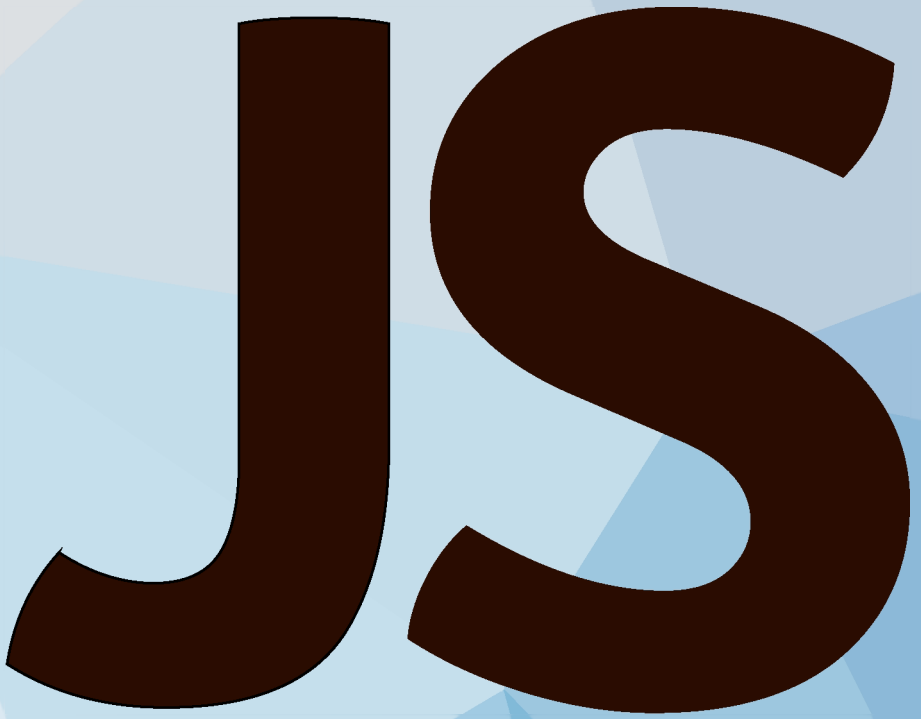
Часть 3

Дополнительные статьи



Илья Кантор

Сборка от 6 сентября 2019 г.

Последняя версия учебника находится на сайте https://learn.javascript.ru.

Мы постоянно работаем над улучшением учебника. При обнаружении ошибок пишите о них на нашем баг-трекере.

- Пользовательские элементы (Custom Elements)
- Shadow DOM
- Элемент "template"
- Слоты теневого DOM, композиция
- Настройка стилей теневого DOM
- Теневой DOM и события

- Регулярные выражения
  - Введение: шаблоны и флаги
  - Символьные классы
  - Юникод: флаг "u" и класс \p{...}
  - Якоря: начало строки ^ и конец $
  - Многострочный режим якорей ^ $, флаг "m"
  - Граница слова: \b
  - Экранирование, специальные символы
  - Наборы и диапазоны [...]
  - Квантификаторы +, *, ? и {n}
  - Жадные и ленивые квантификаторы
  - Скобочные группы
  - Обратные ссылки в шаблоне: \N и \k<имя>
  - Альтернация (или) |
  - Опережающие и ретроспективные проверки
  - Катастрофический возврат
  - Поиск на заданной позиции, флаг "y"
  - Методы RegExp и String
- CSS для JavaScript-разработчика
  - О чём пойдёт речь
  - Единицы измерения: px, em, rem и другие
  - Все значения свойства display
  - Свойство float
  - Свойство position
  - Центрирование горизонтальное и вертикальное
  - Свойства font-size и line-height
  - Свойство white-space
  - Свойство outline
  - Свойство box-sizing
  - Свойство margin

- Лишнее место под IMG
- Свойство overflow
- Особенности свойства height в %
- Знаете ли вы селекторы?
- CSS-спрайты
- Правила форматирования CSS

Фреймы и окна

Открытие окон и методы window

Всплывающее окно («попап» – от англ. Popup window) – один из древнейших способов показать пользователю ещё один документ.

Достаточно запустить:

```
window.open('https://javascript.info/')
```

… и откроется новое окно с указанным URL. Большинство современных браузеров по умолчанию будут открывать новую вкладку вместо отдельного окна.

Попапы существуют с доисторических времён. Они были придуманы для отображения нового контента поверх открытого главного окна. Но с тех пор появились другие способы сделать это: JavaScript может загрузить содержимое вызовом fetch и показать его в тут же созданном `<div>`, так что попапы используются не каждый день.

Кроме того, попапы не очень хороши для мобильных устройств, которые не умеют показывать несколько окон одновременно.

Однако, для некоторых задач попапы ещё используются, например для OAuth-авторизации (вход через Google/Facebook/…), так как:

1. Попап – это отдельное окно со своим JavaScript-окружением. Так что открытие попапа со стороннего, не доверенного сайта вполне безопасно
2. Открыть попап очень просто.
3. Попап может производить навигацию (менять URL) и отсылать сообщения в основное окно.

Блокировка попапов

В прошлом злонамеренные сайты заваливали посетителей всплывающими окнами. Такие страницы могли открывать сотни попапов с рекламой. Поэтому теперь большинство браузеров пытаются заблокировать всплывающие окна, чтобы защитить пользователя.

Всплывающее окно блокируется в том случае, если вызов window.open произошёл не в результате действия посетителя (например, события `onclick`).

Например:

```
// попап заблокирован
window.open('https://javascript.info');

// попап будет показан
button.onclick = () => {
  window.open('https://javascript.info');
};
```

Таким образом браузеры могут защитить пользователя от появления нежелательных попапов, при этом не отключая попапы полностью.

Что, если попап должен открываться в результате `onclick`, но не сразу, а только после выполнения `setTimeout`? Здесь все не так-то просто.

Запустим код:

```
// откроется через 3 секунды
setTimeout(() => window.open('http://google.com'), 3000);
```

Попап откроется в Chrome, но будет заблокирован в Firefox.

Но если мы уменьшим тайм-аут до одной секунды, то попап откроется и в Firefox:

```
// откроется через 1 секунду
setTimeout(() => window.open('http://google.com'), 1000);
```

Мы получили два разных результата из-за того, что Firefox «допускает» таймаут в 2000 мс или менее, но все, что свыше этого – не вызывает его доверия, т.к. предполагается, что в таком случае открытие окна происходит без ведома пользователя. Именно поэтому попап из первого примера будет заблокирован, а из второго – нет.

Полный синтаксис window.open

Синтаксис открытия нового окна: `window.open(url, name, params)`:

url

URL для загрузки в новом окне.

name

Имя нового окна. У каждого окна есть свойство `window.name`, в котором можно задавать, какое окно использовать для попапа. Таким образом, если уже

существует окно с заданным именем – указанный в параметрах URL откроется в нем, в противном случае откроется новое окно.

params

Строка параметров для нового окна. Содержит настройки, разделённые запятыми. Важно помнить, что в данной строке не должно быть пробелов. Например `width:200,height=100`.

Параметры в строке `params`:

- Позиция окна:
  - `left/top` (числа) – координаты верхнего левого угла нового окна на экране. Существует ограничение: новое окно не может быть позиционировано вне видимой области экрана.
  - `width/height` (числа) – ширина и высота нового окна. Существуют ограничения на минимальную высоту и ширину, которые делают невозможным создание невидимого окна.
- Панели окна:
  - `menubar` (yes/no) – позволяет отобразить или скрыть меню браузера в новом окне.
  - `toolbar` (yes/no) – позволяет отобразить или скрыть панель навигации браузера (кнопки вперёд, назад, перезагрузки страницы) нового окна.
  - `location` (yes/no) – позволяет отобразить или скрыть адресную строку нового окна. Firefox и IE не позволяют скрывать эту панель по умолчанию.
  - `status` (yes/no) – позволяет отобразить или скрыть строку состояния. Как и с адресной строкой, большинство браузеров будут принудительно показывать её.
  - `resizable` (yes/no) – позволяет отключить возможность изменения размера нового окна. Не рекомендуется.
  - `scrollbars` (yes/no) – позволяет отключить полосы прокрутки для нового окна. Не рекомендуется.

Помимо этого существует некоторое количество не кроссбраузерных значений, которые обычно не используются. Найти примеры таких свойств можно по ссылке .

Пример: минималистичное окно

Давайте откроем окно с минимальным набором настроек, просто чтобы посмотреть, какие из них браузер позволит отключить:

```
let params = `scrollbars=no,resizable=no,status=no,location=no,toolbar=no,menubar=n
width=0,height=0,left=-1000,top=-1000`;

open('/', 'test', params);
```

В этом примере большинство настроек заблокированы и само окно находится за пределами видимой области экрана. Посмотрим, что получится в результате. Большинство браузеров «исправит» странные значения – как, например, нулевые `width/height` и отрицательные `left/top`. Например, Chrome установит высоту и ширину такого окна равными высоте и ширине экрана, так что попап будет занимать весь экран.

Давайте исправим значения и зададим нормальные координаты (`left` и `top`) и значения размеров окна (`width` и `height`):

```
let params = `scrollbars=no,resizable=no,status=no,location=no,toolbar=no,menubar=n
width=600,height=300,left=100,top=100`;

open('/', 'test', params);
```

Большинство браузеров выведет окно с заданными нами настройками.

Правила для опущенных параметров:

- Если третий аргумент при вызове `open` отсутствует или он пустой, будут использованы настройки окна по умолчанию.
- Если строка параметров передана, но некоторые параметры yes/no пропущены, то считается, что указано `no`, так что соответствующие возможности будут отключены, если на это нет ограничений со стороны браузера. Поэтому при задании параметров убедитесь, что вы явно указали все необходимые yes.
- Если координаты `left/top` не заданы, браузер попытается открыть новое окно рядом с предыдущим открытым окном.
- Если не заданы размеры окна `width/height`, браузер откроет новое окно с теми же размерами, что и предыдущее открытое окно.

Доступ к попапу из основного окна

Вызов `open` возвращает ссылку на новое окно. Эта ссылка может быть использована для управления свойствами окна, например, изменения положения и др.

Например, здесь мы генерируем содержимое попапа из JavaScript:

```
let newWin = window.open("about:blank", "hello", "width=200,height=200");

newWin.document.write("Hello, world!");
```

А здесь содержимое окна модифицируется после загрузки:

```
let newWindow = open('/', 'example', 'width=300,height=300')
newWindow.focus();

alert(newWindow.location.href); // (*) about:blank, загрузка ещё не началась

newWindow.onload = function() {
  let html = `<div style="font-size:30px">Добро пожаловать!</div>`;
  newWindow.document.body.insertAdjacentHTML('afterbegin', html);
};
```

Обратите внимание: сразу после `window.open` новое окно ещё не загружено. Это демонстрируется в строке `(*)`. Так что нужно ждать `onload`, чтобы его изменить. Или же поставить обработчик `DOMContentLoaded` на `newWin.document`.

