), 5000);
```

При прокрутке страницы сообщение будет уплывать от кнопки. Причина в том, что сообщение позиционируется с помощью `position:fixed`, поэтому оно остаётся всегда на том же самом месте в окне при прокрутке

страницы. Чтобы изменить это, нужно использовать другую систему координат, где сообщение позиционировалось бы относительно документа, и свойство `position: absolute`.

### Координаты относительно документа

В такой системе координат отсчёт ведётся от левого верхнего угла документа, не окна. В CSS координаты относительно окна браузера соответствуют свойству `position: fixed`, а координаты относительно документа – свойству `position: absolute` на самом верхнем уровне вложенности.

Можно воспользоваться свойствами `position: absolute` и `top/left`, чтобы привязать что-нибудь к конкретному месту в документе. При этом прокрутка страницы не имеет значения. Но сначала нужно получить верные координаты.

Не существует стандартного метода, который возвращал бы координаты элемента относительно документа, но можно написать его сами.

Две системы координат связаны следующими формулами:

- $\text{pageY} = \text{clientY} + \text{высота вертикально прокрученной части документа}$ ,
- $\text{pageX} = \text{clientX} + \text{ширина горизонтально прокрученной части документа}$ .

Функция `getCoords(elem)` берёт координаты в контексте окна с помощью `elem.getBoundingClientRect()` и добавляет к ним значение соответствующей прокрутки:

```
function getCoords(elem) {  
  let box = elem.getBoundingClientRect();  
  
  return {  
    top: box.top + pageYOffset,  
    left: box.left + pageXOffset  
  };  
}
```

Если бы в примере выше функция использовалась вместе с `position: absolute`, то при прокрутке сообщение оставалось бы рядом с элементом. Модифицированная функция `createMessageUnder`:

```
function createMessageUnder(elem, html) {  
  let message = document.createElement('div');  
  message.style.cssText = "position: absolute; color: red";  
  let coords = getCoords(elem);  
  message.style.left = coords.left + "px";  
  message.style.top = coords.bottom + "px";  
  message.innerHTML = html;  
  
  return message;  
}
```

## 19. Браузерные события.

*Событие* – это сигнал от браузера о том, что что-то произошло. Все DOM-узлы подают такие сигналы (хотя события бывают и не только в DOM). Ниже перечисляются наиболее часто используемые DOM-событий.

События мыши:

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

События на элементах управления:

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

Клавиатурные события:

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

События документа:

- DOMContentLoaded – когда HTML загружен и обработан, DOM документа полностью построен и доступен.

CSS events:

- transitionend – когда CSS-анимация завершена.

Существует множество других событий.

### Обработчики событий

Событию можно назначить обработчик, то есть функцию, которая сработает, как только событие произошло. Именно благодаря обработчикам JavaScript-код может реагировать на действия пользователя. Есть несколько способов назначить событию обработчик.

Обработчик может быть назначен прямо в разметке, в атрибуте, который называется on<событие>. Например, чтобы назначить обработчик события click на элементе input, можно использовать атрибут onclick, вот так:

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

При клике мышкой на кнопке выполнится код, указанный в атрибуте onclick. Обратите внимание, для содержимого атрибута onclick используются одинарные кавычки, так как сам атрибут находится в двойных. Если поставить двойные кавычки внутри атрибута, вот так: onclick="alert("Click!")", код не будет работать.

Атрибут HTML-тега – не самое удобное место для написания большого количества кода, поэтому лучше создать отдельную JavaScript-функцию и

вызвать её там. Следующий пример по клику запускает функцию countRabbits():

```
<script>
  function countRabbits() {
    for(let i=1; i<=3; i++) {
      alert("Кролик номер " + i);
    }
  }
</script>

<input type="button" onclick="countRabbits()" value="Считать кроликов!">
```

Как известно, атрибут HTML-тега не чувствителен к регистру, поэтому ONCLICK будет работать так же, как onClick и onCLICK. Но, как правило, атрибуты пишут в нижнем регистре: onclick.

Можно назначать обработчик, используя свойство DOM-элемента on<событие>. К примеру, elem.onclick:

```
<input id="elem" type="button" value="Нажми меня!">
<script>
  elem.onclick = function() {
    alert('Спасибо');
  };
</script>
```

Если обработчик задан через атрибут, то браузер читает HTML-разметку, создаёт новую функцию из содержимого атрибута и записывает в свойство. Этот способ, по сути, аналогичен предыдущему. Обработчик всегда хранится в свойстве DOM-объекта, а атрибут – лишь один из способов его инициализации.

Эти два примера кода работают одинаково:

1. Только HTML:

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

2. HTML + JS:

```
<input type="button" id="button" value="Кнопка">
<script>
  button.onclick = function() {
    alert('Клик!');
  };
</script>
```

Так как у элемента DOM может быть только одно свойство с именем `onclick`, то назначить более одного обработчика так нельзя. В примере ниже назначение через JavaScript перезапишет обработчик из атрибута:

```
<input type="button" id="elem" onclick="alert('Было')" value="Нажми
меня">
<script>
  elem.onclick = function() {
    alert('Станет');
  };
</script>
```

Кстати, обработчиком можно назначить и уже существующую функцию:

```
function sayThanks() {
  alert('Спасибо!');
}

elem.onclick = sayThanks;
```

Убрать обработчик можно назначением `elem.onclick = null`.

Внутри обработчика события `this` ссылается на текущий элемент, то есть на тот, на котором, как говорят, «висит» (т.е. назначен) обработчик. В коде ниже `button` выводит своё содержимое, используя `this.innerHTML`:

```
<button onclick="alert(this.innerHTML)">Нажми меня</button>
```

Обратите внимание, что функция должна быть присвоена как `sayThanks`, а не `sayThanks()`.

```
// правильно
button.onclick = sayThanks;

// неправильно
button.onclick = sayThanks();
```

Если добавить скобки, то `sayThanks()` – это уже вызов функции, результат которого (равный `undefined`, так как функция ничего не возвращает) будет присвоен `onclick`. Так что это не будет работать. А вот в разметке, в отличие от свойства, скобки нужны:

```
<input type="button" id="button" onclick="sayThanks()">
```



Это различие просто объяснить. При создании обработчика браузером из атрибута, он автоматически создаёт функцию с телом из значения атрибута: `sayThanks()`. Так что разметка генерирует такое свойство:

```
button.onclick = function() {  
    sayThanks();  
};
```

Назначение обработчика строкой `elem.onclick = "alert(1)"` также сработает. Это сделано из соображений совместимости, но делать так не рекомендуется.

Не используйте `setAttribute` для обработчиков. Такой вызов работать не будет, так как атрибуты всегда строки, и функция станет строкой:

```
document.body.setAttribute('onclick', function() { alert(1) });
```

Используйте `elem.onclick`, а не `elem.ONCLICK`, потому что DOM-свойства чувствительны к регистру.

Фундаментальный недостаток описанных выше способов назначения обработчика – невозможность повесить несколько обработчиков на одно событие. Например, одна часть кода хочет при клике на кнопку делать её подсвеченной, а другая – выдавать сообщение, следовательно, надо назначить два обработчика для этого. Но новое DOM-свойство перезапишет предыдущее:

```
input.onclick = function() { alert(1); }  
// ...  
input.onclick = function() { alert(2); }
```

Решить эту проблему позволяют методы `addEventListener` и `removeEventListener`.

Синтаксис добавления обработчика:

```
element.addEventListener(event, handler[, options]);
```

- `event` – имя события, например `"click"`;
- `handler` – ссылка на функцию-обработчик;
- `options` – дополнительный объект со свойствами:
  - `once`: если `true`, тогда обработчик будет автоматически удалён после выполнения;
  - `capture`: фаза, на которой должен сработать обработчик (подробнее об этом будет рассказано в вопросе о всплытии и погружении событий). `options` может быть `false/true`, это тоже самое, что `{capture: false/true}`.

- `passive`: если `true`, то указывает, что обработчик никогда не вызовет `preventDefault()` (подробнее это будет рассматриваться в вопросе о действии браузера по умолчанию).

Для удаления обработчика следует использовать `removeEventListener`:

```
element.removeEventListener(event, handler[, options]);
```

Для удаления обработчика нужно передать именно ту функцию-обработчик которая была назначена. Вот так не работает:

```
elem.addEventListener( "click" , () => alert('Спасибо!'));  
elem.removeEventListener( "click", () => alert('Спасибо!'));
```

Обработчик не будет удалён, т.к. в `removeEventListener` передана не та же функция, а другая, с одинаковым кодом. Вот так правильно:

```
function handler() {  
    alert( 'Спасибо!' );  
}  
  
input.addEventListener("click", handler);  
  
input.removeEventListener("click", handler);
```

Обратим внимание – если функцию обработчик не сохранить где-либо, нельзя будет её удалить. Нет метода, который позволяет получить из элемента обработчики событий, назначенные через `addEventListener`. Метод `addEventListener` позволяет добавлять несколько обработчиков на одно событие одного элемента, например:

```
<input id="elem" type="button" value="Нажми меня"/>  
  
<script>  
    function handler1() {  
        alert('Спасибо!');  
    };  
  
    function handler2() {  
        alert('Спасибо ещё раз!');  
    }  
  
    elem.onclick = () => alert("Привет");  
    elem.addEventListener("click", handler1); // Спасибо!  
    elem.addEventListener("click", handler2); // Спасибо ещё раз!  
</script>
```

Как видно из примера выше, можно одновременно назначать обработчики и через DOM-свойство и через `addEventListener`. Однако, во избежание путаницы, рекомендуется выбрать один способ.

Существуют события, которые нельзя назначить через DOM-свойство, но можно через `addEventListener`. Например, таково событие `transitionend`, то есть окончание CSS-анимации. Код ниже демонстрирует это. В большинстве браузеров, сработает лишь второй обработчик, но не первый:

```
<style>
  input {
    transition: width 1s;
    width: 100px;
  }

  .wide {
    width: 300px;
  }
</style>

<input type="button" id="elem" onclick="this.classList.toggle('wide')"
value="Нажми меня">

<script>
  elem.ontransitionend = function() {
    alert("DOM property"); // не сработает
  };

  elem.addEventListener("transitionend", function() {
    alert("addEventListener"); // сработает по окончании анимации
  });
</script>
```

### Объект события

Чтобы хорошо обработать событие, могут понадобиться детали того, что произошло. Не просто «клик» или «нажатие клавиши», а также — какие координаты указателя мыши, какая клавиша нажата и так далее. Когда происходит событие, браузер создаёт объект события, записывает в него детали и передаёт его в качестве аргумента функции-обработчику. Пример ниже демонстрирует получение координат мыши из объекта события:

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

<script>
  elem.onclick = function(event) {
    // вывести тип события, элемент и координаты клика
    alert(event.type + " на " + event.currentTarget);
    alert("Координаты: " + event.clientX + ":" + event.clientY);
  };
</script>
```

Некоторые свойства объекта event:

- event.type – тип события, в данном случае "click";
- event.currentTarget – элемент, на котором сработал обработчик. Значение – обычно такое же, как и у this, но если обработчик является функцией-стрелкой или при помощи bind привязан другой объект в качестве this, то можно получить элемент из event.currentTarget;
- event.clientX / event.clientY – координаты курсора в момент клика относительно окна, для событий мыши.

Есть также и ряд других свойств, в зависимости от типа событий, которые будут рассмотрены далее.

При назначении обработчика в HTML, тоже можно использовать объект event, вот так:

```
<input type="button" onclick="alert(event.type)" value="Тип события">
```

Это возможно потому, что когда браузер из атрибута создаёт функцию-обработчик, то она выглядит так: function(event) { alert(event.type) }. То есть, её первый аргумент называется "event", а тело взято из атрибута.

Можно назначить обработчиком не только функцию, но и объект при помощи addEventListener. В этом случае, когда происходит событие, вызывается метод объекта handleEvent. К примеру:

```
<button id="elem">Нажми меня</button>

<script>
  elem.addEventListener('click', {
    handleEvent(event) {
      alert(event.type + " на " + event.currentTarget);
    }
  });
</script>
```

Как видим, если addEventListener получает объект в качестве обработчика, он вызывает object.handleEvent(event), когда происходит событие. Также можно использовать класс для этого:

```
<button id="elem">Нажми меня</button>

<script>
  class Menu {
    handleEvent(event) {
      switch(event.type) {
        case 'mousedown':
          elem.innerHTML = "Нажата кнопка мыши";
          break;
        case 'mouseup':
```

```

        elem.innerHTML += "...и отжата.";
        break;
    }
}
}

let menu = new Menu();
elem.addEventListener('mousedown', menu);
elem.addEventListener('mouseup', menu);
</script>

```

Здесь один и тот же объект обрабатывает оба события. Обратите внимание, нужно явно назначить оба обработчика через `addEventListener`. Тогда объект `menu` будет получать события `mousedown` и `mouseup`, но не другие (не назначенные) типы событий.

Метод `handleEvent` не обязательно должен выполнять всю работу сам. Он может вызывать другие методы, которые заточены под обработку конкретных типов событий, вот так:

```

<button id="elem">Нажми меня</button>

<script>
    class Menu {
        handleEvent(event) {
            // mousedown -> onMousedown
            let method = 'on' + event.type[0].toUpperCase() +
event.type.slice(1);
            this[method](event);
        }

        onMousedown() {
            elem.innerHTML = "Кнопка мыши нажата";
        }

        onMouseup() {
            elem.innerHTML += "...и отжата.";
        }
    }

    let menu = new Menu();
    elem.addEventListener('mousedown', menu);
    elem.addEventListener('mouseup', menu);
</script>

```

Теперь обработка событий разделена по методам, что упрощает поддержку кода.