Политика одного источника

Окна имеют свободный доступ к содержимому друг другу только если они с одного источника (у них совпадают домен, протокол и порт (protocol://domain:port).

Иначе, например, если основное окно с `site.com`, а попап с `gmail.com`, это невозможно по соображениям пользовательской безопасности. Детали см. в главе Общение между окнами.

Доступ к открывшему окну из попапа

Попап также может обратиться к открывшему его окну по ссылке `window.opener`. Она равна `null` для всех окон, кроме попапов.

Если вы запустите код ниже, то он заменит содержимое открывшего (текущего) окна на «Тест»:

```
let newWin = window.open("about:blank", "hello", "width=200,height=200");

newWin.document.write(
  "<script>window.opener.document.body.innerHTML = 'Тест'<\/script>"
);
```

Так что связь между окнами двусторонняя: главное окно и попап имеют ссылки друг на друга.

Закрытие попапа

Чтобы закрыть окно: `win.close()`

Для проверки, закрыто ли окно: `win.closed`.

Технически метод `close()` доступен для любого окна, но `window.close()` будет игнорироваться большинством браузеров, если `window` не было создано с помощью `window.open()`. Так что он сработает только для попапов.

Если окно закрыто, то его свойство `closed` имеет значение `true`. Таким образом можно легко проверить, закрыт ли попап (или главное окно) или все ещё открыт. Пользователь может закрыть его в любой момент, и наш код должен учитывать эту возможность.

Этот код откроет и затем закроет окно:

```
let newWindow = open('/', 'example', 'width=300,height=300');

newWindow.onload = function() {
  newWindow.close();
  alert(newWindow.closed); // true
};
```

Прокрутка и изменение размеров

Методы для передвижения и изменения размеров окна:

`win.moveBy(x,y)`

Переместить окно относительно текущей позиции на `x` пикселей вправо и `y` пикселей вниз. Допустимы отрицательные значения (для перемещения окна влево и вверх).

`win.moveTo(x,y)`

Переместить окно на координаты экрана `(x,y)`.

`win.resizeBy(width,height)`

Изменить размер окна на указанные значения `width/height` относительно текущего размера. Допустимы отрицательные значения.

`win.resizeTo(width,height)`

Изменить размер окна до указанных значений.

Также существует событие `window.onresize`.

> ⚠ **Только попапы**
>
> Чтобы предотвратить возможные злоупотребления, браузер обычно блокирует эти методы. Они гарантированно работают только с попапами, которые мы открыли сами и у которых нет дополнительных вкладок.

> ⚠ **Нельзя свернуть/развернуть окно**
>
> Методами JavaScript нельзя свернуть или развернуть («максимизировать») окно на весь экран. За это отвечают функции уровня операционной системы, и они скрыты от фронтенд-разработчиков.
>
> Методы перемещения и изменения размера окна не работают для свернутых и развёрнутых на весь экран окон.

Прокрутка окна

Мы уже говорили о прокрутке окна в главе Размеры и прокрутка окна.

`win.scrollBy(x,y)`

Прокрутить окно на `x` пикселей вправо и `y` пикселей вниз относительно текущей прокрутки. Допустимы отрицательные значения.

`win.scrollTo(x,y)`

Прокрутить окно до заданных координат `(x,y)`.

`elem.scrollIntoView(top = true)`

Прокрутить окно так, чтобы `elem` для `elem.scrollIntoView(false)` появился вверху (по умолчанию) или внизу.

Также существует событие `window.onscroll`.

Установка и потеря фокуса

Теоретически, установить попап в фокус можно с помощью метода `window.focus()`, а убрать из фокуса – с помощью `window.blur()`. Также существуют события `focus/blur`, которые позволяют отследить, когда фокус переводится на какое-то другое окно.

Раньше на «плохих» сайтах эти методы могли становиться средством манипуляции. Например:

```
window.onblur = () => window.focus();
```

Когда пользователь пытается перевести фокус на другое окно, этот код возвращает фокус назад. Таким образом, фокус как бы «блокируется» в попапе, который не нужен пользователю.

Из-за этого в браузерах и появились ограничения, которые препятствуют такого рода поведению фокуса. Эти ограничения нужны для защиты пользователя от назойливой рекламы и «плохих» страниц, и их работа различается в зависимости от конкретного браузера.

Например, мобильный браузер обычно полностью игнорирует такие вызовы метода `window.focus()`. Также фокусировка не работает, когда попап открыт в отдельной вкладке (в отличие от открытия в отдельном окне).

Но все-таки иногда методы фокусировки бывают полезны. Например:

- Когда мы открываем попап, может быть хорошей идеей запустить для него `newWindow.focus()`. Для некоторых комбинаций браузера и операционной системы это устранит неоднозначность – заметит ли пользователь это новое окно.
- Если нужно отследить, когда посетитель использует веб-приложение, можно отслеживать `window.onfocus/onblur`. Это позволит ставить на паузу и продолжать выполнение анимаций и других интерактивных действий на странице. При этом важно помнить, что `blur` означает, что окно больше не в фокусе, но пользователь может по-прежнему видеть его.

Итого

Всплывающие окна используются нечасто. Ведь загрузить новую информацию можно динамически, а показать – в элементе `<div>`, расположенным над страницей (`z-index`). Ещё одна альтернатива – тег `<iframe>`.

Если мы открываем попап, хорошей практикой будет предупредить пользователя об этом. Иконка открывающегося окошка на ссылке поможет посетителю понять, что происходит и не потерять оба окна из поля зрения.

- Новое окно можно открыть с помощью вызова `open(url, name, params)`. Этот метод возвращает ссылку на это новое окно.
- По умолчанию браузеры блокируют вызовы `open`, выполненные не в результате действий пользователя. Обычно браузеры показывают

предупреждение, так что пользователь все-таки может разрешить вызов этого метода.
- Вместо попапа открывается вкладка, если в вызове `open` не указаны его размеры.
- У попапа есть доступ к породившему его окну через свойство `window.opener`.
- Если основное окно и попап имеют один домен и протокол, то они свободно могут читать и изменять друг друга. В противном случае, они могут только изменять положение друг друга и взаимодействовать с помощью сообщений.

Чтобы закрыть попап: метод `close()`. Также попап может закрыть и пользователь (как и любое другое окно). После закрытия окна свойство `window.closed` имеет значение `true`.

- Методы `focus()` и `blur()` позволяют установить или убрать фокус с попапа. Но работают не всегда.
- События `focus` и `blur` позволяют отследить получение и потерю фокуса новым окном. Но пожалуйста, не забывайте, что окно может остаться видимым и после `blur`.

Общение между окнами

Политика «Одинакового источника» (Same Origin) ограничивает доступ окон и фреймов друг к другу.

Идея заключается в том, что если у пользователя открыто две страницы: `john-smith.com` и `gmail.com`, то у скрипта со страницы `john-smith.com` не будет возможности прочитать письма из `gmail.com`. Таким образом, задача политики «Одинакового источника» – защитить данные пользователя от возможной кражи.

Политика "Одинакового источника"

Два URL имеют «одинаковый источник» в том случае, если они имеют совпадающие протокол, домен и порт.

Эти URL имеют одинаковый источник:

- `http://site.com`
- `http://site.com/`
- `http://site.com/my/page.html`

А эти – разные источники:

- `http://www.site.com` (другой домен: `www.` важен)
- `http://site.org` (другой домен: `.org` важен)
- `https://site.com` (другой протокол: `https`)
- `http://site.com:8080` (другой порт: `8080`)

Политика «Одинакового источника» говорит, что:

- если у нас есть ссылка на другой объект `window`, например, на всплывающее окно, созданное с помощью `window.open` или на `window` из `<iframe>` и у этого окна тот же источник, то к нему будет полный доступ.
- в противном случае, если у него другой источник, мы не сможем обращаться к его переменным, объекту `document` и так далее. Единственное исключение – объект `location`: его можно изменять (таким образом перенаправляя пользователя). Но нельзя читать `location` (нельзя узнать, где находится пользователь, чтобы не было никаких утечек информации).

Доступ к содержимому ифрейма

Внутри `<iframe>` находится по сути отдельное окно, то у окно, с собственными объектами `document` и `window`.

Мы можем обращаться к ним, используя свойства:

- `iframe.contentWindow` ссылка на объект `window` внутри `<iframe>`.
- `iframe.contentDocument` – ссылка на объект `document` внутри `<iframe>`, короткая запись для `iframe.contentWindow.document`.

Когда мы обращаемся к встроенному в ифрейм окну, браузер проверяет, имеет ли ифрейм тот же источник. Если это не так, тогда доступ будет запрещён (разрешена лишь запись в `location`, это исключение).

Для примера давайте попробуем чтение и запись в ифрейм с другим источником:

```
<iframe src="https://example.com" id="iframe"></iframe>

<script>
  iframe.onload = function() {
    // можно получить ссылку на внутренний window
    let iframeWindow = iframe.contentWindow; // ОК
    try {
      // ...но не на document внутри него
      let doc = iframe.contentDocument; // ОШИБКА
    } catch(e) {
      alert(e); // Security Error
    }

    // также мы не можем прочитать URL страницы в ифрейме
```

```
    try {
      // Нельзя читать из объекта Location
      let href = iframe.contentWindow.location.href; // ОШИБКА
    } catch(e) {
      alert(e); // Security Error
    }

    // ...но можно писать в него (и загрузить что-то другое в ифрейм)!
    iframe.contentWindow.location = '/'; // ОК

    iframe.onload = null; // уберём обработчик, чтобы не срабатывал после изменения
  };
</script>
```

Код выше выведет ошибку для любых операций, кроме:

- Получения ссылки на внутренний объект `window` из `iframe.contentWindow`
- Изменения `location`.

С другой стороны, если у ифрейма тот же источник, то с ним можно делать всё, что угодно:

```
<!-- ифрейм с того же сайта -->
<iframe src="/" id="iframe"></iframe>

<script>
  iframe.onload = function() {
    // делаем с ним что угодно    iframe.contentDocument.body.prepend("Привет, мир!
  };
</script>
```

`iframe.onload` и `iframe.contentWindow.onload`

Событие `iframe.onload` – по сути то же, что и `iframe.contentWindow.onload`. Оно сработает, когда встроенное окно полностью загрузится со всеми ресурсами.

…Но `iframe.onload` всегда доступно извне ифрейма, в то время как доступ к `iframe.contentWindow.onload` разрешён только из окна с тем же источником.

Окна на поддоменах: document.domain

По определению, если у двух URL разный домен, то у них разный источник.

Но если в окнах открыты страницы с поддоменов одного домена 2-го уровня, например `john.site.com`, `peter.site.com` и `site.com` (так что их общий домен `site.com`), то можно заставить браузер игнорировать это отличие. Так что браузер сможет считать их пришедшими с одного источника при проверке возможности доступа друг к другу.

Для этого в каждом таком окне нужно запустить:

```
document.domain = 'site.com';
```

После этого они смогут взаимодействовать без ограничений. Ещё раз заметим, что это доступно только для страниц с одинаковым доменом второго уровня.

Ифрейм: подождите документ

Когда ифрейм – с того же источника, мы имеем доступ к документу в нём. Но есть подвох. Не связанный с кросс-доменными особенностями, но достаточно важный, чтобы о нём знать.

Когда ифрейм создан, в нём сразу есть документ. Но этот документ – другой, не тот, который в него будет загружен!

Так что если мы тут же сделаем что-то с этим документом, то наши изменения, скорее всего, пропадут.

Вот, взгляните:

```
<iframe src="/" id="iframe"></iframe>

<script>
  let oldDoc = iframe.contentDocument;
  iframe.onload = function() {
    let newDoc = iframe.contentDocument;
    // загруженный document - не тот, который был в iframe при создании изначально!
    alert(oldDoc == newDoc); // false
  };
</script>
```

Нам не следует работать с документом ещё не загруженного ифрейма, так как это не тот документ. Если мы поставим на него обработчики событий – они будут проигнорированы.

Как поймать момент, когда появится правильный документ?

Можно проверять через `setInterval`:

```
<iframe src="/" id="iframe"></iframe>
```

```
<script>
  let oldDoc = iframe.contentDocument;

  // каждый 100 мс проверяем, не изменился ли документ
  let timer = setInterval(() => {
    let newDoc = iframe.contentDocument;
    if (newDoc == oldDoc) return;

    alert("New document is here!");

    clearInterval(timer); // отключим setInterval, он нам больше не нужен
  }, 100);
</script>
```

Коллекция window.frames

Другой способ получить объект `window` из `<iframe>` – забрать его из именованной коллекции `window.frames`:

- По номеру: `window.frames[0]` – объект `window` для первого фрейма в документе.
- По имени: `window.frames.iframeName` – объект `window` для фрейма со свойством `name="iframeName"`.

Например:

```
<iframe src="/" style="height:80px" name="win" id="iframe"></iframe>

<script>
  alert(iframe.contentWindow == frames[0]); // true
  alert(iframe.contentWindow == frames.win); // true
</script>
```

Ифрейм может иметь другие ифреймы внутри. Таким образом, объекты `window` создают иерархию.

Навигация по ним выглядит так:

- `window.frames` – коллекция «дочерних» `window` (для вложенных фреймов).
- `window.parent` – ссылка на «родительский» (внешний) `window`.
- `window.top` – ссылка на самого верхнего родителя.

Например:

```
window.frames[0].parent === window; // true
```

Можно использовать свойство `top`, чтобы проверять, открыт ли текущий документ внутри ифрейма или нет:

```
if (window == top) { // текущий window == window.top?
  alert('Скрипт находится в самом верхнем объекте window, не во фрейме');
} else {
  alert('Скрипт запущен во фрейме!');
}
```

Атрибут ифрейма sandbox

Атрибут `sandbox` позволяет наложить ограничения на действия внутри `<iframe>`, чтобы предотвратить выполнение ненадёжного кода. Атрибут помещает ифрейм в «песочницу», отмечая его как имеющий другой источник и/или накладывая на него дополнительные ограничения.

Существует список «по умолчанию» ограничений, которые накладываются на `<iframe sandbox src="...">`. Их можно уменьшить, если указать в атрибуте список исключений (специальными ключевыми словами), которые не нужно применять, например: `<iframe sandbox="allow-forms allow-popups">`.

Другими словами, если у атрибута `"sandbox"` нет значения, то браузер применяет максимум ограничений, но через пробел можно указать те из них, которые мы не хотим применять.

Вот список ограничений:

`allow-same-origin`

`"sandbox"` принудительно устанавливает «другой источник» для ифрейма. Другими словами, он заставляет браузер воспринимать `iframe`, как пришедший из другого источника, даже если `src` содержит тот же сайт. Со всеми сопутствующими ограничениями для скриптов. Эта опция отключает это ограничение.

`allow-top-navigation`

Позволяет ифрейму менять `parent.location`.

`allow-forms`

Позволяет отправлять формы из ифрейма.

`allow-scripts`

Позволяет запускать скрипты из ифрейма.

`allow-popups`

Позволяет открывать всплывающие окна из ифрейма с помощью `window.open`.

Больше опций можно найти в справочнике.

Пример ниже демонстрирует ифрейм, помещённый в песочницу со стандартным набором ограничений: `<iframe sandbox src="...">`. На странице содержится JavaScript и форма.

Обратите внимание, что ничего не работает. Таким образом, набор ограничений по умолчанию очень строгий:

https://plnkr.co/edit/3i34dqLwib37dThcY9x9?p=preview

> **На заметку:**
>
> Атрибут `"sandbox"` создан только для того, чтобы добавлять ограничения. Он не может удалять их. В частности, он не может ослабить ограничения, накладываемые браузером на ифрейм, приходящий с другого источника.

Обмен сообщениями между окнами

Интерфейс `postMessage` позволяет окнам общаться между собой независимо от их происхождения.

Это способ обойти политику «Одинакового источника». Он позволяет обмениваться информацией, скажем `john-smith.com` и `gmail.com`, но только в том случае, если оба сайта согласны и вызывают соответствующие JavaScript-функции. Это делает общение безопасным для пользователя.

Интерфейс имеет две части.

postMessage

Окно, которое хочет отправить сообщение, должно вызвать метод postMessage окна получателя. Другими словами, если мы хотим отправить сообщение в окно `win`, тогда нам следует вызвать `win.postMessage(data, targetOrigin)`.

Аргументы:

`data`

Данные для отправки. Может быть любым объектом, данные клонируются с использованием «алгоритма структурированного клонирования». IE

поддерживает только строки, поэтому мы должны использовать метод `JSON.stringify` на сложных объектах, чтобы поддержать этот браузер.

`targetOrigin`

Определяет источник для окна-получателя, только окно с данного источника имеет право получить сообщение.

Указание `targetOrigin` является мерой безопасности. Как мы помним, если окно (получатель) происходит из другого источника, мы из окна-отправителя не можем прочитать его `location`. Таким образом, мы не можем быть уверены, какой сайт открыт в заданном окне прямо сейчас: пользователь мог перейти куда-то, окно-отправитель не может это знать.

Если указать `targetOrigin`, то мы можем быть уверены, что окно получит данные только в том случае, если в нём правильный сайт. Особенно это важно, если данные конфиденциальные.

Например, здесь `win` получит сообщения только в том случае, если в нём открыт документ из источника `http://example.com`:

```
<iframe src="http://example.com" name="example">

<script>
  let win = window.frames.example;

  win.postMessage("message", "http://example.com");
</script>
```

Если мы не хотим проверять, то в `targetOrigin` можно указать `*`.

```
<iframe src="http://example.com" name="example">

<script>
  let win = window.frames.example;

  win.postMessage("message", "*");
</script>
```

onmessage

Чтобы получать сообщения, окно-получатель должно иметь обработчик события `message` (сообщение). Оно срабатывает, когда был вызван метод `postMessage` (и проверка `targetOrigin` пройдена успешно).

Объект события имеет специфичные свойства:

`data`

Данные из `postMessage`.

`origin`

Источник отправителя, например, `http://javascript.info`.

`source`

Ссылка на окно-отправитель. Можно сразу отправить что-то в ответ, вызвав `source.postMessage(...)`.

Чтобы добавить обработчик, следует использовать метод `addEventListener`, короткий синтаксис `window.onmessage` не работает.

Вот пример:

```
window.addEventListener("message", function(event) {
  if (event.origin != 'http://javascript.info') {
    // что-то пришло с неизвестного домена. Давайте проигнорируем это
    return;
  }

  alert( "received: " + event.data );

  // can message back using event.source.postMessage(...)
});
```

Полный пример:

https://plnkr.co/edit/G3Tj8rNmhfzrLbIblIQl?p=preview

Без задержек

Между `postMessage` и событием `message` не существует задержки. Событие происходит синхронно, быстрее, чем `setTimeout(...,0)`.

Итого

Чтобы вызвать метод или получить содержимое из другого окна, нам по-первых необходимо иметь ссылку на него.

Для всплывающих окон (попапов) доступны ссылки в обе стороны:

- При открытии окна: `window.open` открывает новое окно и возвращает ссылку на него,
- Изнутри открытого окна: `window.opener` – ссылка на открывающее окно.

Для ифреймов мы можем иметь доступ к родителям/потомкам, используя:

- `window.frames` – коллекция объектов `window` вложенных ифреймов,
- `window.parent`, `window.top` – это ссылки на родительское окно и окно самого верхнего уровня,
- `iframe.contentWindow` – это объект `window` внутри тега `<iframe>`.

Если окна имеют одинаковый источник (протокол, домен, порт), то они могут делать друг с другом всё, что угодно.

В противном случае возможны только следующие действия:

- Изменение свойства location другого окна (доступ только на запись).
- Отправить туда сообщение.

Исключения:

- Окна, которые имеют общий домен второго уровня: `a.site.com` и `b.site.com`. Установка свойства `document.domain='site.com'` в обоих окнах переведёт их в состояние «Одинакового источника».
- Если у ифрейма установлен атрибут `sandbox`, это принудительно переведёт окна в состояние «разных источников», если не установить в атрибут значение `allow-same-origin`. Это можно использовать для запуска ненадёжного кода в ифрейме с того же сайта.

Метод `postMessage` позволяет общаться двум окнам с любыми источниками:

1. Отправитель вызывает `targetWin.postMessage(data, targetOrigin)`.
2. Если `targetOrigin` не `'*'`, тогда браузер проверяет имеет ли `targetWin` источник `targetOrigin`.
3. Если это так, тогда `targetWin` вызывает событие `message` со специальными свойствами:
   - `origin` – источник окна отправителя (например, `http://my.site.com`)
   - `source` – ссылка на окно отправитель.
   - `data` – данные, может быть объектом везде, кроме IE (в IE только строки).

   В окне-получателе следует добавить обработчик для этого события с помощью метода `addEventListener`.

Атака типа clickjacking

Атака типа clickjacking (англ. «захват клика») позволяет вредоносной странице кликнуть по сайту-жертве от имени посетителя.

Многие сайты были взломаны подобным способом, включая Twitter, Facebook, Paypal и другие. Все они, конечно же, сейчас защищены.

Идея

Идея этой атаки очень проста.