## 20. Всплытие и погружение событий.

Рассмотрим пример. Этот обработчик для `<div>` сработает, если вы кликните по любому из вложенных тегов, будь то `<em>` или `<code>`:

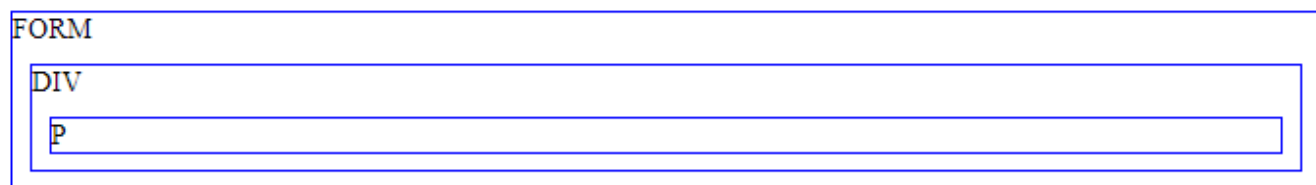
```
<div onclick="alert('Обработчик!')">
  <em>Если вы кликните на <code>EM</code>, сработает обработчик на
  <code>DIV</code></em>
</div>
```

### Всплытие

Принцип всплытия очень простой. Когда на элементе происходит событие, обработчики сначала срабатывают на нём, потом на его родителе, затем выше и так далее, вверх по цепочке предков. Например, есть 3 вложенных элемента `FORM > DIV > P` с обработчиком на каждом:

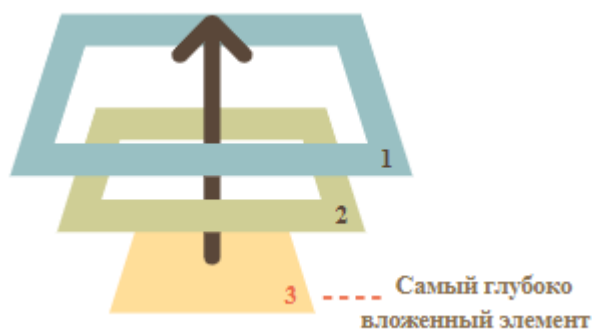
```
<style>
  body * {
    margin: 10px;
    border: 1px solid blue;
  }
</style>

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



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

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



Поэтому если кликнуть на `<p>`, то появятся три оповещения: `p → div → form`. Этот процесс называется «всплытием», потому что события «всплывают» от внутреннего элемента вверх через родителей подобно тому, как всплывает пузырёк воздуха в воде. Не все события всплывают. Например, событие `focus` не всплывает. Однако, это исключение, всё-таки большинство событий всплывают.

### **event.target**

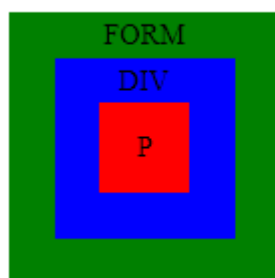
Всегда можно узнать, на каком конкретно элементе произошло событие. Самый глубокий элемент, который вызывает событие, называется целевым элементом, и он доступен через `event.target`. Отличия от `this` (`=event.currentTarget`):

- `event.target` – это «целевой» элемент, на котором произошло событие, в процессе всплытия он неизменен.
- `this` – это «текущий» элемент, до которого дошло всплытие, на нём сейчас выполняется обработчик.

Например, если стоит только один обработчик `form.onclick`, то он «поймает» все клики внутри формы. Где бы ни был клик внутри – он всплывёт до элемента `<form>`, на котором сработает обработчик. При этом внутри обработчика `form.onclick`:

- `this` (`=event.currentTarget`) всегда будет элемент `<form>`, так как обработчик сработал на ней.
- `event.target` будет содержать ссылку на конкретный элемент внутри формы, на котором произошёл клик.

Клик покажет оба: и `event.target`, и `this` для сравнения:



```
// script.js

form.onclick = function(event) {
    event.target.style.backgroundColor = 'yellow';

    // браузеру нужно некоторое время, чтобы зарисовать всё жёлтым
    setTimeout(() => {
        alert("target = " + event.target.tagName + ", this=" +
this.tagName);
        event.target.style.backgroundColor = ''
    }, 0);
};
```

```
// example.css
```

```
form {
    background-color: green;
    position: relative;
    width: 150px;
    height: 150px;
    text-align: center;
    cursor: pointer;
}
```

```
div {
    background-color: blue;
    position: absolute;
    top: 25px;
    left: 25px;
    width: 100px;
    height: 100px;
}
```

```
p {
    background-color: red;
    position: absolute;
    top: 25px;
    left: 25px;
    width: 50px;
    height: 50px;
    line-height: 50px;
    margin: 0;
}
```

```
body {
    line-height: 25px;
    font-size: 16px;
}
```

```
// index.html
```

```
<!DOCTYPE HTML>
```



```

<html>
<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="example.css">
</head>

<body>
  Клик покажет оба: и event.target, и this
  для сравнения:

  <form id="form">FORM
    <div>DIV
      <p>P</p>
    </div>
  </form>

  <script src="script.js"></script>
</body>
</html>

```

Возможна и ситуация, когда `event.target` и `this` – один и тот же элемент, например, если клик был непосредственно на самом элементе `<form>`, не на его подэлементе.

### Прекращение всплытия

Всплытие идёт с «целевого» элемента прямо вверх. По умолчанию событие будет всплывать до элемента `<html>`, а затем до объекта `document`, а иногда даже до `window`, вызывая все обработчики на своём пути. Но любой промежуточный обработчик может решить, что событие полностью обработано, и остановить всплытие. Для этого нужно вызвать метод `event.stopPropagation()`. Например, здесь при клике на кнопку `<button>` обработчик `body.onclick` не сработает:

```

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

```

Если у элемента есть несколько обработчиков на одно событие, то даже при прекращении всплытия все они будут выполнены. То есть, `event.stopPropagation()` препятствует продвижению события дальше, но на текущем элементе все обработчики будут вызваны.

Для того, чтобы полностью остановить обработку, существует метод `event.stopImmediatePropagation()`. Он не только предотвращает всплытие, но и останавливает обработку событий на текущем элементе.

Не прекращайте всплытие без необходимости. Зачастую прекращение всплытия через `event.stopPropagation()` имеет свои подводные камни, которые со временем могут стать проблемами. Например:

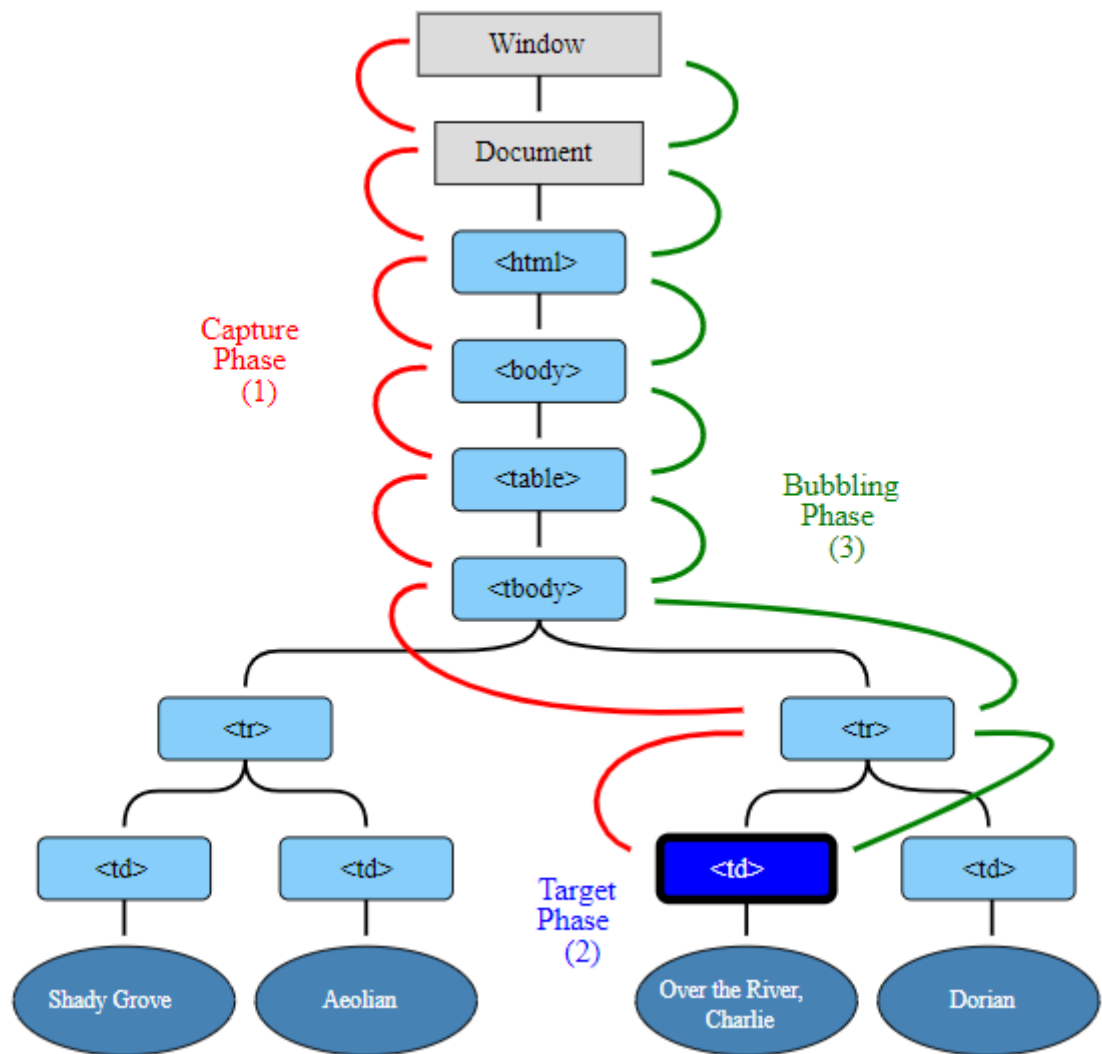
1. При создании вложенного меню, каждое подменю обрабатывает клики на своих элементах и делает для них `stopPropagation`, чтобы не срабатывало внешнее меню.
2. Позже было решено отслеживать все клики в окне для какой-то своей функциональности, к примеру, для статистики – где вообще кликают пользователи. Обычно используют `document.addEventListener('click'...)`, чтобы отлавливать все клики.
3. Аналитика не будет работать над областью, где клики прекращаются `stopPropagation`, получилась «мёртвая зона».

### **Погружение**

Существует ещё одна фаза из жизненного цикла события – «погружение» (иногда её называют «перехват»). Она очень редко используется в реальном коде, однако тоже может быть полезной. Стандарт DOM Events описывает 3 фазы прохода события:

1. Фаза погружения (`capturing phase`) – событие сначала идёт сверху вниз.
2. Фаза цели (`target phase`) – событие достигло целевого(исходного) элемента.
3. Фаза всплытия (`bubbling stage`) – событие начинает всплывать.

Картинка из спецификации демонстрирует, как это работает при клике по ячейке `<td>`, расположенной внутри таблицы:



То есть при клике на `<td>` событие путешествует по цепочке родителей сначала вниз к элементу (погружается), затем оно достигает целевой элемент (фаза цели), а потом идёт вверх (всплытие), вызывая по пути обработчики.

Обработчики, добавленные через `on<event>`-свойство или через HTML-атрибуты, или через `addEventListener(event, handler)` с двумя аргументами, ничего не знают о фазе погружения, а работают только на 2-ой и 3-ей фазах. Чтобы поймать событие на стадии погружения, нужно использовать третий аргумент `capture` вот так:

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

Существуют два варианта значений опции `capture`:

- Если аргумент `false` (по умолчанию), то событие будет поймано при всплытии.
- Если аргумент `true`, то событие будет перехвачено при погружении.

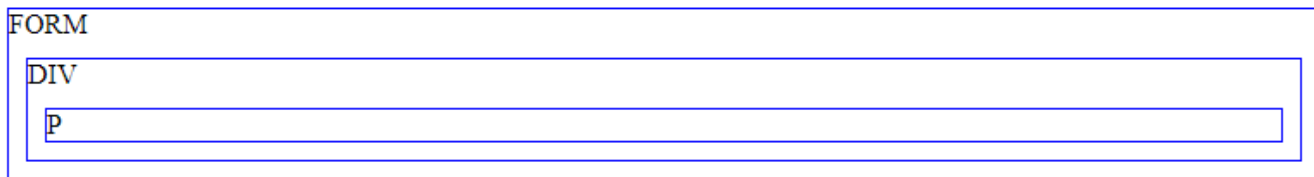
Обратите внимание, что хоть и формально существует 3 фазы, 2-ую фазу («фазу цели»: событие достигло элемента) нельзя обработать отдельно, при её

достижении вызываются все обработчики: и на всплытие, и на погружение. Посмотрим и всплытие и погружение в действии:

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

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

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



Здесь обработчики назначаются каждому элементу в документе, чтобы увидеть в каком порядке они вызываются по мере прохода события. Если кликнуть по `<p>`, то последовательность следующая:

1. HTML → BODY → FORM → DIV (фаза погружения, первый обработчик).
2. P (фаза цели, срабатывают обработчики, установленные и на погружение, и на всплытие, так что выведется два раза).
3. DIV → FORM → BODY → HTML (фаза всплытия, второй обработчик).

Существует свойство `event.eventPhase`, содержащее номер фазы, на которой событие было поймано. Но оно используется редко, обычно это и так известно в обработчике.

Если обработчик добавлен с помощью `addEventListener(..., true)`, то надо передать то же значение аргумента `capture` в `removeEventListener(..., true)`, когда снимаем обработчик.

Если есть несколько обработчиков одного события, назначенных `addEventListener` на один элемент, в рамках одной фазы, то их порядок срабатывания – тот же, в котором они установлены:

```
elem.addEventListener("click", e => alert(1)); // всегда работает
перед следующим
elem.addEventListener("click", e => alert(2));
```

## 21. Делегирование событий. Действия браузера по умолчанию.

Всплытие и перехват событий позволяет реализовать один из самых важных приёмов разработки – *делегирование*. Идея в том, что если есть много элементов, события на которых нужно обрабатывать похожим образом, то вместо того, чтобы назначать обработчик каждому, назначается один обработчик на их общего предка. Из него можно получить целевой элемент `event.target`, понять на каком именно потомке произошло событие и обработать его.

Рассмотрим пример – таблица. Её HTML (схематично):