Вот как clickjacking-атака была проведена на Facebook:

1. Посетителя заманивают на вредоносную страницу (неважно как).
2. На странице есть ссылка, которая выглядит безобидно (например, «Разбогатей прямо сейчас» или «Нажми здесь, это очень смешно»).
3. Поверх этой ссылки вредоносная страница размещает прозрачный `<iframe>` с `src` с сайта facebook.com таким образом, что кнопка «like» находится прямо над этой ссылкой. Обычно это делается с помощью `z-index` в CSS.
4. При попытке клика на эту ссылку посетитель на самом деле нажимает на кнопку.

Демонстрация

Вот как выглядит вредоносная страница. Для наглядности `<iframe>` полупрозрачный (на реальных вредоносных страницах он полностью прозрачен):

```
<style>
iframe { /* ифрейм с сайта-жертвы */
  width: 400px;
  height: 100px;
  position: absolute;
  top:0; left:-20px;
  opacity: 0.5; /* в реальности opacity:0 */
  z-index: 1;
}
</style>

<div>Нажми, чтобы разбогатеть:</div>

<!-- Url с сайта-жертвы -->
<iframe src="/clickjacking/facebook.html"></iframe>

<button>Нажмите сюда!</button>

<div>...И всё будет супер (у меня, хакера)!</div>
```

Полная демонстрация атаки:

https://plnkr.co/edit/pclwSvIMry1YrCYN45Zq?p=preview

Здесь у нас есть полупрозрачный `<iframe src="facebook.html">`, и в примере мы видим его висящим поверх кнопки. Клик на кнопку фактически кликает на ифрейм, но этого не видно пользователю, потому что ифрейм прозрачный.

В результате, если пользователь авторизован на сайте Facebook («Запомнить меня» обычно активировано), то он добавляет «лайк». В Twitter это будет кнопка «читать», и т.п.

Вот тот же пример, но более приближенный к реальности с `opacity:0` для `<iframe>`:

https://plnkr.co/edit/WTqBDanbrXPXpYdUoEAp?p=preview

Всё, что нам необходимо для атаки — это расположить `<iframe>` на вредоносной странице так, чтобы кнопка находилась прямо над ссылкой. Так что пользователь, кликающий по ссылке, на самом деле будет нажимать на кнопку в `<iframe>`. Обычно это можно сделать с помощью CSS-позиционирования.

Clickjacking-атака для кликов мыши, а не для клавиатуры

Эта атака срабатывает только на действия мыши (или аналогичные, вроде нажатия пальцем на мобильном устройстве).

Клавиатурный ввод гораздо сложнее перенаправить. Технически, если у нас есть текстовое поле для взлома, мы можем расположить ифрейм таким образом, чтобы текстовые поля перекрывали друг друга. Тогда посетитель при попытке сфокусироваться на текстовом поле, которое он видит на странице, фактически будет фокусироваться на текстовом поле внутри ифрейм.

Но есть одна проблема. Всё, что посетитель печатает, будет скрыто, потому что ифрейм не виден.

Обычно люди перестают печатать, когда не видят на экране новых символов.

Примеры слабой защиты

Самым старым вариантом защиты является код JavaScript, запрещающий открытие страницы во фрейме (это называют «framebusting»).

Выглядит он вот так:

```
if (top != window) {
  top.location = window.location;
}
```

В этом случае, если окно обнаруживает, что оно открыто во фрейме, оно автоматически располагает себя сверху.

Этот метод не является надёжной защитой, поскольку появилось множество способов его обойти. Рассмотрим некоторые из них.

Блокировка top-навигации

Мы можем заблокировать переход, вызванный сменой `top.location` в обработчике события beforeunload.

Внешняя страница (принадлежащая хакеру) устанавливает обработчик на это событие, отменяющий его, например, такой:

```
window.onbeforeunload = function() {
  return false;
};
```

Когда iframe пытается изменить `top.location`, посетитель увидит сообщение с вопросом действительно ли он хочет покинуть эту страницу. В большинстве случаев посетитель ответит отрицательно, поскольку он не знает об ифрейме: всё, что он видит – это верхнюю страницу, которую нет причин покидать. Поэтому `top.location` не изменится!

В действии:

https://plnkr.co/edit/Yruccyj5f63m0aegBdiH?p=preview

Атрибут «sandbox»

Одним из действий, которые можно ограничить атрибутом `sandbox`, является навигация. Соответственно ифрейм внутри sandbox не изменит `top.location`.

Поэтому мы можем добавить ифрейм с `sandbox="allow-scripts allow-forms"`. Это снимет некоторые ограничения, разрешая при этом использование скриптов и форм. Но мы опускаем `allow-top-navigation`, чтобы изменение `top.location` было запрещено.

Вот код этого примера:

```
<iframe sandbox="allow-scripts allow-forms" src="facebook.html"></iframe>
```

Есть и другие способы обойти эту простую защиту.

Заголовок X-Frame-Options

Заголовок `X-Frame-Options` со стороны сервера может разрешать или запрещать отображение страницы внутри фрейма.

Это должен быть именно HTTP-заголовок: браузер проигнорирует его, если найдёт в HTML-теге `<meta>`. Поэтому при `<meta http-equiv="X-Frame-Options"...>` ничего не произойдёт.

Заголовок может иметь 3 значения:

`DENY`

Никогда не показывать страницу внутри фрейма.

`SAMEORIGIN`

Разрешить открытие страницы внутри фрейма только в том случае, если родительский документ имеет тот же источник.

`ALLOW-FROM domain`

Разрешить открытие страницы внутри фрейма только в том случае, если родительский документ находится на указанном в заголовке домене.

Например, Twitter использует `X-Frame-Options: SAMEORIGIN`.

Отображение с ограниченными возможностями

У заголовка `X-Frame-Options` есть побочный эффект. Другие сайты не смогут отобразить нашу страницу во фрейме, даже если у них будут на то веские причины.

Так что есть другие решения… Например, мы можем «накрыть» страницу блоком `<div>` со стилями `height: 100%; width: 100%;`, чтобы он перехватывал все клики. Этот `<div>` будем убирать, если `window == top` или если мы поймём, что защита нам не нужна.

Примерно так:

```
<style>
  #protector {
    height: 100%;
    width: 100%;
    position: absolute;
    left: 0;
    top: 0;
```

```
    z-index: 99999999;
  }
</style>

<div id="protector">
  <a href="/" target="_blank">Перейти к сайту</a>
</div>

<script>
  // Здесь будет отображаться ошибка, если верхнее окно имеет другое происхождение
  // а здесь будет код, если всё в порядке
  if (top.document.domain == document.domain) {
    protector.remove();
  }
</script>
```

Демонстрация:

https://plnkr.co/edit/UiCEcyufsWocXTRBtBpY?p=preview

Атрибут cookie: samesite

Атрибут `samesite` также может помочь избежать clickjacking-атаки.

Файл куки с таким атрибутом отправляется на сайт только в том случае, если он открыт напрямую, не через фрейм или каким-либо другим способом. Подробно об этом – в главе Куки, document.cookie.

Если сайт, такой как Facebook, при установке авторизующего куки ставит атрибут `samesite`:

```
Set-Cookie: authorization=secret; samesite
```

… Тогда такие куки не будут отправляться, когда Facebook будет открыт в ифрейме с другого сайта. Так что атака не удастся.

Атрибут `samesite` не играет никакой роли, если куки не используются. Так что другие веб-сайты смогут отображать публичные, не требующие авторизации, страницы в ифрейме.

Однако, это даёт возможность в некоторых ситуациях осуществить clickjacking-атаку, например, на сайт для анонимных опросов, который предотвращает повторное голосование пользователя путём проверки IP-адреса. Он останется уязвимым к атаке, потому что не аутентифицирует пользователей с помощью куки.

Итого

Атака clickjacking – это способ хитростью «заставить» пользователей кликнуть на сайте-жертве, без понимания, что происходит. Она опасна, если по клику могут быть произведены важные действия.

Хакер может разместить ссылку на свою вредоносную страницу в сообщении или найти другие способы, как заманить пользователей. Вариантов множество.

С одной стороны — эта атака «неглубокая», ведь хакер перехватывает только один клик. Но с другой стороны, если хакер знает, что после этого клика появятся другие элементы управления, то он может хитростью заставить пользователя кликнуть на них.

Этот вид атаки довольно опасен, ведь при разработке интерфейсов мы не предполагаем, что хакер может кликнуть от имени пользователя. Поэтому уязвимости могут быть обнаружены в совершенно неожиданных местах.

- Для защиты от этой атаки рекомендуется использовать `X-Frame-Options: SAMEORIGIN` на страницах или даже целиком сайтах, которые не предназначены для просмотра во фрейме.
- Или, если мы хотим разрешить отображение страницы во фрейме и при этом оставаться в безопасности, то можно использовать перекрывающий блок `<div>`.

Бинарные данные и файлы

Работа с бинарными данными и файлами в JavaScript.

ArrayBuffer, бинарные массивы

В сфере веб-разработки мы встречаемся с бинарными данными чаще всего тогда, когда требуется выполнить какие-то действия над файлами (создать, загрузить или скачать). Другим типичным примером такой встречи является обработка изображений.

Всё это возможно осуществить на JavaScript. Более того, операции над бинарными данными являются высокопроизводительными.

Обилие классов для работы с бинарными данными может немного запутать. Вот некоторые из них:

- `ArrayBuffer`, `Uint8Array`, `DataView`, `Blob`, `File` и т.д.

Работа с бинарными данными в JavaScript реализована нестандартно по сравнению с другими языками программирования. Но когда мы в этом разберёмся, то всё окажется весьма просто.

Базовый объект для работы с бинарными данными имеет тип `ArrayBuffer` и представляет собой ссылку на непрерывную область

памяти фиксированной длины.

Вот так мы можем создать его экземпляр:

```
let buffer = new ArrayBuffer(16); // создаётся буфер длиной 16 байт
alert(buffer.byteLength); // 16
```

Инструкция выше выделяет непрерывную область памяти размером 16 байт и заполняет её нулями.

> ⚠️ **`ArrayBuffer` – это не массив!**
>
> Давайте внесём ясность, чтобы не запутаться. `ArrayBuffer` не имеет ничего общего с `Array`:
>
> - его длина фиксирована, мы не можем увеличивать или уменьшать её.
> - `ArrayBuffer` занимает ровно столько места в памяти, сколько указывается при создании.
> - Для доступа к отдельным байтам нужен вспомогательный объект-представление, `buffer[index]` не сработает.

`ArrayBuffer` – это область памяти. Что там хранится? Этой информации нет. Просто необработанный («сырой») массив байтов.

Для работы с `ArrayBuffer` нам нужен специальный объект, реализующий «представление» данных.