```
<table>
  <tr>
    <th colspan="3">Квадрат <em>Bagua</em>: Направление, Элемент,
Цвет, Значение</th>
  </tr>
  <tr>
    <td>...<strong>Северо-Запад</strong>...</td>
    <td>...</td>
    <td>...</td>
  </tr>
  <tr>...ещё 2 строки такого же вида...</tr>
  <tr>...ещё 2 строки такого же вида...</tr>
</table>
```

В этой таблице всего 9 ячеек, но могло бы быть и 99, и даже 9999. Задача – реализовать подсветку ячейки `<td>` при клике. Вместо того, чтобы назначать обработчик `onclick` для каждой ячейки `<td>` (их может быть очень много) – назовем «единый» обработчик на элемент `<table>`. Он будет использовать `event.target`, чтобы получить элемент, на котором произошло событие, и подсветить его. Код будет таким:

```
let selectedTd;

table.onclick = function(event) {
  let target = event.target;

  if (target.tagName !== 'TD') return;

  highlight(target);
};

function highlight(td) {
  if (selectedTd) { // убрать существующую подсветку, если есть
    selectedTd.classList.remove('highlight');
  }
```

```

    }
    selectedTd = td;
    selectedTd.classList.add('highlight'); // подсветить новый td
  }

```

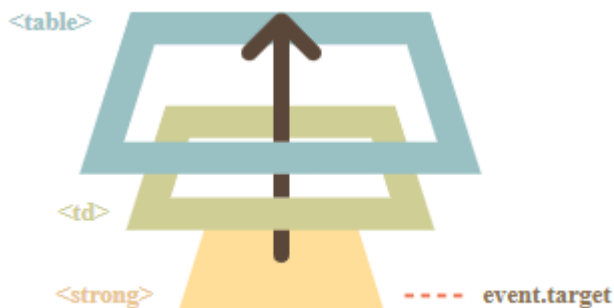
Такому коду нет разницы, сколько ячеек в таблице. Можно добавлять, удалять `<td>` из таблицы динамически в любое время, и подсветка будет стабильно работать. Однако, у текущей версии кода есть недостаток. Клик может быть не на теге `<td>`, а внутри него. В рассматриваемом примере, если взглянуть на HTML-код таблицы внимательно, видно, что ячейка `<td>` содержит вложенные теги, например, `<strong>`:

```

<td>
  <strong>Северо-Запад</strong>
  ...
</td>

```

Естественно, если клик произойдёт на элементе `<strong>`, то он станет значением `event.target`.



Внутри обработчика `table.onclick` нужно с помощью `event.target` определить, был клик внутри `<td>` или нет. Вот улучшенный код:

```

table.onclick = function(event) {
  let td = event.target.closest('td'); // (1)

  if (!td) return; // (2)

  if (!table.contains(td)) return; // (3)

  highlight(td); // (4)
};

```

Рассмотрим пример:

1. Метод `elem.closest(selector)` возвращает ближайшего предка, соответствующего селектору. В данном случае нужен `<td>`, находящийся выше по дереву от исходного элемента.
2. Если `event.target` не содержится внутри элемента `<td>`, то вызов вернёт `null`, и ничего не произойдёт.
3. Если таблицы вложенные, `event.target` может содержать элемент `<td>`, находящийся вне текущей таблицы. В таких случаях надо проверить действительно ли это `<td>` рассматриваемой таблицы.
4. И если это так, то подсветить его.

В итоге получится короткий код подсветки, быстрый и эффективный, не зависящий от того, сколько всего в таблице `<td>`.

Есть и другие применения делегирования. Например, нужно сделать меню с разными кнопками: «Сохранить (save)», «Загрузить (load)», «Поиск (search)» и т.д. И есть объект с соответствующими методами `save`, `load`, `search`. Надо добавить один обработчик для всего меню и атрибуты `data-action` для каждой кнопки в соответствии с методами, которые они вызывают:

```
<button data-action="save">Нажмите, чтобы Сохранить</button>
```

Обработчик считывает содержимое атрибута и выполняет метод. Рабочий пример:

```
<div id="menu">
  <button data-action="save">Сохранить</button>
  <button data-action="load">Загрузить</button>
  <button data-action="search">Поиск</button>
</div>

<script>
  class Menu {
    constructor(elem) {
      this._elem = elem;
      elem.onclick = this.onClick.bind(this); // (*)
    }

    save() {
      alert('сохраняю');
    }

    load() {
      alert('загружаю');
    }

    search() {
      alert('ищу');
    }

    onClick(event) {
```

```

        let action = event.target.dataset.action;
        if (action) {
            this[action]();
        }
    };
}

new Menu(menu);
</script>

```

Сохранить Загрузить Поиск

Обратите внимание, что метод `this.onClick` в строке, отмеченной звёздочкой (\*), привязывается к контексту текущего объекта `this`. Это важно, т.к. иначе `this` внутри него будет ссылаться на DOM-элемент (`elem`), а не на объект `Menu`, и `this[action]` будет не тем, что нужно.

Здесь преимущество делегирования заключается в следующем:

- Не нужно писать код, чтобы присвоить обработчик каждой кнопке. Достаточно просто создать один метод и поместить его в разметку.
- Структура HTML становится по-настоящему гибкой. Можно добавлять/удалять кнопки в любое время.

Также можно использовать классы `.action-save`, `.action-load`, но подход с использованием атрибутов `data-action` является более семантическим. Их можно использовать и для стилизации в правилах CSS.

### Приём проектирования «поведение»

Делегирование событий можно использовать для добавления элементам «поведения» (`behavior`), декларативно задавая обработчики установкой специальных HTML-атрибутов и классов. Приём проектирования «поведение» состоит из двух частей:

1. Элементу устанавливается пользовательский атрибут, описывающий его поведение.
2. При помощи делегирования назначается обработчик на документ, который ловит все клики (или другие события) и, если элемент имеет нужный атрибут – производит соответствующее действие.

Например, здесь HTML-атрибут `data-counter` добавляет кнопкам поведение: «увеличить значение при клике»:

Счётчик: `<input type="button" value="1" data-counter>`  
 Ещё счётчик: `<input type="button" value="2" data-counter>`

```

<script>
    document.addEventListener('click', function(event) {

        if (event.target.dataset.counter !== undefined) { // если есть
атрибут
            event.target.value++;

```



```

    }
  });
</script>

```

Счётчик:  Ещё счётчик:

Счётчик:  Ещё счётчик:

Если нажать на кнопку – значение увеличится.

Элементов с атрибутом data-counter может быть сколько угодно. Новые могут добавляться в HTML-код в любой момент. При помощи делегирования, фактически, добавляется новый «псевдостандартный» атрибут в HTML, который добавляет элементу новую возможность («поведение»).

Когда устанавливается обработчик событий на объект document, всегда надо использовать метод addEventListener, а не document.on<событие>, т.к. в случае последнего могут возникать конфликты: новые обработчики будут перезаписывать уже существующие. Для реального проекта совершенно нормально иметь много обработчиков на элементе document, установленных из разных частей кода.

Ещё один пример поведения: при клике на элемент с атрибутом data-toggle-id скрывается/показывается элемент с заданным id:

```

<button data-toggle-id="subscribe-mail">
  Показать форму подписки
</button>

<form id="subscribe-mail" hidden>
  Ваша почта: <input type="email">
</form>

<script>
  document.addEventListener('click', function(event) {
    let id = event.target.dataset.toggleId;
    if (!id) return;

    let elem = document.getElementById(id);

    elem.hidden = !elem.hidden;
  });
</script>

```

Показать форму подписки

Показать форму подписки

Ваша почта:

Теперь для того, чтобы добавить скрытие-раскрытие любому элементу, даже не надо знать JavaScript, можно просто написать атрибут `data-toggle-id`. Это бывает очень удобно – не нужно писать JavaScript-код для каждого элемента, который должен так себя вести. Просто используем поведение. Обработчики на уровне документа сделают это возможным для элемента в любом месте страницы.

Можно комбинировать несколько вариантов поведения на одном элементе. Шаблон «поведение» может служить альтернативой для фрагментов JS-кода в вёрстке.

Многие события автоматически влекут за собой действие браузера.

Например:

- клик по ссылке инициирует переход на новый URL;
- нажатие на кнопку «отправить» в форме – отсылку её на сервер;
- зажатие кнопки мыши над текстом и её движение в таком состоянии – инициирует его выделение.

Если событие обрабатывается в JavaScript, то зачастую такое действие браузера не нужно. Его можно отменить.

### Отмена действия браузера

Есть два способа отменить действие браузера:

- Основной способ – это воспользоваться объектом `event`. Для отмены действия браузера существует стандартный метод `event.preventDefault()`.
- Если же обработчик назначен через `on<событие>` (не через `addEventListener`), то также можно вернуть `false` из обработчика.

В следующем примере при клике по ссылке переход не произойдет:

```
<a href="/" onclick="return false">Нажми здесь</a>  
или  
<a href="/" onclick="event.preventDefault()">здесь</a>
```

Обычно значение, которое возвращает обработчик события, игнорируется. Единственное исключение – это `return false` из обработчика, назначенного через `on<событие>`. В других случаях `return` не нужен, он никак не обрабатывается.

Рассмотрим пример меню для сайта:

```
<ul id="menu" class="menu">  
  <li><a href="/html">HTML</a></li>  
  <li><a href="/javascript">JavaScript</a></li>  
  <li><a href="/css">CSS</a></li>  
</ul>
```

Данный пример при помощи CSS может выглядеть так:

HTML

JavaScript

CSS

В HTML-разметке все элементы меню являются не кнопками, а ссылками, то есть тегами `<a>`. В этом подходе есть некоторые преимущества, например:

- некоторые посетители очень любят сочетание «правый клик – открыть в новом окне». Если использовать `<button>` или `<span>`, то данное сочетание работать не будет;
- поисковые движки переходят по ссылкам `<a href="...">` при индексации.

Поэтому в разметке используется `<a>`. Но в примере требуется обрабатывать клики в JavaScript, а стандартное действие браузера (переход по ссылке) – отменить. Например, вот так:

```
menu.onclick = function(event) {  
    if (event.target.nodeName !== 'A') return;  
  
    let href = event.target.getAttribute('href');  
    alert( href );  
  
    return false; // отменить действие браузера (переход по ссылке)  
};
```

Если убрать `return false`, то после выполнения обработчика события браузер выполнит «действие по умолчанию» – переход по адресу из `href`. А это здесь не нужно, так как клик обрабатывается.

Использование здесь делегирования событий делает меню очень гибким. Можно добавить вложенные списки и стилизовать их с помощью CSS – обработчик не потребует изменений.

Некоторые события естественным образом вытекают друг из друга. Если отменить первое событие, то последующие не возникнут. Например, событие `mousedown` для поля `<input>` приводит к фокусировке на нём и запускает событие `focus`. Если отменить событие `mousedown`, то фокусирования не произойдёт.

В следующем примере если нажать на первом `<input>` – произойдет событие `focus`. Но если нажать по второму элементу, то события `focus` не будет.

```
<input value="Фокус работает" onfocus="this.value=''">  
<input onmousedown="return false" onfocus="this.value=''"  
value="Клики меня">
```

Фокус работает

Клики меня

Это потому, что отменено стандартное действие `mousedown`. Впрочем, фокусировка на элементе всё ещё возможна, если использовать другой способ. Например, нажатием клавиши `Tab` можно перейти от первого поля ввода ко второму. Но только не через клик мышью на элемент, это больше не работает.

Необязательная опция `passive: true` для `addEventListener` сигнализирует браузеру, что обработчик не собирается выполнять `preventDefault()`. Это может быть полезно, так как есть некоторые события, как `touchmove` на мобильных устройствах (когда пользователь перемещает палец по экрану), которое по умолчанию начинает прокрутку, но можно отменить это действие, используя `preventDefault()` в обработчике. Поэтому, когда браузер обнаружит такое событие, он должен для начала запустить все обработчики и после, если `preventDefault` не вызывается нигде, он может начать прокрутку. Это может вызвать ненужные задержки в пользовательском интерфейсе.

Опция `passive: true` сообщает браузеру, что обработчик не собирается отменять прокрутку. Тогда браузер начинает её немедленно, обеспечивая максимально плавный интерфейс, параллельно обрабатывая событие. Для некоторых браузеров (Firefox, Chrome) опция `passive` по умолчанию включена в `true` для таких событий, как `touchstart` и `touchmove`.

Свойство `event.defaultPrevented` установлено в `true`, если действие по умолчанию было предотвращено, и `false`, если нет. Рассмотрим практическое применение этого свойства для улучшения архитектуры. `event.defaultPrevented` можно использовать вместо остановки всплытия с помощью `event.stopPropagation()`, чтобы просигнализировать другим обработчикам, что событие обработано.

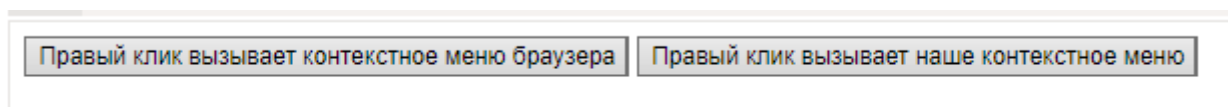
Рассмотрим практический пример. По умолчанию браузер при событии `contextmenu` (клик правой кнопкой мыши) показывает контекстное меню со стандартными опциями. Можно отменить событие по умолчанию и показать своё меню, как здесь:

```
<button>Правый клик вызывает контекстное меню браузера</button>
```

```
<button oncontextmenu="alert('Рисуем наше меню'); return false">
```

Правый клик вызывает наше контекстное меню

```
</button>
```



Теперь в дополнение к этому контекстному меню реализуем контекстное меню для всего документа. При правом клике должно показываться ближайшее контекстное меню.