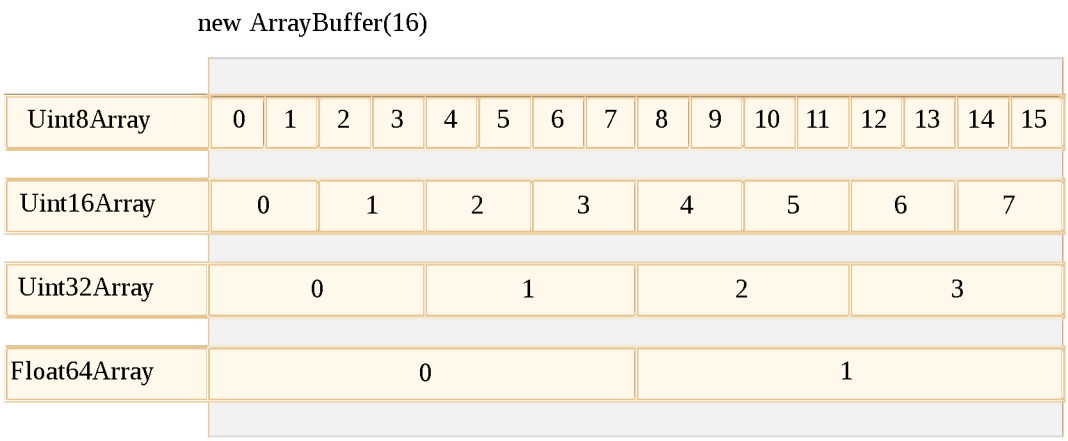
Такие объекты не хранят какое-то собственное содержимое. Они интерпретируют бинарные данные, хранящиеся в `ArrayBuffer`.

Например:

- **`Uint8Array`** – представляет каждый байт в `ArrayBuffer` как отдельное число; возможные значения находятся в промежутке от 0 до 255 (в байте 8 бит, отсюда такой набор). Такое значение называется «8-битное целое без знака».
- **`Uint16Array`** – представляет каждые 2 байта в `ArrayBuffer` как целое число; возможные значения находятся в промежутке от 0 до 65535. Такое значение называется «16-битное целое без знака».
- **`Uint32Array`** – представляет каждые 4 байта в `ArrayBuffer` как целое число; возможные значения находятся в промежутке от 0 до 4294967295. Такое значение называется «32-битное целое без знака».
- **`Float64Array`** – представляет каждые 8 байт в `ArrayBuffer` как число с плавающей точкой; возможные значения находятся в промежутке между

`5.0x10^-324` и `1.8x10^308`.

Таким образом, бинарные данные из `ArrayBuffer` размером 16 байт могут быть представлены как 16 чисел маленькой разрядности или как 8 чисел большей разрядности (по 2 байта каждое), или как 4 числа ещё большей разрядности (по 4 байта каждое), или как 2 числа с плавающей точкой высокой точности (по 8 байт каждое).



`ArrayBuffer` – это корневой объект, основа всего, необработанные бинарные данные.

Но если мы собираемся что-то записать в него или пройтись по его содержимому, да и вообще для любых действий мы должны использовать какой-то объект-представление («view»), например:

```
let buffer = new ArrayBuffer(16); // создаётся буфер длиной 16 байт

let view = new Uint32Array(buffer); // интерпретируем содержимое как последовательн

alert(Uint32Array.BYTES_PER_ELEMENT); // 4 байта на каждое целое число

alert(view.length); // 4, именно столько чисел сейчас хранится в буфере
alert(view.byteLength); // 16, размер содержимого в байтах

// давайте запишем какое-нибудь значение
view[0] = 123456;

// теперь пройдёмся по всем значениям
for(let num of view) {
  alert(num); // 123456, потом 0, 0, 0 (всего 4 значения)
}
```

TypedArray

Общий термин для всех таких представлений (`Uint8Array`, `Uint32Array` и т.д.) – это TypedArray, типизированный массив. У них имеется набор одинаковых свойств и методов.

Они уже намного больше напоминают обычные массивы: элементы проиндексированы, и возможно осуществить обход содержимого.

Конструкторы типизированных массивов (будь то `Int8Array` или `Float64Array`, без разницы) ведут себя по-разному в зависимости от типа передаваемого им аргумента.

Есть 5 вариантов создания типизированных массивов:

```
new TypedArray(buffer, [byteOffset], [length]);
new TypedArray(object);
new TypedArray(typedArray);
new TypedArray(length);
new TypedArray();
```

1. Если передан аргумент типа `ArrayBuffer`, то создаётся объект-представление для него. Мы уже использовали этот синтаксис ранее.

   Дополнительно можно указать аргументы `byteOffset` (0 по умолчанию) и `length` (до конца буфера по умолчанию), тогда представление будет покрывать только часть данных в `buffer`.

2. Если в качестве аргумента передан `Array` или какой-нибудь псевдомассив, то будет создан типизированный массив такой же длины и с тем же содержимым.

   Мы можем использовать эту возможность, чтобы заполнить типизированный массив начальными данными:

   ```
   let arr = new Uint8Array([0, 1, 2, 3]);
   alert( arr.length ); // 4, создан бинарный массив той же длины
   alert( arr[1] ); // 1, заполнен 4-мя байтами с указанными значениями
   ```

3. Если в конструктор передан другой объект типа `TypedArray`, то делается то же самое: создаётся типизированный массив с такой же длиной и в него копируется содержимое. При необходимости значения будут приведены к новому типу.

   ```
   let arr16 = new Uint16Array([1, 1000]);
   let arr8 = new Uint8Array(arr16);
   alert( arr8[0] ); // 1
   alert( arr8[1] ); // 232, потому что 1000 не помещается в 8 бит (разъяснения буду
   ```

4. Если передано число `length` – будет создан типизированный массив, содержащий именно столько элементов. Размер нового массива в байтах будет равен числу элементов `length`, умноженному на размер одного элемента `TypedArray.BYTES_PER_ELEMENT`:

```
let arr = new Uint16Array(4); // создаём типизированный массив для 4 целых 16-бит
alert( Uint16Array.BYTES_PER_ELEMENT ); // 2 байта на число
alert( arr.byteLength ); // 8 (размер массива в байтах)
```

5. При вызове без аргументов будет создан пустой типизированный массив.

Как видим, можно создавать типизированные массивы `TypedArray` напрямую, не передавая в конструктор объект типа `ArrayBuffer`. Но представления не могут существовать сами по себе без двоичных данных, так что на самом деле объект `ArrayBuffer` создаётся автоматически во всех случаях, кроме первого, когда он явно передан в конструктор представления.

Для доступа к `ArrayBuffer` есть следующие свойства:

- `arr.buffer` – ссылка на объект `ArrayBuffer`.
- `arr.byteLength` – размер содержимого `ArrayBuffer` в байтах.

Таким образом, мы всегда можем перейти от одного представления к другому:

```
let arr8 = new Uint8Array([0, 1, 2, 3]);

// другое представление на тех же данных
let arr16 = new Uint16Array(arr8.buffer);
```

Список типизированных массивов:

- `Uint8Array`, `Uint16Array`, `Uint32Array` – целые беззнаковые числа по 8, 16 и 32 бита соответственно.
  - `Uint8ClampedArray` – 8-битные беззнаковые целые, обрезаются по верхней и нижней границе при присвоении (об этом ниже).
- `Int8Array`, `Int16Array`, `Int32Array` – целые числа со знаком (могут быть отрицательными).
- `Float32Array`, `Float64Array` – 32- и 64-битные числа со знаком и плавающей точкой.

⚠ **Не существует примитивных типов данных `int8` и т.д.**

Обратите внимание: несмотря на названия вроде `Int8Array`, в JavaScript нет примитивных типов данных `int` или `int8`.

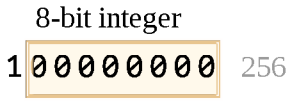
Это логично, потому что `Int8Array` – это не массив отдельных значений, а представление, основанное на бинарных данных из объекта типа `ArrayBuffer`.

Что будет, если выйти за пределы допустимых значений?

Что если мы попытаемся записать в типизированный массив значение, которое превышает допустимое для данного массива? Ошибки не будет. Лишние биты просто будут отброшены.

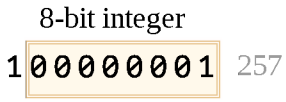
Например, давайте попытаемся записать число 256 в объект типа `Uint8Array`. В двоичном формате 256 представляется как `100000000` (9 бит), но `Uint8Array` предоставляет только 8 бит для значений. Это определяет диапазон допустимых значений от 0 до 255.

Если наше число больше, то только 8 младших битов (самые правые) будут записаны, а лишние – отбросятся:



Таким образом, вместо 256 запишется 0.

Число 257 в двоичном формате выглядит как `100000001` (9 бит), но принимаются во внимание только 8 самых правых битов, так что в объект будет записана единичка:



Другими словами, записываются только значения по модулю 2^8.

Вот демо:

```
let uint8array = new Uint8Array(16);

let num = 256;
alert(num.toString(2)); // 100000000 (в двоичном формате)

uint8array[0] = 256;
uint8array[1] = 257;

alert(uint8array[0]); // 0
alert(uint8array[1]); // 1
```

`Uint8ClampedArray`, упомянутый ранее, ведёт себя по-другому в данных обстоятельствах. В него записываются значения 255 для чисел, которые больше 255, и 0 для отрицательных чисел. Такое поведение полезно в некоторых ситуациях, например при обработке изображений.

Методы TypedArray

Типизированные массивы `TypedArray`, за некоторыми заметными исключениями, имеют те же методы, что и массивы `Array`.

Мы можем обходить их, вызывать `map`, `slice`, `find`, `reduce` и т.д.

Однако, есть некоторые вещи, которые нельзя осуществить:

- Нет метода `splice` – мы не можем удалять значения, потому что типизированные массивы – это всего лишь представления данных из буфера, а буфер – это непрерывная область памяти фиксированной длины. Мы можем только записать 0 вместо значения.
- Нет метода `concat`.

Но зато есть два дополнительных метода:

- `arr.set(fromArr, [offset])` копирует все элементы из `fromArr` в `arr`, начиная с позиции `offset` (0 по умолчанию).
- `arr.subarray([begin, end])` создаёт новое представление того же типа для данных, начиная с позиции `begin` до `end` (не включая). Это похоже на метод `slice` (который также поддерживается), но при этом ничего не копируется – просто создаётся новое представление, чтобы совершать какие-то операции над указанными данными.

Эти методы позволяют нам копировать типизированные массивы, смешивать их, создавать новые на основе существующих и т.д.

DataView

DataView – это специальное супергибкое нетипизированное представление данных из `ArrayBuffer`. Оно позволяет обращаться к данным на любой позиции и в любом формате.