```
<p>Правый клик здесь вызывает контекстное меню документа</p>
```

```
<button id="elem">Правый клик здесь вызывает контекстное меню кнопки</button>
```

```

<script>
  elem.oncontextmenu = function(event) {
    event.preventDefault();
    alert("Контекстное меню кнопки");
  };

  document.oncontextmenu = function(event) {
    event.preventDefault();
    alert("Контекстное меню документа");
  };
</script>

```

Правый клик здесь вызывает контекстное меню документа

Правый клик здесь вызывает контекстное меню кнопки

Проблема заключается в том, что когда происходит клик по элементу `elem`, то получаем два меню: контекстное меню для кнопки и (событие всплывает вверх) контекстное меню для документа. Чтобы это исправить, надо остановить всплытие когда обрабатывается правый клик в обработчике на кнопке, и вызвать `event.stopPropagation()`:

```

<p>Правый клик вызывает меню документа</p>
<button id="elem">Правый клик вызывает меню кнопки (добавлен
event.stopPropagation)</button>

```

```

<script>
  elem.oncontextmenu = function(event) {
    event.preventDefault();
    event.stopPropagation();
    alert("Контекстное меню кнопки");
  };

  document.oncontextmenu = function(event) {
    event.preventDefault();
    alert("Контекстное меню документа");
  };
</script>

```

Правый клик вызывает меню документа

Правый клик вызывает меню кнопки (добавлен `event.stopPropagation`)

Теперь контекстное меню для кнопки работает как задумано. Но навсегда запрещен доступ к информации о правых кликах для любого внешнего кода, включая счётчики, которые могли бы собирать статистику, и т.п. Это неудобно.

Альтернативным решением было бы проверить в обработчике `document`, было ли отменено действие по умолчанию? Если да, тогда событие было обработано, и не нужно на него реагировать.

```

<p>Правый клик вызывает меню документа (добавлена проверка
event.defaultPrevented)</p>
<button id="elem">Правый клик вызывает меню кнопки</button>

<script>
  elem.oncontextmenu = function(event) {
    event.preventDefault();
    alert("Контекстное меню кнопки");
  };

  document.oncontextmenu = function(event) {
    if (event.defaultPrevented) return;

    event.preventDefault();
    alert("Контекстное меню документа");
  };
</script>

```

Правый клик вызывает меню документа (добавлена проверка event.defaultPrevented)

Правый клик вызывает меню кнопки

Сейчас всё работает правильно. Если есть вложенные элементы и каждый из них имеет контекстное меню, то код также будет работать. Просто надо убедиться, что осуществляется проверка event.defaultPrevented в каждом обработчике contextmenu.

event.stopPropagation() и event.preventDefault() (также известный как return false) – это две разные функции. Они никак не связаны друг с другом.

## 22. Генерация событий.

Можно не только назначать обработчики, но и генерировать события из JavaScript-кода. Пользовательские события могут быть использованы при создании графических компонентов. Например, корневой элемент меню может генерировать события, относящиеся к этому меню: open (меню раскрыто), select (выбран пункт меню) и т.п. Также можно генерировать встроенные события, такие как click, mousedown и другие, что бывает полезно для автоматического тестирования.

### Конструктор Event

Встроенные классы для событий формируют иерархию, аналогично классам для DOM-элементов. Её корнем является встроенный класс Event. Событие встроенного класса Event можно создать так:

```
let event = new Event(type[, options]);
```

Где:

- type – тип события, строка, например, "click" или другая;
- options – объект с двумя необязательными свойствами:
  - bubbles: true/false – если true, тогда событие всплывает;
  - cancelable: true/false – если true, тогда можно отменить действие по умолчанию.

По умолчанию оба свойства установлены в false: {bubbles: false, cancelable: false}.

После того, как объект события создан, надо запустить событие на элементе, вызвав метод `elem.dispatchEvent(event)`. Затем обработчики отреагируют на него, как будто это обычное встроенное событие. Если при создании указан флаг bubbles, то оно будет всплывать. В примере ниже событие click инициируется JavaScript-кодом так же, как если бы кликнули по кнопке:

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

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

В то же время можно легко отличить «настоящее» событие от сгенерированного кодом. Свойство `event.isTrusted` принимает значение true для событий, порождаемых реальными действиями пользователя, и false для генерируемых кодом.

Можно создать всплывающее событие с именем "hello" и поймать его на document. Всё, что нужно сделать – это установить флаг bubbles в true:

```
<h1 id="elem">Привет из кода!</h1>

<script>
  document.addEventListener("hello", function(event) { // (1)
    alert("Привет от " + event.target.tagName); // Привет от H1
  });

  // запуск события на элементе
  let event = new Event("hello", {bubbles: true}); // (2)
  elem.dispatchEvent(event);
</script>
```

Надо использовать `addEventListener` для собственных событий, т.к. `on<event>`-свойства существуют только для встроенных событий, то есть `document.onhello` не работает. Надо передавать флаг `bubbles:true`, иначе событие не будет всплывать.

Механизм всплытия идентичен как для встроенного события (click), так и для пользовательского события (hello). Также одинакова работа фаз всплытия и погружения.

### Конструкторы MouseEvent, KeyboardEvent и другие

Для некоторых конкретных типов событий есть свои специфические конструкторы. Вот небольшой список конструкторов для различных событий пользовательского интерфейса, которые можно найти в спецификации UI Event: UIEvent, FocusEvent, MouseEvent, WheelEvent, KeyboardEvent.

Стоит использовать их вместо new Event, если надо создавать такие события. К примеру, new MouseEvent("click"). Специфический конструктор позволяет указать стандартные свойства для данного типа события. Например, clientX/clientY для события мыши:

```
let event = new MouseEvent("click", {
  bubbles: true,
  cancelable: true,
  clientX: 100,
  clientY: 100
});

alert(event.clientX); // 100
```

Обратите внимание: это нельзя было бы сделать с обычным конструктором Event. Проверим:

```
let event = new Event("click", {
  bubbles: true, // только свойства bubbles и cancelable
  cancelable: true, // работают в конструкторе Event
  clientX: 100,
  clientY: 100
});

alert(event.clientX); // undefined, неизвестное свойство
проигнорировано
```

Впрочем, использование конкретного конструктора не является обязательным, можно обойтись Event, а свойства записать в объект отдельно, после создания, вот так: event.clientX=100. Здесь это скорее вопрос удобства и желания следовать правилам. События, которые генерирует браузер, всегда имеют правильный тип. Полный список свойств по типам событий можно найти в спецификации, например для MouseEvent.

### Пользовательские события

Для генерации пользовательских событий, таких как "hello", следует использовать конструктор new CustomEvent.



Технически CustomEvent абсолютно идентичен Event за исключением одной небольшой детали. У второго аргумента-объекта есть дополнительное свойство detail, в котором можно указывать информацию для передачи в событие. Например:

```
<h1 id="elem">Привет для Васи!</h1>

<script>
  elem.addEventListener("hello", function(event) {
    alert(event.detail.name);
  });

  elem.dispatchEvent(new CustomEvent("hello", {
    detail: { name: "Вася" }
  }));
</script>
```

Свойство detail может содержать любые данные. Надо сказать, что никто не мешает и в обычное new Event записать любые свойства. Но CustomEvent предоставляет специальное поле detail во избежание конфликтов с другими свойствами события. Класс события содержит информацию о том, что это за событие, и если оно не браузерное, а пользовательское, то стоит всё-таки использовать CustomEvent.

На сгенерированном событии обработчик может вызвать метод event.preventDefault(), если задан флаг cancelable:true. Для пользовательских событий, названия которых браузеру неизвестны, соответственно, нет никаких действий браузера по умолчанию. Но код, который генерирует событие, может предусматривать какие-то ещё действия после dispatchEvent.

Вызов event.preventDefault() является возможностью для обработчика события сообщить в сгенерировавший событие код, что эти действия надо отменить. Тогда вызов elem.dispatchEvent(event) возвратит false. И код, сгенерировавший событие, приостановит выполнение.

В примере ниже есть функция hide(), которая при вызове генерирует событие "hide" на элементе #rabbit, уведомляя всех интересующихся, что кролик собирается спрятаться. Любой обработчик может узнать об этом, подписавшись на событие hide через rabbit.addEventListener('hide',...) и, при желании, отменить действие по умолчанию через event.preventDefault(). Тогда кролик не исчезнет:

```
<pre id="rabbit">
  | \   / |
  \|_  | /
  / . . \
  =\_Y_/=
  {>o<}
</pre>
```

```

<script>
  function hide() {
    let event = new CustomEvent("hide", {
      cancelable: true
    });
    if (!rabbit.dispatchEvent(event)) {
      alert('действие отменено обработчиком');
    } else {
      rabbit.hidden = true;
    }
  }

  rabbit.addEventListener('hide', function(event) {
    if (confirm("Вызвать preventDefault?")) {
      event.preventDefault();
    }
  });

  setTimeout(hide, 2000);

</script>

```

Обычно события обрабатываются асинхронно. То есть, если браузер обрабатывает onclick и в процессе этого произойдёт новое событие, то оно ждёт, пока закончится обработка onclick. Исключением является ситуация, когда событие инициировано из обработчика другого события. Тогда управление сначала переходит в обработчик вложенного события и уже после этого возвращается назад. В примере ниже событие menu-open обрабатывается синхронно во время обработки onclick:

```

<button id="menu">Меню (нажми меня)</button>

<script>
  menu.onclick = function() {
    alert(1);

    menu.dispatchEvent(new CustomEvent("menu-open", {
      bubbles: true
    }));

    alert(2);
  };

  document.addEventListener('menu-open', () => alert('вложенное событие'))
</script>

```

Обратите внимание, что вложенное событие `menu-open` всплывает и обрабатывается на `document`. Обработка вложенного события полностью завершается до того, как управление возвращается во внешний код (`onclick`). Это справедливо не только для `dispatchEvent`, но и для других случаев. JavaScript в обработчике события может вызвать другие методы, которые приведут к другим событиям – они тоже обрабатываются синхронно.

Также можно либо поместить `dispatchEvent` (или любой другой код иницирующий события) в конец обработчика `onclick`, либо обернуть такой код в `setTimeout(..., 0)`:

```
<button id="menu">Меню (нажми меня)</button>

<script>
  menu.onclick = function() {
    alert(1);

    setTimeout(() => menu.dispatchEvent(new CustomEvent("menu-open", {
      bubbles: true
    })));

    alert(2);
  };

  document.addEventListener('menu-open', () => alert('вложенное
событие'))
</script>
```

Теперь `dispatchEvent` запускается асинхронно после исполнения текущего кода, включая `mouse.onclick`, поэтому обработчики полностью независимы.

## 23. События мыши. События `mouseover/out`, `mouseenter/leave`.

События мыши происходят не только от манипуляций мышью, но и эмулируются на сенсорных устройствах, чтобы сделать их совместимыми.

### Типы событий мыши

Можно разделить события мыши на две категории: простые и комплексные.

Простые события (наиболее часто используемые):

- `mousedown/mouseup` – кнопка мыши нажата/отпущена над элементом;
- `mouseover/mouseout` – курсор мыши появляется над элементом и уходит с него;
- `mousemove` – каждое движение мыши над элементом генерирует это событие.

Комплексные события:

- click – вызывается при mousedown , а затем mouseup над одним и тем же элементом, если использовалась левая кнопка мыши;
- contextmenu – вызывается при mousedown правой кнопкой мыши;
- dblclick – вызывается двойным кликом на элементе.

Комплексные события состоят из простых. С ними удобнее работать.

### Порядок событий

Одно действие может вызвать несколько событий. Например, клик мышью вначале вызывает mousedown, когда кнопка нажата, затем mouseup и click, когда она отпущена. В случае, когда одно действие инициирует несколько событий, порядок их выполнения фиксирован. То есть обработчики событий вызываются в следующем порядке: mousedown → mouseup → click. События обрабатываются в той же последовательности: onmouseup завершается до того, как запускается onclick.

События, связанные с кликом, всегда имеют свойство which, которое позволяет определить нажатую кнопку мыши. Это свойство не используется для событий click и contextmenu, поскольку первое происходит только при нажатии левой кнопкой мыши, а второе – правой. События mousedown и mouseup срабатывают на любой кнопке и свойство which позволяет различать между собой «нажатие правой кнопки» и «нажатие левой кнопки» мыши.

Есть три возможных значения:

- event.which == 1 – левая кнопка;
- event.which == 2 – средняя кнопка;
- event.which == 3 – правая кнопка.

### Модификаторы: shift, alt, ctrl и meta

Все события мыши включают в себя информацию о нажатых клавишах-модификаторах. Их свойства: shiftKey, altKey, ctrlKey, metaKey (Cmd для Mac). Например, кнопка внизу работает только при комбинации Alt+Shift+клик:

```
<button id="button">Нажми Alt+Shift+Click на мне!</button>
```

```
<script>
  button.onclick = function(event) {
    if (event.altKey && event.shiftKey) {
      alert('Ура!');
    }
  };
</script>
```

---

Нажми Alt+Shift+Click на мне!

В Windows и Linux клавишами-модификаторами являются Alt, Shift и Ctrl. На Mac есть ещё одна: Cmd, она соответствует свойству metaKey. В большинстве случаев, когда в Windows/Linux используется Ctrl, на Mac пользователи используют Cmd. Поэтому, когда пользователь Windows нажимает Ctrl+Enter и Ctrl+A, пользователь Mac нажимает Cmd+Enter или Cmd+A, и так далее, большинство приложений используют Cmd вместо Ctrl. Поэтому, если надо поддерживать такие комбинации, как Ctrl+клик, то для Mac имеет смысл использовать Cmd+клик. Это удобнее для пользователей Mac.

Левый клик в сочетании с Ctrl интерпретируется как правый клик на Mac и генерирует событие contextmenu, а не click как на Windows/Linux. Поэтому, если надо, чтобы пользователям всех операционных систем было удобно, то вместе с ctrlKey нужно использовать metaKey. Для JS-кода это означает, что надо проверить if (event.ctrlKey || event.metaKey).

Все события мыши имеют координаты двух видов: относительно окна (event.clientX и event.clientY) и относительно документа (event.pageX и event.pageY).

Клики мышью имеют побочный эффект, который может быть неудобен в некоторых интерфейсах: двойной клик мышью выделяет текст. Это действие браузера по умолчанию при наступлении события mousedown. Поэтому альтернативным решением проблемы будет обработать событие mousedown и предотвратить его:

```
<b ondblclick="alert('Клик!')" onmousedown="return false">  
  Сделайте двойной клик на мне  
</b>
```

Теперь выделенный жирным элемент не выделяется при двойном клике. Текст внутри него по-прежнему можно выделить. Однако, выделение должно начаться не на самом тексте, а до него или после.

Вместо предотвращения выделения, можно отменить его «постфактум» в обработчике событий. Например, так:

```
<b ondblclick="getSelection().removeAllRanges()">  
  Сделайте двойной клик на мне  
</b>
```

При двойном клике на элементе, выделенном жирным шрифтом, выделение появится и тут же будет немедленно снято. Выглядит это не очень красиво.

Если надо отключить выделение для защиты контента от копирования, то можно использовать другое событие: oncopy.