- В случае типизированных массивов конструктор строго задаёт формат данных. Весь массив состоит из однотипных значений. Доступ к i-ому элементу можно получить как `arr[i]`.
- В случае `DataView` доступ к данным осуществляется посредством методов типа `.getUint8(i)` или `.getUint16(i)`. Мы выбираем формат данных в момент обращения к ним, а не в момент их создания.

Синтаксис:

```
new DataView(buffer, [byteOffset], [byteLength])
```

- **`buffer`** – ссылка на бинарные данные `ArrayBuffer`. В отличие от типизированных массивов, `DataView` не создаёт буфер автоматически. Нам нужно заранее подготовить его самим.
- **`byteOffset`** – начальная позиция данных для представления (по умолчанию 0).
- **`byteLength`** – длина данных (в байтах), используемых в представлении (по умолчанию – до конца `buffer`).

Например, извлечём числа в разных форматах из одного и того же буфера двоичных данных:

```
// бинарный массив из 4х байт, каждый имеет максимальное значение 255
let buffer = new Uint8Array([255, 255, 255, 255]).buffer;

let dataView = new DataView(buffer);

// получим 8-битное число на позиции 0
alert( dataView.getUint8(0) ); // 255

// а сейчас мы получим 16-битное число на той же позиции 0, оно состоит из 2-х байт
alert( dataView.getUint16(0) ); // 65535 (максимальное 16-битное беззнаковое целое)

// получим 32-битное число на позиции 0
alert( dataView.getUint32(0) ); // 4294967295 (максимальное 32-битное беззнаковое ц

dataView.setUint32(0, 0); // при установке 4-байтового числа в 0, во все его 4 байт
```

Представление `DataView` отлично подходит, когда мы храним данные разного формата в одном буфере. Например, мы храним последовательность

пар, первое значение пары 16-битное целое, а второе – 32-битное с плавающей точкой. `DataView` позволяет легко получить доступ к обоим.

Итого

`ArrayBuffer` – это корневой объект, ссылка на непрерывную область памяти фиксированной длины.

Чтобы работать с объектами типа `ArrayBuffer`, нам нужно представление («view»).

- Это может быть типизированный массив `TypedArray`:
  - `Uint8Array`, `Uint16Array`, `Uint32Array` – для беззнаковых целых по 8, 16 и 32 бита соответственно.
  - `Uint8ClampedArray` – для 8-битных беззнаковых целых, которые обрезаются по верхней и нижней границе при присвоении.
  - `Int8Array`, `Int16Array`, `Int32Array` – для знаковых целых чисел (могут быть отрицательными).
  - `Float32Array`, `Float64Array` – для 32- и 64-битных знаковых чисел с плавающей точкой.
- Или `DataView` – представление, использующее отдельные методы, чтобы уточнить формат данных при обращении, например, `getUint8(offset)`.

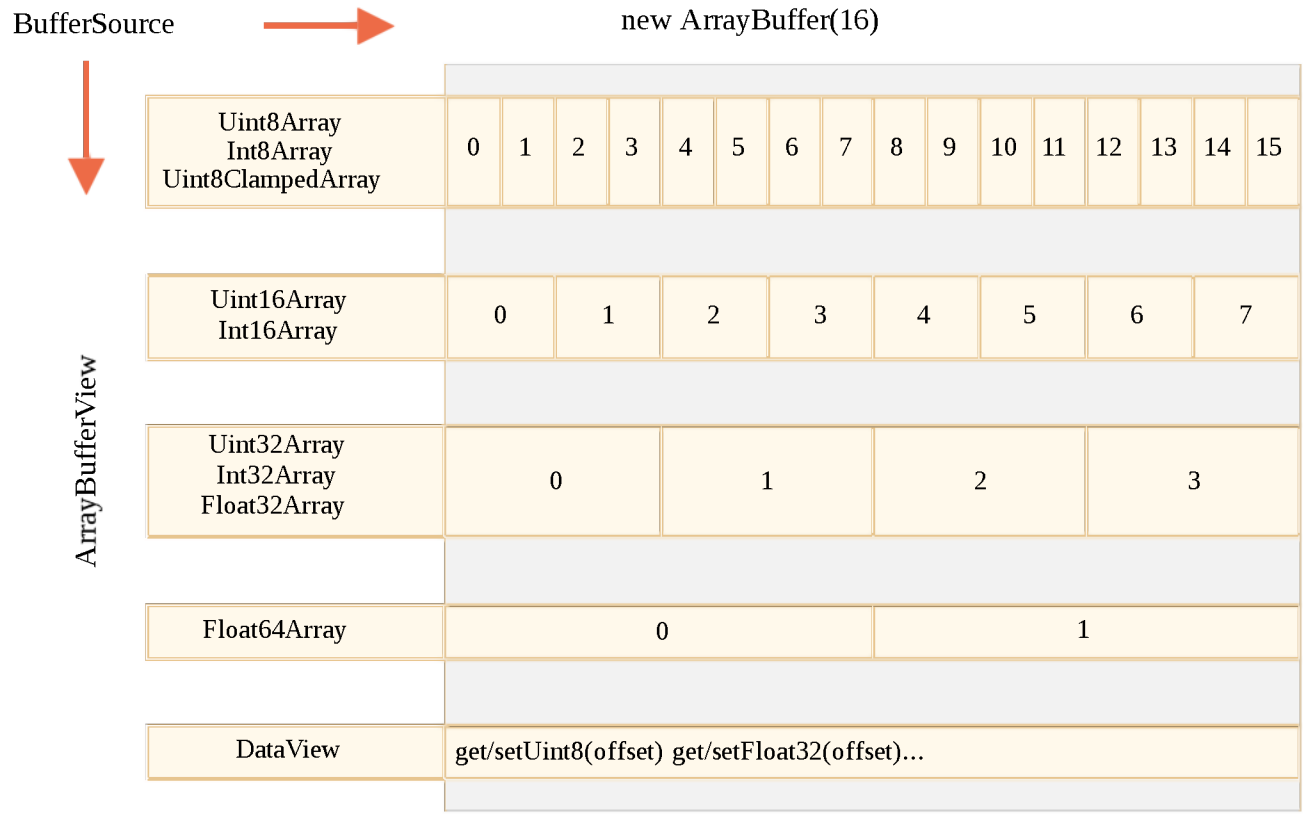
Обычно мы создаём и работаем с типизированными массивами, оставляя `ArrayBuffer` «под капотом». Но мы можем в любой момент получить к нему доступ с помощью `.buffer` и при необходимости создать другое представление.

Существуют ещё 2 дополнительных термина, которые используются в описаниях методов, работающих с бинарными данными:

- `ArrayBufferView` – это общее название для представлений всех типов.
- `BufferSource` – это общее название для `ArrayBuffer` или `ArrayBufferView`.

Мы встретимся с ними в следующих главах. `BufferSource` встречается очень часто и означает «бинарные данные в любом виде» – `ArrayBuffer` или его представление.

Вот шпаргалка:



Задачи

Соедините типизированные массивы

Дан массив из типизированных массивов `Uint8Array`. Напишите функцию `concat(arrays)`, которая объединяет эти массивы в один типизированный массив и возвращает его.

Открыть песочницу с тестами для задачи.

К решению

TextDecoder и TextEncoder

Что если бинарные данные фактически являются строкой? Например, мы получили файл с текстовыми данными.

Встроенный объект TextDecoder позволяет декодировать данные из бинарного буфера в обычную строку.

Для этого прежде всего нам нужно создать сам декодер:

```
let decoder = new TextDecoder([label], [options]);
```

- **`label`** – тип кодировки, `utf-8` используется по умолчанию, но также поддерживаются `big5`, `windows-1251` и многие другие.
- **`options`** – объект с дополнительными настройками:
  - **`fatal`** – boolean, если значение `true`, тогда генерируется ошибка для невалидных (не декодируемых) символов, в ином случае (по умолчанию) они заменяются символом `\uFFFD`.
  - **`ignoreBOM`** – boolean, если значение `true`, тогда игнорируется BOM (дополнительный признак, определяющий порядок следования байтов), что необходимо крайне редко.

…и после использовать его метод decode:

```
let str = decoder.decode([input], [options]);
```

- **`input`** – бинарный буфер (`BufferSource`) для декодирования.
- **`options`** – объект с дополнительными настройками:
  - **`stream`** – true для декодирования потока данных, при этом `decoder` вызывается вновь и вновь для каждого следующего фрагмента данных. В этом случае многобайтовый символ может иногда быть разделён и попасть в разные фрагменты данных. Это опция указывает `TextDecoder` запомнить символ, на котором остановился процесс, и декодировать его со следующим фрагментом.

Например:

```
let uint8Array = new Uint8Array([72, 101, 108, 108, 111]);

alert( new TextDecoder().decode(uint8Array) ); // Hello
```

```
let uint8Array = new Uint8Array([228, 189, 160, 229, 165, 189]);

alert( new TextDecoder().decode(uint8Array) ); // 你好
```

Мы можем декодировать часть бинарного массива, создав подмассив:

```
let uint8Array = new Uint8Array([0, 72, 101, 108, 108, 111, 0]);
```

```
// Возьмём строку из середины массива
// Также обратите внимание, что это создаёт только новое представление без копирова
// Изменения в содержимом созданного подмассива повлияют на исходный массив и наобо
let binaryString = uint8Array.subarray(1, -1);

alert( new TextDecoder().decode(binaryString) ); // Hello
```

TextEncoder

TextEncoder поступает наоборот – кодирует строку в бинарный массив.

Имеет следующий синтаксис:

```
let encoder = new TextEncoder();
```

Поддерживается только кодировка «utf-8».

Кодировщик имеет следующие два метода:

- **`encode(str)`** – возвращает бинарный массив `Uint8Array`, содержащий закодированную строку.
- **`encodeInto(str, destination)`** – кодирует строку (`str`) и помещает её в `destination`, который должен быть экземпляром `Uint8Array`.

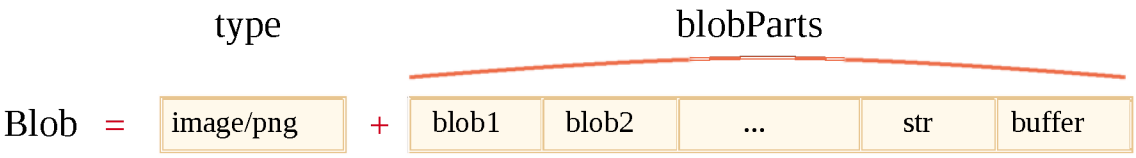
```
let encoder = new TextEncoder();

let uint8Array = encoder.encode("Hello");
alert(uint8Array); // 72,101,108,108,111
```

Blob

`ArrayBuffer` и бинарные массивы являются частью ECMA-стандарта и, соответственно, частью JavaScript.

Кроме того, в браузере имеются дополнительные высокоуровневые объекты, описанные в File API.

Объект `Blob` состоит из необязательной строки `type` (обычно MIME-тип) и `blobParts` – последовательности других объектов `Blob`, строк и `BufferSource`.



Благодаря `type` мы можем загружать и скачивать Blob-объекты, где `type` естественно становится `Content-Type` в сетевых запросах.

Конструктор имеет следующий синтаксис:

```
new Blob(blobParts, options);
```

- **`blobParts`** – массив значений `Blob`/`BufferSource`/`String`.
- **`options`** – необязательный объект с дополнительными настройками:
  - **`type`** – тип объекта, обычно MIME-тип, например. `image/png`,
  - **`endings`** – если указан, то окончания строк создаваемого `Blob` будут изменены в соответствии с текущей операционной системой (`\r\n` или `\n`). По умолчанию `"transparent"` (ничего не делать), но также может быть `"native"` (изменять).

Например:

```
// создадим Blob из строки
let blob = new Blob(["<html>…</html>"], {type: 'text/html'});
// обратите внимание: первый аргумент должен быть массивом [...]
```

```
// создадим Blob из типизированного массива и строк
let hello = new Uint8Array([72, 101, 108, 108, 111]); // "hello" в бинарной форме

let blob = new Blob([hello, ' ', 'world'], {type: 'text/plain'});
```

Мы можем получить срез Blob, используя:

```
blob.slice([byteStart], [byteEnd], [contentType]);
```

- **`byteStart`** – стартовая позиция байта, по умолчанию 0.
- **`byteEnd`** – последний байт, по умолчанию до конца.
- **`contentType`** – тип `type` создаваемого Blob-объекта, по умолчанию такой же, как и исходный.

Аргументы – как в `array.slice`, отрицательные числа также разрешены.

> **Blob не изменяем (immutable)**
>
> Мы не можем изменять данные напрямую в Blob, но мы можем делать срезы и создавать новый Blob на их основе, объединять несколько объектов в новый и так далее.
>
> Это поведение аналогично JavaScript-строке: мы не можем изменить символы в строке, но мы можем создать новую исправленную строку на базе имеющейся.

Blob как URL

Blob может быть использован как URL для `<a>`, `<img>` или других тегов, для показа содержимого.

Давайте начнём с простого примера. При клике на ссылку мы загружаем динамически генерируемый Blob с `hello world` содержимым как файл:

```
<!-- download атрибут указывает браузеру делать загрузку вместо навигации -->
<a download="hello.txt" href='#' id="link">Загрузить</a>

<script>
let blob = new Blob(["Hello, world!"], {type: 'text/plain'});

link.href = URL.createObjectURL(blob);
</script>
```

Мы также можем создать ссылку динамически, используя только JavaScript, и эмулировать на ней клик, используя `link.click()`, тогда загрузка начнётся автоматически.

Далее простой пример создания «на лету» и загрузки Blob-объекта, без использования HTML:

```
let link = document.createElement('a');
link.download = 'hello.txt';

let blob = new Blob(['Hello, world!'], {type: 'text/plain'});

link.href = URL.createObjectURL(blob);

link.click();

URL.revokeObjectURL(link.href);
```

`URL.createObjectURL` берёт Blob и создаёт уникальный URL для него в формате `blob:<origin>/<uuid>`.

Вот как выглядит сгенерированный URL:

```
blob:https://javascript.info/1e67e00e-860d-40a5-89ae-6ab0cbee6273
```

Браузер для каждого URL, сгенерированного через `URL.createObjectURL`, сохраняет внутреннее соответствие URL → `Blob`. Так образом, такие URL короткие, но дают доступ к большому объекту `Blob`.

Сгенерированный url действителен, только пока текущий документ открыт. Это позволяет ссылаться на сгенерированный в нём Blob в `<img>`, `<a>` или в любом другом объекте, где ожидается url в качестве одного из параметров.

В данном случае возможен побочный эффект. Пока в карте соответствия существует ссылка на Blob, он находится в памяти. Браузер не может освободить память, занятую Blob-объектом.

Ссылка в карте соответствия автоматически удаляется при выгрузке документа, после этого также освобождается память. Но если приложение имеет длительный жизненный цикл, это может произойти не скоро. Таким образом, если мы создадим URL для Blob, он будет висеть в памяти, даже если в нём нет больше необходимости.

`URL.revokeObjectURL(url)` удаляет внутреннюю ссылку на объект, что позволяет удалить его (если нет другой ссылки) сборщику мусора, и память будет освобождена.

В последнем примере мы использовали Blob только единожды, для мгновенной загрузки, после мы сразу же вызвали `URL.revokeObjectURL(link.href)`.

В предыдущем примере с кликабельной HTML-ссылкой мы не вызывали `URL.revokeObjectURL(link.href)`, потому что это сделало бы ссылку недействительной. После удаления внутренней ссылки на `Blob`, URL больше не будет работать.

Blob to base64

Альтернатива `URL.createObjectURL` – конвертация Blob-объекта в строку с кодировкой base64.

Эта кодировка представляет двоичные данные в виде строки с безопасными для чтения символами в ASCII-кодах от 0 до 64. И что более важно – мы можем использовать эту кодировку для «data-urls».

data url имеет форму `data:[<mediatype>][;base64],<data>`. Мы можем использовать такой url где угодно наряду с «обычным» url.

Например, смайлик:

```
<img src="
```

Браузер декодирует строку и показывает смайлик:

Для трансформации Blob в base64 мы будем использовать встроенный в браузер объект типа `FileReader`. Он может читать данные из Blob в множестве форматов. В следующей главе мы рассмотрим это более глубоко.

Вот пример загрузки Blob при помощи base64:

```
let link = document.createElement('a');
link.download = 'hello.txt';

let blob = new Blob(['Hello, world!'], {type: 'text/plain'});

let reader = new FileReader();
reader.readAsDataURL(blob); // конвертирует Blob в base64 и вызывает onload

reader.onload = function() {
  link.href = reader.result; // url с данными
  link.click();
};
```

Оба варианта могут быть использованы для создания URL с Blob. Но обычно `URL.createObjectURL(blob)` является более быстрым и безопасным.

URL.createObjectURL(blob)

- Нужно отзывать объект для освобождения памяти.
- Прямой доступ к Blob, без «кодирования/декодирования».

Blob to data url

- Нет необходимости что-либо отзывать.
- Потеря производительности и памяти при декодировании больших Blob-объектов.

Изображение в Blob

Мы можем создать Blob для изображения, части изображения или даже создать скриншот страницы. Что удобно для последующей загрузки куда-либо.

Операции с изображениями выполняются через элемент `<canvas>`:

1. Для отрисовки изображения (или его части) на холсте (canvas) используется canvas.drawImage .
2. Вызов canvas-метода .toBlob(callback, format, quality) создаёт Blob и вызывает функцию `callback` при завершении.

В примере ниже изображение просто копируется, но мы можем взять его часть или трансформировать его на canvas перед созданием Blob:

```
// берём любое изображение
let img = document.querySelector('img');

// создаём <canvas> того же размера
let canvas = document.createElement('canvas');
canvas.width = img.clientWidth;
canvas.height = img.clientHeight;

let context = canvas.getContext('2d');

// копируем изображение в  canvas (метод позволяет вырезать часть изображения)
context.drawImage(img, 0, 0);
// мы можем вращать изображение при помощи context.rotate() и делать множество друг

// toBlob является асинхронной операцией, для которой callback-функция вызывается п
canvas.toBlob(function(blob) {
  // после того, как Blob создан, загружаем его
  let link = document.createElement('a');
  link.download = 'example.png';

  link.href = URL.createObjectURL(blob);
  link.click();

  // удаляем внутреннюю ссылку на Blob, что позволит браузеру очистить память
  URL.revokeObjectURL(link.href);
}, 'image/png');
```

Или если вы предпочитаете `async/await` вместо колбэка:

```
let blob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png'));
```

Для создания скриншота страницы мы можем использовать такую библиотеку, как https://github.com/niklasvh/html2canvas . Всё, что она делает, это просто

проходит страницу и отрисовывает её в `<canvas>`. После этого мы может получить Blob одним из вышеуказанных способов.

Из Blob в ArrayBuffer

Конструктор `Blob` позволяет создать Blob-объект практически из чего угодно, включая `BufferSource`.

Но если нам нужна производительная низкоуровневая обработка, мы можем использовать `ArrayBuffer` из `FileReader`:

```
// получаем arrayBuffer из Blob
let fileReader = new FileReader();

fileReader.readAsArrayBuffer(blob);

fileReader.onload = function(event) {
  let arrayBuffer = fileReader.result;
};
```

Итого

В то время как `ArrayBuffer`, `Uint8Array` и другие `BufferSource` являются «бинарными данными», Blob представляет собой «бинарные данные с типом».

Это делает Blob удобным для операций загрузки/выгрузки данных, которые так часто используются в браузере.

Методы, которые выполняют сетевые запросы, такие как XMLHttpRequest, fetch и подобные, могут изначально работать с `Blob` так же, как и с другими объектами, представляющими двоичные данные.

Мы можем легко конвертировать `Blob` в низкоуровневые бинарные типы данных и обратно:

- Мы можем создать Blob из типизированного массива, используя конструктор `new Blob(...)`.
- Мы можем обратно создать `ArrayBuffer` из Blob, используя `FileReader`, а затем создать его представление для низкоуровневых операций.

File и FileReader

Объект File наследуется от объекта `Blob` и обладает возможностями по взаимодействию с файловой системой.

Есть два способа его получить.

Во-первых, есть конструктор, похожий на `Blob`:

```
new File(fileParts, fileName, [options])
```

- **`fileParts`** – массив значений `Blob`/`BufferSource`/строки.
- **`fileName`** – имя файла, строка.
- **`options`** – необязательный объект со свойством:
  - **`lastModified`** – дата последнего изменения в формате таймстамп (целое число).

Во-вторых, чаще всего мы получаем файл из `<input type="file">` или через перетаскивание с помощью мыши, или из других интерфейсов браузера. В этом случае файл получает эту информацию из ОС.

Так как `File` наследует от `Blob`, у объектов `File` есть те же свойства плюс:

- `name` – имя файла,
- `lastModified` – таймстамп для даты последнего изменения.

В этом примере мы получаем объект `File` из `<input type="file">`:

```
<input type="file" onchange="showFile(this)">

<script>
function showFile(input) {
  let file = input.files[0];

  alert(`File name: ${file.name}`); // например, my.png
  alert(`Last modified: ${file.lastModified}`); // например, 1552830408824
}
</script>
```

На заметку:

Через `<input>` можно выбрать несколько файлов, поэтому `input.files` – псевдомассив выбранных файлов. Здесь у нас только один файл, поэтому мы просто берём `input.files[0]`.