```
<div oncopy="alert('Копирование запрещено!');return false">  
  Уважаемый пользователь,
```

Копирование информации запрещено.  
</div>

Скопировать текст в <div> не получится, потому что срабатывание события onclick по умолчанию запрещено.

Рассмотрим подробнее события, возникающие при движении указателя (курсора) мыши над элементами страницы.

### Mouseover/mouseout, relatedTarget

Событие mouseover происходит в момент, когда курсор оказывается над элементом, а событие mouseout – в момент, когда курсор уходит с элемента.



Эти события являются особенными, потому что у них имеется свойство relatedTarget. Оно дополняет target. Когда мышь переходит с одного элемента на другой, то один из них будет храниться в target, а другой в relatedTarget.

Для события mouseover:

- event.target – это элемент, на который курсор перешёл;
- event.relatedTarget – это элемент, с которого курсор ушёл.

Для события mouseout наоборот:

- event.target – это элемент, с которого курсор ушёл;
- event.relatedTarget – это элемент, на который курсор перешёл.

Свойство relatedTarget может быть null, это означает, что указатель мыши перешёл не с другого элемента, а из-за пределов окна браузера. Или, наоборот, ушёл за пределы окна.

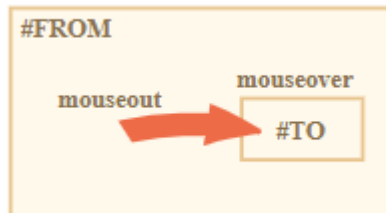
Событие mousemove происходит при движении мыши. Однако, это не означает, что указанное событие генерируется при прохождении каждого пикселя. Браузер периодически проверяет позицию курсора и, заметив изменения, генерирует события mousemove. Это означает, что если пользователь двигает мышкой очень быстро, то некоторые DOM-элементы могут быть пропущены:



Если курсор мыши двигается очень быстро с #FROM на #TO элемент, как это показано выше, то лежащие между ними элементы <div> (или некоторые из них) могут быть пропущены. Событие mouseout может

запуститься на элементе #FROM и затем сразу же сгенерируется mouseover на элементе #TO.

Представьте ситуацию – курсор мыши перешёл на элемент. Сгенерировано событие mouseover. Затем курсор перешёл на дочерний элемент, и сгенерировано mouseout. То есть курсор всё ещё на элементе, но событие mouseout.



По логике браузера, курсор мыши может быть только над одним элементом в любой момент времени – над самым глубоко вложенным (и верхним по z-index). Таким образом, если курсор переходит на другой элемент (пусть даже дочерний), то он покидает предыдущий.

### **События mouseenter и mouseleave**

События mouseenter/mouseleave похожи на mouseover/mouseout. Они тоже генерируются, когда курсор мыши переходит на элемент или покидает его. Но есть и пара важных отличий:

1. Переходы внутри элемента по дочерним элементам не считаются.
2. События mouseenter/mouseleave не всплывают.

Когда курсор становится над элементом – генерируется mouseenter, и не имеет значения, где именно находится курсор внутри элемента. Событие mouseleave происходит, когда курсор покидает элемент.

События mouseenter/leave простые и легкие в использовании, но они не всплывают, а это значит что их нельзя делегировать.

## **24. Drag'n'Drop с событиями мыши. События клавиатуры: keyup, keydown. Прокрутка: событие scroll.**

События Drag'n'Drop генерируются, когда пользователь захватывает элемент мышкой и перетаскивает его. Основной алгоритм Drag'n'Drop выглядит так:

1. Отслеживание события mousedown на переносимом элементе.
2. При нажатии – подготовка элемента к перемещению (например, создание его копии).
3. При наступлении mousemove перемещение элемента на новые координаты путём смены left/top и position:absolute.
4. На mouseup (при отпускании кнопки мыши) – остановка переноса элемента.

В следующем примере эти шаги реализованы для переноса мяча:

```

ball.onmousedown = function(event) { // (1) отследить нажатие

    // (2) подготовить к перемещению: разместить поверх остального
    // содержимого и в абсолютных координатах
    ball.style.position = 'absolute';
    ball.style.zIndex = 1000;
    // переместим в body, чтобы мяч был точно не внутри
    position:relative
    document.body.append(ball);
    // и установим абсолютно спозиционированный мяч под курсор

    moveAt(event.pageX, event.pageY);

    // передвинуть мяч под координаты курсора
    // и сдвинуть на половину ширины/высоты для центрирования
    function moveAt(pageX, pageY) {
        ball.style.left = pageX - ball.offsetWidth / 2 + 'px';
        ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
    }

    function onMouseMove(event) {
        moveAt(event.pageX, event.pageY);
    }

    // (3) перемещать по экрану
    document.addEventListener('mousemove', onMouseMove);

    // (4) положить мяч, удалить более ненужные обработчики событий
    ball.onmouseup = function() {
        document.removeEventListener('mousemove', onMouseMove);
        ball.onmouseup = null;
    };
};

```

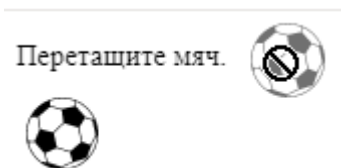
Если запустить этот код, то можно заметить, что при начале переноса мяч «раздваивается» и переносится не сам мяч, а его копия. Всё потому, что браузер имеет свой собственный Drag'n'Drop, который автоматически запускается и вступает в конфликт с нашим. Это происходит именно для картинок и некоторых других элементов. Его нужно отключить:

```

ball.ondragstart = function() {
    return false;
};

```

Ещё одна особенность правильного Drag'n'Drop – событие `mousemove` отслеживается на `document`, а не на `ball`. С первого взгляда кажется, что мышь





всегда над мячом и обработчик `mousemove` можно повесить на сам мяч, а не на документ. Но не стоит забывать, что событие `mousemove` возникает хоть и часто, но не для каждого пикселя. Поэтому из-за быстрого движения курсор может сместиться с мяча и оказаться где-нибудь в середине документа (или даже за пределами окна). Поэтому стоит отслеживать `mousemove` на всём `document`, чтобы поймать его.

В примерах выше мяч позиционируется так, что его центр оказывается под курсором мыши:

```
ball.style.left = pageX - ball.offsetWidth / 2 + 'px';  
ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
```

У такого позиционирования есть один недостаток: в самом начале переноса, особенно если мячик «взят» за край – он резко «прыгает» центром под курсор мыши. Было бы лучше, если бы изначальный сдвиг курсора относительно элемента сохранялся, т.е. где захватили, за ту «часть элемента» и переносим:



1. Когда пользователь нажимает на мячик (`mousedown`) – нужно зафиксировать расстояние от курсора до левого верхнего угла шара в переменных `shiftX/shiftY` и запомнить это расстояние при перетаскивании.

Чтобы получить этот сдвиг, можно вычесть координаты:

```
// onmousedown  
let shiftX = event.clientX - ball.getBoundingClientRect().left;  
let shiftY = event.clientY - ball.getBoundingClientRect().top;
```

2. Далее при переносе он позиционируется с тем же сдвигом относительно указателя мыши (`position:absolute`):

```
// onmousemove  
ball.style.left = event.pageX - shiftX + 'px';  
ball.style.top = event.pageY - shiftY + 'px';
```

Итоговый код с правильным позиционированием:

```
ball.onmousedown = function(event) {  
  
    let shiftX = event.clientX - ball.getBoundingClientRect().left;  
    let shiftY = event.clientY - ball.getBoundingClientRect().top;  
  
    ball.style.position = 'absolute';  
    ball.style.zIndex = 1000;  
    document.body.append(ball);  
  
    moveAt(event.pageX, event.pageY);  
}
```

```

function moveAt(pageX, pageY) {
    ball.style.left = pageX - shiftX + 'px';
    ball.style.top = pageY - shiftY + 'px';
}

function onMouseMove(event) {
    moveAt(event.pageX, event.pageY);
}

document.addEventListener('mousemove', onMouseMove);

ball.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
    ball.onmouseup = null;
};

};

ball.ondragstart = function() {
    return false;
};

```

### Цели переноса (draggable)

В предыдущих примерах мяч можно было бросить просто где угодно в пределах окна. В реальности пользователь обычно берёт один элемент и перетаскивает в другой. Например, файл в папку или что-то ещё. Т.е. он берёт перетаскиваемый (draggable) элемент и помещает его в другой элемент «цель переноса» (draggable).

Т.е. нужно знать, куда пользователь положил элемент в конце переноса и над какой потенциальной целью (например, изображение папки) он находится в процессе переноса, чтобы подсветить её.

Существует метод `document.elementFromPoint(clientX, clientY)`. Он возвращает наиболее глубоко вложенный элемент по заданным координатам окна (или `null`, если указанные координаты находятся за пределами окна). Таким образом, из любого обработчика событий мыши можно обнаружить потенциальную цель переноса вот так:

```

ball.hidden = true; // (*)
let elemBelow = document.elementFromPoint(event.clientX,
event.clientY);
ball.hidden = false;
// elemBelow - элемент под мячом (цель переноса)

```

Обратите внимание, что нужно спрятать мяч перед вызовом функции (\*). Иначе получим координаты мяча, ведь это и есть элемент непосредственно под указателем: `elemBelow=ball`. Можно использовать этот код для проверки того, над каким элементом находится мяч, в любое время. И

обработать окончание переноса, когда оно случится. Расширенный код `onMouseMove` с поиском потенциальных целей переноса:

```
let currentDroppable = null; // потенциальная цель переноса, над
которой мяч сейчас находится

function onMouseMove(event) {
  moveAt(event.pageX, event.pageY);

  ball.hidden = true;
  let elemBelow = document.elementFromPoint(event.clientX,
event.clientY);
  ball.hidden = false;

  if (!elemBelow) return; // (1)

  let droppableBelow = elemBelow.closest('.droppable'); // (2)

  if (currentDroppable !== droppableBelow) { // (3)

    if (currentDroppable) {
      leaveDroppable(currentDroppable); // (4)
    }
    currentDroppable = droppableBelow;
    if (currentDroppable) {
      enterDroppable(currentDroppable);
    }
  }
}
```

В приведённом примере, когда мяч перетаскивается через футбольные ворота, ворота подсвечиваются. Событие `mousemove` может произойти, когда курсор находится за пределами окна, тогда `elementFromPoint` вернёт `null` (1). Потенциальные цели переноса помечены классом `droppable` (2). Надо проверить перемещался элемент или нет, тогда он либо располагается над целью, либо нет (3). При этом и `currentDroppable`, и `droppableBelow` могут быть равны `null`. `currentDroppable=null`, если элемент располагался не над `droppable` до этого события (например, над пустым пространством), `droppableBelow=null`, если элемент располагается не над `droppable` именно сейчас, во время этого события. Если элемент перемещен из `droppable`, то удаляем подсветку (4).

Перетащите мяч.



Теперь в течение всего процесса в переменной `currentDroppable` хранится текущая потенциальная цель переноса, над которой сейчас располагается мяч, можно её подсветить или сделать что-то ещё.

События прокрутки позволяют реагировать на прокрутку страницы или элемента. Есть много хороших вещей, которые можно сделать при этом.

Например:

- Показать/скрыть дополнительные элементы управления или информацию, основываясь на том, в какой части документа находится пользователь.
- Загрузить больше данных, когда пользователь прокручивает страницу вниз до конца.

Вот небольшая функция для отображения текущей прокрутки:

```
window.addEventListener('scroll', function() {  
    document.getElementById('showScroll').innerHTML = pageYOffset + 'px';  
});
```

В действии:

Текущая прокрутка = 563.3333740234375px

Событие scroll работает как на window, так и на прокручиваемых элементах.

Предотвращение прокрутки

Как можно сделать что-то непрокручиваемым? Нельзя предотвратить прокрутку, используя event.preventDefault() в обработчике onscroll, потому что он срабатывает после того, как прокрутка уже произошла.

Но можно предотвратить прокрутку, используя event.preventDefault() на событии, которое вызывает прокрутку.

Например:

- На событии wheel – прокрутка колеса мыши («прокручивание» тачпада также его генерирует).
- На событии keydown для клавиш pageUp и pageDown.

Если поставить на них обработчики, в которых вызвать event.preventDefault(), то прокрутка не начнётся.

Иногда это может помочь, но более надёжный способ – использовать CSS, чтобы сделать что-то непрокручиваемым, например, свойство overflow. Вот несколько задач, которые вы можете решить или просмотреть, чтобы увидеть применение onscroll.

События клавиатуры должны использоваться, если необходимо обрабатывать взаимодействие пользователя с клавиатурой (в том числе виртуальной). К примеру, если нужно реагировать на стрелочные клавиши Up и Down или горячие клавиши (включая комбинации клавиш).

Событие keydown происходит при нажатии клавиши, а keyup – при отпускании.

У объекта события есть свойство key, чьё значение – символ, и свойство code – «физический код клавиши». Например, клавишу Z можно нажать с клавишей Shift и без неё. В результате получится два разных символа: z в нижнем регистре и Z в верхнем регистре.

Свойство `event.key` – это непосредственно символ, и он может различаться. Но `event.code` всегда будет тот же:

Клавиша	<code>event.key</code>	<code>event.code</code>
Z	z (нижний регистр)	KeyZ
Shift+Z	Z (Верхний регистр)	KeyZ

Если пользователь работает с разными языками, то при переключении на другой язык символ изменится с "Z" на совершенно другой. Получившееся станет новым значением `event.key`, тогда как `event.code` останется тем же: "KeyZ".

У каждой клавиши есть код, который зависит от её расположения на клавиатуре. Подробно о клавишных кодах можно прочитать в спецификации о кодах событий UI. Например:

- буквенные клавиши имеют коды по типу "Key<буква>": "KeyA", "KeyB" и т.д.;
- коды числовых клавиш строятся по принципу: "Digit<число>": "Digit0", "Digit1" и т.д.;
- код специальных клавиш – это их имя: "Enter", "Backspace", "Tab" и т.д.

Существует несколько широко распространённых раскладок клавиатуры, и в спецификации приведены клавишные коды к каждой из них.

Регистр важен: правильно KeyZ, а не keyZ. Условие `event.code=="keyZ"` работать не будет: первая буква в слове "Key" должна быть заглавная.

Если клавиша не буквенно-цифровая, например, Shift или F1, или какая-либо другая специальная клавиша, то значение свойства `event.key` примерно тоже, что и у `event.code`:

Клавиша	<code>event.key</code>	<code>event.code</code>
F1	F1	F1
Backspace	Backspace	Backspace
Shift	Shift	ShiftRight или ShiftLeft

Большинство клавиатур имеют по две клавиши Shift: слева и справа. `event.code` уточняет, какая именно из них была нажата, в то время как `event.key` сообщает о том, что вообще было нажато (Shift).

Допустим, надо обработать горячую клавишу Ctrl+Z (или Cmd+Z для Mac). Можно поставить обработчик событий на `keydown` и проверять, какая клавиша была нажата. Возникает вопрос: надо проверять значение `event.key` или `event.code`?

С одной стороны, значение `event.key` изменяется в зависимости от языка, и, если у пользователя установлено в ОС несколько языков, и он переключается между ними, нажатие на одну и ту же клавишу будет давать разные символы. Так что имеет смысл проверять `event.code`, ведь его значение всегда одно и тоже. Вот пример кода:

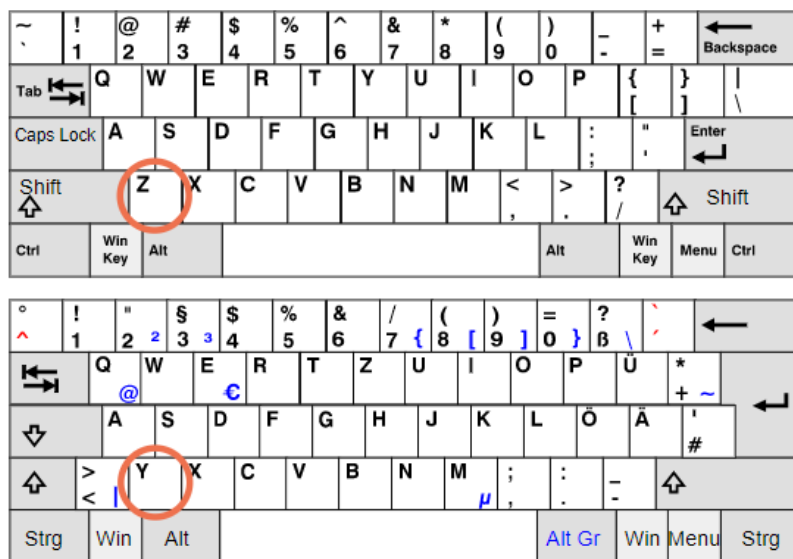
```
document.addEventListener('keydown', function(event) {  
  if (event.code == 'KeyZ' && (event.ctrlKey || event.metaKey)) {
```

```

    alert('Отменить!')
  }
});

```

С другой стороны, с event.code тоже есть проблемы. На разных раскладках к одной и той же клавише могут быть привязаны разные символы (буквы). Например, вот схема стандартной (US) раскладки («QWERTY») и под ней немецкой («QWERTZ») раскладки (предоставлены Википедией):



Для одной и той же клавиши в американской раскладке значение event.code равно «Z», в то время как в немецкой «Y». Таким образом, для пользователей с немецкой раскладкой event.code при нажатии на «Y» будет равен KeyZ.

В спецификации напрямую упоминается такое поведение.

- Преимущество event.code заключается в том, что его значение всегда остается неизменным, будучи привязанным к физическому местоположению клавиши, даже если пользователь меняет язык. Так что горячие клавиши, использующие это свойство, будут работать даже в случае переключения языка.
- event.code может содержать неправильный символ на нестандартной раскладке. Одни и те же буквы на разных раскладках могут сопоставляться с разными физическими клавишами, что приводит к разным кодам. Это происходит не со всеми кодами, а с несколькими, например, keyA, keyQ, keyZ, и не происходит со специальными клавишами, такими как Shift. Полный список проблемных кодов можно найти в спецификации.

Таким образом, чтобы отслеживать символы, зависящие от раскладки, event.key надёжнее.

При долгом нажатии клавиши возникает автоповтор: keydown срабатывает снова и снова, и когда клавишу отпускают, то отработывает keyup. Так что ситуация, когда много keydown и один keyup, абсолютно нормальна. Для событий, вызванных автоповтором, у объекта события свойство event.repeat равно true.

Действия по умолчанию весьма разнообразны, много чего можно инициировать нажатием на клавиатуре: появление символа, удаление символа, прокрутка страницы, открытие диалогового окна браузера «Сохранить» и так далее. Предотвращение стандартного действия с помощью `event.preventDefault()` работает практически во всех сценариях, кроме тех, которые происходят на уровне операционной системы. Например, комбинация `Alt+F4` инициирует закрытие браузера в Windows, как бы эта комбинация не обрабатывалась в JavaScript. Для примера, `<input>` ниже ожидает телефонный номер, так что ничего кроме чисел, `+`, `()` или `-` принято не будет:

```
<script>
  function checkPhoneKey(key) {
    return (key >= '0' && key <= '9') || key == '+' || key == '('
      || key == ')' || key == '-';
  }
</script>
<input onkeydown="return checkPhoneKey(event.key)"
placeholder="Введите номер телефона" type="tel">
```

---

---

Заметьте, что специальные клавиши по типу `Backspace`, `Left`, `Right`, `Ctrl+V` не работают как должны в поле для ввода. Это побочный эффект жесткого фильтра `checkPhoneKey`. Доработаем код:

```
<script>
function checkPhoneKey(key) {
  return (key >= '0' && key <= '9') || key == '+' || key == '(' || key
    == ')' || key == '-' ||
    key == 'ArrowLeft' || key == 'ArrowRight' || key == 'Delete' ||
    key == 'Backspace';
}
</script>
<input onkeydown="return checkPhoneKey(event.key)"
placeholder="Введите номер телефона" type="tel">
```

---

---



## 25. Свойства и методы формы.

Формы и элементы управления, такие как `<input>`, имеют множество специальных свойств и событий.

Формы в документе входят в специальную коллекцию `document.forms`. Это — так называемая «именованная» коллекция: можно использовать для получения формы как её имя, так и порядковый номер в документе.

`document.forms.my` - форма с именем "my" (`name="my"`)

`document.forms[0]` - первая форма в документе

Когда форма уже получена, любой элемент доступен в именованной коллекции `form.elements`. Например:

```
<form name="my">
  <input name="one" value="1">
  <input name="two" value="2">
</form>

<script>
  // получаем форму
  let form = document.forms.my; // <form name="my"> element

  // получаем элемент
  let elem = form.elements.one; // <input name="one"> element

  alert(elem.value); // 1
</script>
```

Может быть несколько элементов с одним и тем же именем, это часто бывает с кнопками-переключателями `radio`. В этом случае `form.elements[name]` является коллекцией, например:

```
<form>
  <input type="radio" name="age" value="10">
  <input type="radio" name="age" value="20">
</form>

<script>
let form = document.forms[0];

let ageElems = form.elements.age;

alert(ageElems[0].value); // 10, the first input value
</script>
```



Эти навигационные свойства не зависят от структуры тегов внутри формы. Все элементы, как бы глубоко они ни находились в форме, доступны в коллекции `form.elements`.

Форма может содержать один или несколько элементов `<fieldset>` внутри себя. Они также поддерживают свойство `elements`. Например:

```
<body>
  <form id="form">
    <fieldset name="userFields">
      <legend>info</legend>
      <input name="login" type="text">
    </fieldset>
  </form>

  <script>
    alert(form.elements.login); // <input name="login">

    let fieldset = form.elements.userFields;
    alert(fieldset); // HTMLFieldSetElement

    // можно получить информацию как из формы, так и из fieldset
    alert(fieldset.elements.login == form.elements.login); // true
  </script>
</body>
```

Есть более короткая запись: можно получить доступ к элементу через `form[index/name]`. Вместо `form.elements.login` можно написать `form.login`. Это также работает, но есть небольшая проблема: если надо получить элемент, а затем менять его свойство `name`, то он всё ещё будет доступен под старым именем (также, как и под новым). В этом легче разобраться на примере:

```
<form id="form">
  <input name="login">
</form>

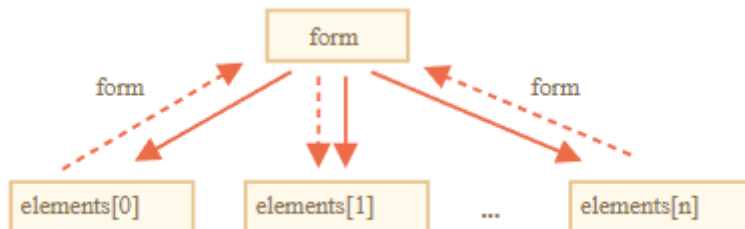
<script>
  alert(form.elements.login == form.login); // true, ведь это
  одинаковые <input>

  form.login.name = "username"; // изменяем свойство name у элемента
  input

  // form.elements обновили свои имена:
  alert(form.elements.login); // undefined
  alert(form.elements.username); // input
```

```
// теперь для прямого доступа можно использовать оба имени: новое и
старое
alert(form.username == form.login); // true
</script>
```

Для любого элемента форма доступна через `element.form`. Так что форма ссылается на все элементы, а эти элементы ссылаются на форму. Вот иллюстрация:



Пример:

```
<form id="form">
  <input type="text" name="login">
</form>

<script>
  // form -> element
  let login = form.login;

  // element -> form
  alert(login.form); // HTMLFormElement
</script>
```

Рассмотрим элементы управления, используемые в формах, обращая внимание на их особенности.

### **input и textarea**

К их значению можно получить доступ через свойство `input.value` (строка) или `input.checked` (булево значение) для чекбоксов. Вот так:

```
input.value = "Новое значение";
textarea.value = "Новый текст";

input.checked = true; // для чекбоксов и переключателей
```

Обратите внимание: хоть `<textarea>...</textarea>` и хранит значение как вложенный HTML, не следует использовать `textarea.innerHTML`. Там хранится только тот HTML, который был изначально на странице, а не текущее значение.

## select и option

Элемент `<select>` имеет 3 важных свойства:

1. `select.options` – коллекция из элементов `<option>`,
2. `select.value` – значение выбранного в данный момент `<option>`,
3. `select.selectedIndex` – номер выбранного `<option>`.

Имеется три способа задать значение для `<select>`:

1. Найти необходимый `<option>` и установить в `option.selected` значение `true`.
2. Установить в `select.value` значение нужного нам `<option>`.
3. Установить в `select.selectedIndex` номер `<option>`.

Первый способ наиболее понятный, но (2) и (3) являются более удобными при работе. Вот эти способы на примере:

```
<select id="select">
  <option value="apple">Яблоко</option>
  <option value="pear">Груша</option>
  <option value="banana">Банан</option>
</select>
```

```
<script>
  // все три строки делают одно и то же
  select.options[2].selected = true;
  select.selectedIndex = 2;
  select.value = 'banana';
</script>
```

В отличие от большинства других элементов управления, `<select multiple>` позволяет выбрать несколько вариантов. В этом случае необходимо пройти по `select.options`, чтобы получить все выбранные значения. Например так:

```
<select id="select" multiple>
  <option value="blues" selected>Блюз</option>
  <option value="rock" selected>Рок</option>
  <option value="classic">Классика</option>
</select>
```

```
<script>
  // получаем все выбранные значения из списка множественного выбора
  let selected = Array.from(select.options)
    .filter(option => option.selected)
    .map(option => option.value);

  alert(selected); // Блюз,Рок
</script>
```

Полное описание элемента `<select>` доступно в спецификации <https://html.spec.whatwg.org/multipage/forms.html#the-select-element>.

Элемент `<option>` редко используется сам по себе. В описании элемента `option` есть короткий синтаксис для создания элемента:

```
option = new Option(text, value, defaultSelected, selected);
```

Параметры:

- `text` – текст внутри,
- `value` – значение,
- `defaultSelected` – если `true`, то ставится HTML-атрибут `selected`,
- `selected` – если `true`, то элемент `<option>` будет выбранным.

Пример:

```
let option = new Option("Текст", "value");  
// создаст <option value="value">Текст</option>
```

Тот же элемент, но выбранный:

```
let option = new Option("Текст", "value", true, true);
```

Элементы `<option>` имеют дополнительные свойства:

- `selected` – выбрана ли опция,
  - `index` – номер опции среди других в списке `<select>`,
- `text` – содержимое опции (то, что видит посетитель).

## 26. Фокусировка элементов формы.

Элемент получает фокус, когда пользователь кликает по нему или использует клавишу `Tab`. Также существует HTML-атрибут `autofocus`, который устанавливает фокус на элемент, когда страница загружается. Есть и другие способы получения фокуса.

Фокусировка обычно означает: «приготовься к вводу данных на этом элементе», это хороший момент, чтобы инициализировать или загрузить что-нибудь. Момент потери фокуса («blur») может быть важнее. Это момент, когда пользователь кликает куда-то ещё или нажимает `Tab`, чтобы переключиться на следующее поле формы. Есть другие причины потери фокуса.

Потеря фокуса обычно означает «данные введены», и можно выполнить проверку введенных данных или даже отправить эти данные на сервер и так далее. В работе с событиями фокусировки есть важные особенности.

### События `focus/blur`

Событие `focus` вызывается в момент фокусировки, а `blur` – когда элемент теряет фокус. Используем их для валидации (проверки) введённых данных. В примере ниже:

- Обработчик `blur` проверяет, введён ли `email`, и если нет – показывает ошибку.

- Обработчик focus скрывает это сообщение об ошибке (в момент потери фокуса проверка повторится):

```
<style>
  .invalid { border-color: red; }
  #error { color: red }
</style>
```

Ваш email: <input type="email" id="input">

```
<div id="error"></div>
```

```
<script>
input.onblur = function() {
  if (!input.value.includes('@')) { // не email
    input.classList.add('invalid');
    error.innerHTML = 'Пожалуйста, введите правильный email.'
  }
};

input.onfocus = function() {
  if (this.classList.contains('invalid')) {
    // удаляем индикатор ошибки, т.к. пользователь хочет ввести данные
    заново
    this.classList.remove('invalid');
    error.innerHTML = "";
  }
};
</script>
```

Современный HTML позволяет делать валидацию с помощью атрибутов required, pattern и т.д. Иногда – этого достаточно. JavaScript можно использовать, когда требуется больше гибкости. А также можно отправлять изменённое значение на сервер, если оно правильное.

### Методы focus/blur

Методы elem.focus() и elem.blur() устанавливают/снимают фокус. Например, запретим посетителю переключаться с поля ввода, если введённое значение не прошло валидацию:

```
<style>
  .error {
    background: red;
```

```
}  
</style>
```

```
Ваш email: <input type="email" id="input">  
<input type="text" style="width:280px" placeholder="введите неверный  
email и кликните сюда">
```

```
<script>  
  input.onblur = function() {  
    if (!this.value.includes('@')) { // не email  
      // показать ошибку  
      this.classList.add("error");  
      // вернуть фокус обратно  
      input.focus();  
    } else {  
      this.classList.remove("error");  
    }  
  };  
</script>
```

Ваш email:  введите неверный email и кликните сюда

Ваш email:  введите неверный email и кликните сюда

Если что-нибудь ввести и нажать Tab или кликнуть в другое место, тогда onblur вернёт фокус обратно.

Нельзя отменить потерю фокуса, вызвав event.preventDefault() в обработчике onblur потому, что onblur срабатывает после потери фокуса элементом.

Потеря фокуса может произойти по множеству причин. Одна из них – когда посетитель кликает куда-то ещё. Но и JavaScript может быть причиной, например:

- alert переводит фокус на себя – элемент теряет фокус (событие blur), а когда alert закрывается – элемент получает фокус обратно (событие focus);
- если элемент удалить из DOM, фокус также будет потерян. Если элемент добавить обратно, то фокус не вернётся.

Из-за этих особенностей обработчики focus/blur могут сработать тогда, когда это не требуется.

Многие элементы по умолчанию не поддерживают фокусировку. Какие именно – зависит от браузера, но одно всегда верно: поддержка focus/blur гарантирована для элементов, с которыми посетитель может взаимодействовать: <button>, <input>, <select>, <a> и т.д. С другой стороны, элементы форматирования <div>, <span>, <table> – по умолчанию не могут

получить фокус. Метод `elem.focus()` не работает для них, и события `focus/blur` никогда не срабатывают.

Это можно изменить HTML-атрибутом `tabindex`. Цель этого атрибута – указать порядковый номер элемента, когда клавиша Tab используется для переключения между элементами. То есть: если имеется два элемента, первый имеет `tabindex="1"`, а второй `tabindex="2"`, то находясь в первом элементе и нажав Tab – фокус переместится на второй.

Есть два специальных значения:

- `tabindex="0"` делает элемент последним,
- `tabindex="-1"` значит, что Tab игнорирует этот элемент.

Любой элемент поддерживает фокусировку, если имеет `tabindex`. Например, список ниже. Кликните первый пункт в списке и нажмите Tab:

Кликните первый пункт в списке и нажмите Tab. Продолжайте следить за порядком. Обратите внимание, что много последовательных нажатий Tab могут вывести фокус из `iframe` с примером.

```
<ul>
  <li tabindex="1">Один</li>
  <li tabindex="0">Ноль</li>
  <li tabindex="2">Два</li>
  <li tabindex="-1">Минус один</li>
</ul>

<style>
  li { cursor: pointer; }
  :focus { outline: 1px dashed green; }
</style>
```

Кликните первый пункт в списке и нажмите Tab. Продолжайте следить за порядком. Обратите внимание, что много последовательных нажатий Tab могут вывести фокус из `iframe` с примером.

- Один
- Ноль
- Два
- Минус один

Порядок такой: 1 - 2 - 0 (ноль всегда последний). Обычно `<li>` не поддерживает фокусировку, но `tabindex` полностью включает её, а также события и стилизацию псевдоклассом `:focus`.

Можно добавить `tabindex` из JavaScript, используя свойство `elem.tabIndex`. Это даст тот же эффект.

### События `focusin/focusout`

События `focus` и `blur` не всплывают. Например, нельзя использовать `onfocus` на `<form>`, чтобы подсветить её:

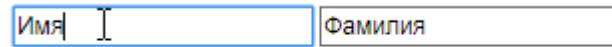
```
<!-- добавить класс при фокусировке на форме -->
<form onfocus="this.className='focused'">
  <input type="text" name="name" value="Имя">
```

```

    <input type="text" name="surname" value="Фамилия">
  </form>

<style> .focused { outline: 1px solid red; } </style>

```

Пример выше не работает, потому что когда пользователь перемещает фокус на `<input>`, событие `focus` срабатывает только на этом элементе. Это событие не всплывает. Следовательно, `form.onfocus` никогда не срабатывает.

У этой проблемы два решения. Первое: забавная особенность – `focus/blur` не всплывают, но передаются вниз на фазе перехвата. Это сработает:


```

<form id="form">
  <input type="text" name="name" value="Имя">
  <input type="text" name="surname" value="Фамилия">
</form>

<style> .focused { outline: 1px solid red; } </style>

<script>
  // установить обработчик на фазе перехвата (последний аргумент true)
  form.addEventListener("focus", () => form.classList.add('focused'),
true);
  form.addEventListener("blur", () =>
form.classList.remove('focused'), true);
</script>

```

Второе решение: события `focusin` и `focusout` – такие же, как и `focus/blur`, но они всплывают. Заметьте, что эти события должны использоваться с `elem.addEventListener`, но не с `on<event>`. Второй рабочий вариант:

```

<form id="form">
  <input type="text" name="name" value="Имя">
  <input type="text" name="surname" value="Фамилия">
</form>

<style> .focused { outline: 1px solid red; } </style>

<script>

```



```

form.addEventListener("focusin", () =>
form.classList.add('focused'));
form.addEventListener("focusout", () =>
form.classList.remove('focused'));
</script>

```

## 27.Изменение значений элемента формы. Формы: отправка, событие и метод submit.

Рассмотрим различные события, сопутствующие обновлению данных.

### Событие: change

Событие change срабатывает по окончании изменения элемента. Для текстовых `<input>` это означает, что событие происходит при потере фокуса. Пока пользователь печатает в текстовом поле в примере ниже, событие не происходит. Но когда перемещает фокус в другое место, например, нажимая на кнопку, то произойдёт событие change:

```

<input type="text" onchange="alert(this.value)">
<input type="button" value="Button">

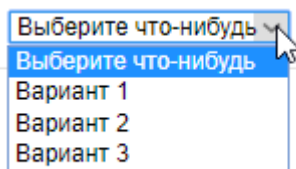
```

Для других элементов: select, input type=checkbox/radio событие запускается сразу после изменения значения:

```

<select onchange="alert(this.value)">
  <option value="">Выберите что-нибудь</option>
  <option value="1">Вариант 1</option>
  <option value="2">Вариант 2</option>
  <option value="3">Вариант 3</option>
</select>

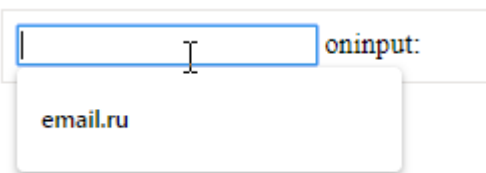
```



### Событие: input

Событие input срабатывает каждый раз при изменении значения. В отличие от событий клавиатуры, оно работает при любых изменениях значений, даже если они не связаны с клавиатурными действиями: вставка с помощью мыши или распознавание речи при диктовке текста. Например:

```
<input type="text" id="input"> oninput: <span id="result"></span>
<script>
  input.oninput = function() {
    result.innerHTML = input.value;
  };
</script>
```



Если надо обрабатывать каждое изменение в `<input>`, то это событие является лучшим выбором. С другой стороны, событие input не происходит при вводе с клавиатуры или иных действиях, если при этом не меняется значение в текстовом поле, т.е. нажатия клавиш `↵`, `⇧` и подобных при фокусе на текстовом поле не вызовут это событие.

Событие input происходит после изменения значения. Поэтому нельзя использовать `event.preventDefault()` там – будет уже слишком поздно, никакого эффекта не будет.

### События: cut, copy, paste

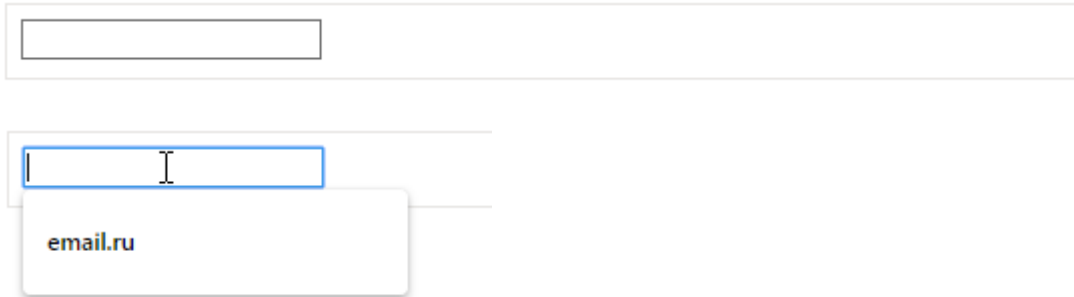
Эти события происходят при вырезании/копировании/вставке данных. Они относятся к классу `ClipboardEvent` и обеспечивают доступ к копируемым/вставляемым данным. Также можно использовать `event.preventDefault()` для предотвращения действия по умолчанию, и в итоге ничего не скопируется/не вставится. Например, код, приведённый ниже, предотвращает все подобные события и показывает, что надо вырезать/копировать/вставить:

```
<input type="text" id="input">
```

```

<script>
  input.oncut = input.oncopy = input.onpaste = function(event) {
    alert(event.type + ' - ' +
event.clipboardData.getData('text/plain'));
    return false;
  };
</script>

```



Технически, можно скопировать/вставить всё. Например, можно скопировать файл из файловой системы и вставить его. Существует список методов в спецификации для работы с различными типами данных, чтения/записи в буфер обмена. Но обратите внимание, что буфер обмена работает глобально, на уровне ОС. Большинство браузеров в целях безопасности разрешают доступ на чтение/запись в буфер обмена только в рамках определённых действий пользователя, к примеру, в обработчиках событий onclick.

Также запрещается генерировать «пользовательские» события буфера обмена при помощи dispatchEvent во всех браузерах, кроме Firefox.

## 28. JavaScript-анимация.

С помощью JavaScript-анимаций можно делать вещи, которые нельзя реализовать на CSS. Например, движение по сложному пути с временной функцией, отличной от кривой Безье, или canvas-анимации.

### Использование setInterval

Анимация реализуется через последовательность кадров, каждый из которых немного меняет HTML/CSS-свойства. Например, изменение style.left от 0px до 100px – двигает элемент. И если делать это с помощью setInterval, изменяя на 2px с небольшими интервалами времени, например, 50 раз в секунду, тогда изменения будут выглядеть плавными. Принцип такой же, как в кино: 24 кадров в секунду достаточно, чтобы создать эффект плавности. Код мог бы выглядеть так:

```

let timer = setInterval(function() {
  if (animation complete) clearInterval(timer);
  else increase style.left by 2px
}, 20); // изменять на 2px каждые 20ms, это около 50 кадров в секунду

```

Более детальная реализация этой анимации:

```
let start = Date.now(); // запомнить время начала

let timer = setInterval(function() {
  let timePassed = Date.now() - start;

  if (timePassed >= 2000) {
    clearInterval(timer); // закончить анимацию через 2 секунды
    return;
  }

  // отрисовать анимацию
  draw(timePassed);

}, 20);

// в то время как timePassed изменяется от 0 до 2000
// left изменяет значение от 0px до 400px
function draw(timePassed) {
  train.style.left = timePassed / 5 + 'px';
}
```

### Использование requestAnimationFrame

Допустим есть несколько анимаций, работающих одновременно. Если запустить их независимо с помощью `setInterval(..., 20)`, тогда браузеру будет необходимо выполнять отрисовку гораздо чаще, чем раз в 20ms.

Это происходит из-за того, что каждая анимация имеет своё собственное время старта и «каждые 20 миллисекунд» для разных анимаций – разные. Интервалы не выравнены и будет несколько независимых срабатываний в течение 20ms. Другими словами:

```
setInterval(function() {
  animate1();
  animate2();
  animate3();
}, 20)
```

меньше нагружают систему, чем три независимых функции:

```
setInterval(animate1, 20); // независимые анимации
setInterval(animate2, 20); // в разных местах кода
setInterval(animate3, 20);
```

Эти независимые анимации лучше сгруппировать вместе, тогда они будут легче для браузера, а значит – не грузить процессор и более плавно выглядеть.

Также стоит учитывать следующее: когда CPU перегружен или есть другие причины делать перерисовку реже (например, когда вкладка браузера скрыта), не следует делать её каждые 20ms.

Спецификация Animation timing описывает функцию requestAnimationFrame, которая решает все описанные проблемы и делает даже больше. Синтаксис:

```
let requestId = requestAnimationFrame(callback)
```

Такой вызов планирует запуск функции callback на ближайшее время, когда браузер сочтёт возможным осуществить анимацию. Если в callback происходит изменение элемента, тогда оно будет сгруппировано с другими requestAnimationFrame и CSS-анимациями. Таким образом браузер выполнит один геометрический пересчёт и отрисовку, вместо нескольких. Значение requestId может быть использовано для отмены анимации:

```
// отмена запланированного запуска callback  
cancelAnimationFrame(requestId);
```

Функция callback имеет один аргумент – время, прошедшее с момента начала загрузки страницы в миллисекундах. Это значение может быть получено с помощью вызова performance.now(). Как правило, callback запускается очень быстро, если только не перегружен CPU или не разряжена батарея ноутбука, или у браузера нет какой-то ещё причины замедлиться.

Код ниже показывает время между первыми 10 запусками requestAnimationFrame. Обычно оно 10-20 мс:

```
<script>  
  let prev = performance.now();  
  let times = 0;  
  
  requestAnimationFrame(function measure(time) {  
    document.body.insertAdjacentHTML("beforeEnd", Math.floor(time -  
prev) + " ");  
    prev = time;  
  
    if (times++ < 10) requestAnimationFrame(measure);  
  })  
</script>
```

### Структура анимации

Теперь можно создать более сложную функцию анимации с помощью requestAnimationFrame:

```
function animate({timing, draw, duration}) {
```

```

let start = performance.now();

requestAnimationFrame(function animate(time) {
  // timeFraction изменяется от 0 до 1
  let timeFraction = (time - start) / duration;
  if (timeFraction > 1) timeFraction = 1;

  // вычисление текущего состояния анимации
  let progress = timing(timeFraction);

  draw(progress); // отрисовать её

  if (timeFraction < 1) {
    requestAnimationFrame(animate);
  }
});
}

```

Функция `animate` имеет три аргумента, которые описывают анимацию:

1. `duration` – продолжительность анимации. Например, 1000.
2. `timing(timeFraction)` – функция расчёта времени, как CSS-свойство `transition-timing-function`, которая будет вычислять прогресс анимации (как ось  $y$  у кривой Безье) в зависимости от прошедшего времени (0 в начале, 1 в конце).

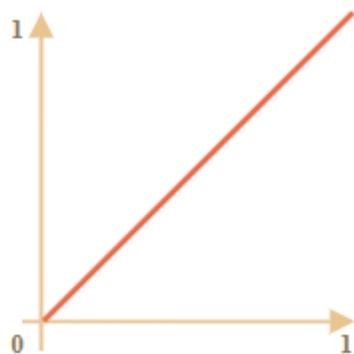
Например, линейная функция значит, что анимация идёт с одной и той же скоростью:

```

function linear(timeFraction) {
  return timeFraction;
}

```

График функции:



Это похоже на значение `linear` для `transition-timing-function`.

3. `draw(progress)` – функция отрисовки, которая получает аргументом значение прогресса анимации и отрисовывает его.

Значение `progress=0` означает что анимация находится в начале, и значение `progress=1` – в конце.

Эта та функция, которая рисует анимацию. Пример функции для сдвига элемента:

```
function draw(progress) {  
    train.style.left = progress + 'px';  
}
```

В отличие от CSS-анимаций, можно создать любую функцию расчёта времени и любую функцию отрисовки. Функция расчёта времени не будет ограничена только кривой Безье, а функция `draw` может менять не только свойства, но и создавать новые элементы (например, для создания анимации фейерверка).

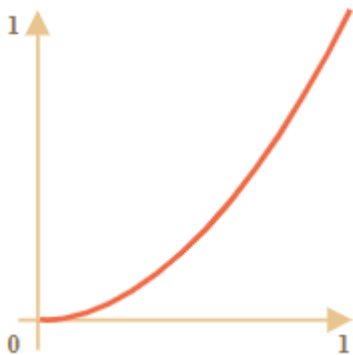
### Функции расчёта времени

Самый простой пример линейной функции расчёта времени рассматривался выше. Рассмотрим другие.

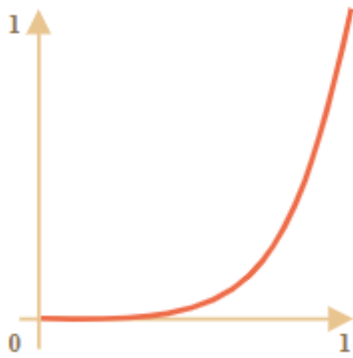
*Степень  $n$ .* Если надо ускорить анимацию, то можно возвести `progress` в степень  $n$ . Например, параболическая кривая:

```
function quad(timeFraction) {  
    return Math.pow(timeFraction, 2)  
}
```

График:



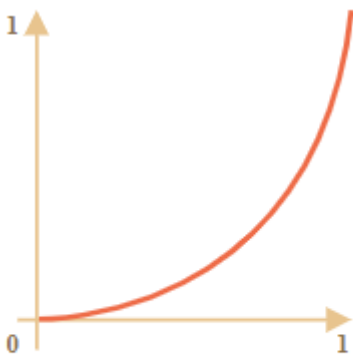
Повышение степени увеличивает скорость анимации. Вот график для функции `progress` в степени 5:



*Дуга. Функция:*

```
function circ(timeFraction) {
  return 1 - Math.sin(Math.acos(timeFraction));
}
```

График:

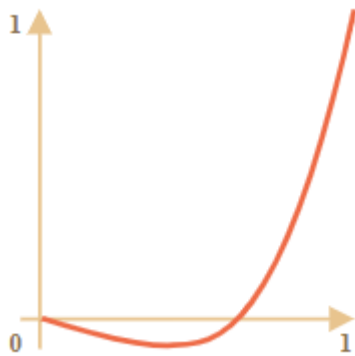


*Обратно: выстрел из лука.* Эта функция совершает «выстрел из лука». В начале «натягивается тетива», а затем «выстрел». В отличие от предыдущей функции, теперь всё зависит от дополнительного параметра  $x$  – «коэффициента эластичности». Он определяет силу «натяжения тетивы». Код:

```
function back(x, timeFraction) {
  return Math.pow(timeFraction, 2) * ((x + 1) * timeFraction - x)
}
```

График для  $x = 1.5$ :





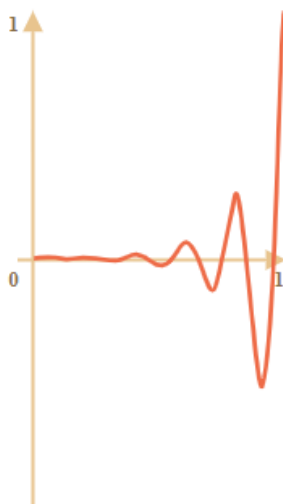
*Отскоки.* Представьте, что бросили мяч вниз. Он падает, ударяется о землю, подскакивает несколько раз и останавливается. Функции `bounce` делает то же самое, но в обратном порядке: «отскоки» начинаются сразу. Для этого заданы специальные коэффициенты:

```
function bounce(timeFraction) {
  for (let a = 0, b = 1, result; 1; a += b, b /= 2) {
    if (timeFraction >= (7 - 4 * a) / 11) {
      return -Math.pow((11 - 6 * a - 11 * timeFraction) / 4, 2) +
Math.pow(b, 2)
    }
  }
}
```

*Эластичная анимация.* Ещё одна «эластичная» функция, которая принимает дополнительный параметр  $x$  для «начального отрезка».

```
function elastic(x, timeFraction) {
  return Math.pow(2, 10 * (timeFraction - 1)) * Math.cos(20 * Math.PI
* x / 3 * timeFraction)
}
```

График для  $x = 1.5$ :



## Реверсивные функции: ease\*

Прямое использование функций расчёта времени называется «easeIn». Иногда нужно показать анимацию в обратном режиме. Преобразование функции, которое даёт такой эффект, называется «easeOut».

В режиме «easeOut» timing функции оборачиваются функцией timingEaseOut:

```
timingEaseOut(timeFraction) = 1 - timing(1 - timeFraction)
```

Другими словами, имеется функция «преобразования» – makeEaseOut, которая берет «обычную» функцию расчёта времени и возвращает обёртку над ней (преобразованный вариант):

```
function makeEaseOut(timing) {  
  return function(timeFraction) {  
    return 1 - timing(1 - timeFraction);  
  }  
}
```

Например, можно взять функцию bounce описанную выше:

```
let bounceEaseOut = makeEaseOut(bounce);
```

Таким образом, отскоки будут не в начале функции, а в конце:

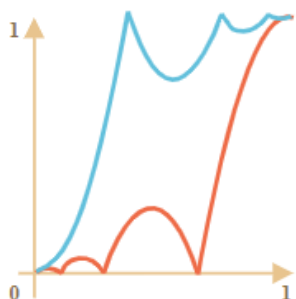
```
function makeEaseOut(timing) {  
  return function(timeFraction) {  
    return 1 - timing(1 - timeFraction);  
  }  
}  
  
function bounce(timeFraction) {  
  for (let a = 0, b = 1, result; 1; a += b, b /= 2) {  
    if (timeFraction >= (7 - 4 * a) / 11) {  
      return -Math.pow((11 - 6 * a - 11 * timeFraction) / 4, 2) +  
Math.pow(b, 2)  
    }  
  }  
}  
  
let bounceEaseOut = makeEaseOut(bounce);  
  
brick.onclick = function() {  
  animate({  
    duration: 3000,  
    timing: bounceEaseOut,  
    draw: function(progress) {  
      brick.style.left = progress * 500 + 'px';  
    }  
  })  
}
```

```

    }
  });
};

```

Ниже можно увидеть, как трансформации изменяют поведение функции:



Если раньше анимационный эффект, такой как отскоки, был в начале, то после трансформации он будет показан в конце. На графике выше красным цветом обозначена обычная функция и синим – после `easeOut`. Обычный скачок – объект сначала медленно скачет вниз, а затем резко подпрыгивает вверх. Обратный `easeOut` – объект вначале прыгает вверх, и затем скачет там.

Можно применить эффект дважды – в начале и конце анимации. Такая трансформация называется «`easeInOut`». Для функции расчёта времени, анимация будет вычисляться следующим образом:

```

if (timeFraction <= 0.5) { // первая половина анимации
  return timing(2 * timeFraction) / 2;
} else { // вторая половина анимации
  return (2 - timing(2 * (1 - timeFraction))) / 2;
}

```

Код функции-обёртки:

```

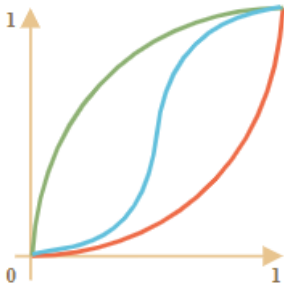
function makeEaseInOut(timing) {
  return function(timeFraction) {
    if (timeFraction < .5)
      return timing(2 * timeFraction) / 2;
    else
      return (2 - timing(2 * (1 - timeFraction))) / 2;
  }
}

bounceEaseInOut = makeEaseInOut(bounce);

```

Функция «`easeInOut`» объединяет два графика в один: `easeIn` (обычный) для первой половины анимации and `easeOut` (обратный) – для второй половины.

Разница хорошо заметна, если сравнивать графики easeIn, easeOut и easeInOut для функции circ. Красный – обычный вариант circ (easeIn), зелёный – easeOut, синий – easeInOut:



Как видно, график первой половины анимации представляет собой уменьшенный easeIn, а второй – уменьшенный easeOut. В результате, анимация начинается и заканчивается одинаковым эффектом.

Вместо передвижения элемента можно делать что-нибудь ещё. Всё, что нужно – это правильно написать функцию draw.

Вот пример «скачущей» анимации набирающегося текста:

```
Но взял он меч, и взял он щит,  
Высоких полон дум.  
В глущобу путь его лежит  
Под дерево Тумтум.
```

Запустить анимацию набора текста!

```
// style.css
```

```
textarea {  
  display: block;  
  border: 1px solid #BBB;  
  color: #444;  
  font-size: 110%;  
}
```

```
button {  
  margin-top: 10px;  
}
```

```
// index.html
```

```
<!DOCTYPE HTML>  
<html>  
  
<head>  
  <meta charset="utf-8">  
  <link rel="stylesheet" href="style.css">  
  <script src="https://js.cx/libs/animate.js"></script>  
</head>
```

```

<body>
  <textarea id="textExample" rows="5" cols="60">Но взял он меч, и взял
он щит,
Высоких полон дум.
В глушобу путь его лежит
Под дерево Тумтум.
  </textarea>

  <button onclick="animateText(textExample)">Запустить анимацию набора
текста!</button>

  <script>
    function animateText(textArea) {
      let text = textArea.value;
      let to = text.length,
          from = 0;

      animate({
        duration: 5000,
        timing: bounce,
        draw: function(progress) {
          let result = (to - from) * progress + from;
          textArea.value = text.substr(0, Math.ceil(result))
        }
      });
    }

    function bounce(timeFraction) {
      for (let a = 0, b = 1, result; 1; a += b, b /= 2) {
        if (timeFraction >= (7 - 4 * a) / 11) {
          return -Math.pow((11 - 6 * a - 11 * timeFraction) / 4, 2) +
Math.pow(b, 2)
        }
      }
    }
  </script>
</body>
</html>

```