FileReader

FileReader объект, цель которого читать данные из `Blob` (и, следовательно, из `File` тоже).

Данные передаются при помощи событий, так как чтение с диска может занять время.

Конструктор:

```
let reader = new FileReader(); // без аргументов
```

Основные методы:

- **`readAsArrayBuffer(blob)`** – считать данные как `ArrayBuffer`
- **`readAsText(blob, [encoding])`** – считать данные как строку (кодировка по умолчанию: `utf-8`)
- **`readAsDataURL(blob)`** – считать данные как base64-кодированный URL.
- **`abort()`** – отменить операцию.

Выбор метода для чтения зависит от того, какой формат мы предпочитаем, как мы хотим далее использовать данные.

- `readAsArrayBuffer` – для бинарных файлов, для низкоуровневой побайтовой работы с бинарными данными. Для высокоуровневых операций у `File` есть свои методы, унаследованные от `Blob`, например, `slice`, мы можем вызвать их напрямую.
- `readAsText` – для текстовых файлов, когда мы хотим получить строку.
- `readAsDataURL` – когда мы хотим использовать данные в `src` для `img` или другого тега. Есть альтернатива – можно не читать файл, а вызвать `URL.createObjectURL(file)`, детали в главе Blob.

В процессе чтения происходят следующие события:

- `loadstart` – чтение начато.
- `progress` – срабатывает во время чтения данных.
- `load` – нет ошибок, чтение окончено.
- `abort` – вызван `abort()`.
- `error` – произошла ошибка.
- `loadend` – чтение завершено (успешно или нет).

Когда чтение закончено, мы сможем получить доступ к его результату следующим образом:

- `reader.result` результат чтения (если оно успешно)

- `reader.error` объект ошибки (при неудаче).

Наиболее часто используемые события – это, конечно же, `load` и `error`.

Вот пример чтения файла:

```
<input type="file" onchange="readFile(this)">

<script>
function readFile(input) {
  let file = input.files[0];

  let reader = new FileReader();

  reader.readAsText(file);

  reader.onload = function() {
    console.log(reader.result);
  };

  reader.onerror = function() {
    console.log(reader.error);
  };

}
</script>
```

`FileReader` для Blob

Как упоминалось в главе Blob, `FileReader` работает для любых объектов Blob, а не только для файлов.

Поэтому мы можем использовать его для преобразования Blob в другой формат:

- `readAsArrayBuffer(blob)` – в `ArrayBuffer`,
- `readAsText(blob, [encoding])` – в строку (альтернатива `TextDecoder`),
- `readAsDataURL(blob)` – в формат base64-кодированного URL.

Для Web Worker также доступен `FileReaderSync`

Для веб-воркеров доступен синхронный вариант `FileReader`, именуемый FileReaderSync .

Его методы считывания `read*` не генерируют события, а возвращают результат, как это делают обычные функции.

Но это только внутри веб-воркера, поскольку задержки в синхронных вызовах, которые возможны при чтении из файла, в веб-воркерах менее важны. Они не влияют на страницу.

Итого

`File` объекты наследуют от `Blob`.

Помимо методов и свойств `Blob`, объекты `File` также имеют свойства `name` и `lastModified` плюс внутреннюю возможность чтения из файловой системы. Обычно мы получаем объекты `File` из пользовательского ввода, например, через `<input>` или перетаскиванием с помощью мыши, в событии `dragend`.

Объекты `FileReader` могут читать из файла или Blob в одном из трёх форматов:

- Строка (`readAsText`).
- `ArrayBuffer` (`readAsArrayBuffer`).
- URL в формате base64 (`readAsDataURL`).

Однако, во многих случаях нам не нужно читать содержимое файла. Как и в случае с Blob, мы можем создать короткий URL с помощью `URL.createObjectURL(file)` и использовать его в теге `<a>` или `<img>`. Таким образом, файл может быть загружен или показан в виде изображения, как часть canvas и т.д.

А если мы собираемся отправить `File` по сети, то это также легко, поскольку в сетевые методы, такие как `XMLHttpRequest` или `fetch`, встроена возможность отсылки `File`.

Сетевые запросы

Fetch

JavaScript может отправлять сетевые запросы на сервер и подгружать новую информацию по мере необходимости.

Например, мы можем использовать сетевой запрос, чтобы:

- Отправить заказ,
- Загрузить информацию о пользователе,
- Запросить последние обновления с сервера,
- …и т.п.

Для сетевых запросов из JavaScript есть широко известный термин «AJAX» (аббревиатура от **A**synchronous **J**avaScript **A**nd **X**ML). XML мы использовать не обязаны, просто термин старый, поэтому в нём есть это слово. Возможно, вы его уже где-то слышали.

Есть несколько способов делать сетевые запросы и получать информацию с сервера.

Метод `fetch()` — современный и очень мощный, поэтому начнём с него. Он не поддерживается старыми (можно использовать полифил), но поддерживается всеми современными браузерами.

Базовый синтаксис:

```
let promise = fetch(url, [options])
```

- **`url`** – URL для отправки запроса.
- **`options`** – дополнительные параметры: метод, заголовки и так далее.

Без `options` это простой GET-запрос, скачивающий содержимое по адресу `url`.

Браузер сразу же начинает запрос и возвращает промис, который внешний код использует для получения результата.

Процесс получения ответа обычно происходит в два этапа.

Во-первых, `promise` выполняется с объектом встроенного класса Response в качестве результата, как только сервер пришлёт заголовки ответа.

На этом этапе мы можем проверить статус HTTP-запроса и определить, выполнился ли он успешно, а также посмотреть заголовки, но пока без тела ответа.

Промис завершается с ошибкой, если `fetch` не смог выполнить HTTP-запрос, например при ошибке сети или если нет такого сайта. HTTP-статусы такие как 404 или 500, не являются ошибкой.

Мы можем увидеть HTTP-статус в свойствах ответа:

- **`status`** – код статуса HTTP-запроса, например 200.

- **`ok`** – логическое значение: будет `true`, если код HTTP-статуса в диапазоне 200-299.

Например:

```
let response = await fetch(url);

if (response.ok) { // если HTTP-статус в диапазоне 200-299
  // получаем тело ответа (см. про этот метод ниже)
  let json = await response.json();
} else {
  alert("Ошибка HTTP: " + response.status);
}
```

Во-вторых, для получения тела ответа нам нужно использовать дополнительный вызов метода.

`Response` предоставляет несколько методов, основанных на промисах, для доступа к телу ответа в различных форматах:

- **`response.text()`** – читает ответ и возвращает как обычный текст,
- **`response.json()`** – декодирует ответ в формате JSON,
- **`response.formData()`** – возвращает ответ как объект `FormData` (разберём его в следующей главе),
- **`response.blob()`** – возвращает объект как Blob (бинарные данные с типом),
- **`response.arrayBuffer()`** – возвращает ответ как ArrayBuffer (низкоуровневое представление бинарных данных),
- помимо этого, `response.body` – это объект ReadableStream, с помощью которого можно считывать тело запроса по частям. Мы рассмотрим и такой пример несколько позже.

Например, получим JSON-объект с последними коммитами из репозитория на GitHub:

```
let url = 'https://api.github.com/repos/javascript-tutorial/en.javascript.info/comm
let response = await fetch(url);

let commits = await response.json(); // читаем ответ в формате JSON

alert(commits[0].author.login);
```

То же самое без `await`, с использованием промисов:

```
fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits'
  .then(response => response.json())
  .then(commits => alert(commits[0].author.login));
```

Для получения ответа в виде текста используем `await response.text()` вместо `.json()`:

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.jav

let text = await response.text(); // прочитать тело ответа как текст

alert(text.slice(0, 80) + '...');
```

В качестве примера работы с бинарными данными, давайте запросим и выведем на экран логотип спецификации «fetch» (см. главу Blob, чтобы узнать про операции с `Blob`):

```
let response = await fetch('/article/fetch/logo-fetch.svg');

let blob = await response.blob(); // скачиваем как Blob-объект

// создаём <img>
let img = document.createElement('img');
img.style = 'position:fixed;top:10px;left:10px;width:100px';
document.body.append(img);

// выводим на экран
img.src = URL.createObjectURL(blob);

setTimeout(() => { // прячем через три секунды
  img.remove();
  URL.revokeObjectURL(img.src);
}, 3000);
```

Важно:

Мы можем выбрать только один метод чтения ответа.

Если мы уже получили ответ с `response.text()`, тогда `response.json()` не сработает, так как данные уже были обработаны.

```
let text = await response.text(); // тело ответа обработано
let parsed = await response.json(); // ошибка (данные уже были обработаны)
```

Заголовки ответа

Заголовки ответа хранятся в похожем на `Map` объекте `response.headers`.

Это не совсем `Map`, но мы можем использовать такие же методы, как с `Map`, чтобы получить заголовок по его имени или перебрать заголовки в цикле:

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.jav

// получить один заголовок
alert(response.headers.get('Content-Type')); // application/json; charset=utf-8

// перебрать все заголовки
for (let [key, value] of response.headers) {
  alert(`${key} = ${value}`);
}
```

Заголовки запроса

Для установки заголовка запроса в `fetch` мы можем использовать опцию `headers`. Она содержит объект с исходящими заголовками, например:

```
let response = fetch(protectedUrl, {
  headers: {
    Authentication: 'secret'
  }
});
```

Есть список запрещённых HTTP-заголовков, которые мы не можем установить:

- `Accept-Charset`, `Accept-Encoding`
- `Access-Control-Request-Headers`
- `Access-Control-Request-Method`
- `Connection`
- `Content-Length`
- `Cookie`, `Cookie2`
- `Date`
- `DNT`
- `Expect`
- `Host`
- `Keep-Alive`

- `Origin`
- `Referer`
- `TE`
- `Trailer`
- `Transfer-Encoding`
- `Upgrade`
- `Via`
- `Proxy-*`
- `Sec-*`

Эти заголовки обеспечивают достоверность данных и корректную работу протокола HTTP, поэтому они контролируются исключительно браузером.

POST-запросы

Для отправки `POST`-запроса или запроса с другим методом, нам необходимо использовать `fetch` параметры:

- **`method`** – HTTP метод, например `POST`,
- **`body`** – тело запроса, одно из списка:
  - строка (например, в формате JSON),
  - объект `FormData` для отправки данных как `form/multipart`,
  - `Blob`/`BufferSource` для отправки бинарных данных,
  - URLSearchParams для отправки данных в кодировке `x-www-form-urlencoded`, используется редко.

Чаще всего используется JSON.

Например, этот код отправляет объект `user` как JSON:

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

let response = await fetch('/article/fetch/post/user', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
});
```

```
let result = await response.json();
alert(result.message);
```

Заметим, что так как тело запроса `body` – строка, то заголовок `Content-Type` по умолчанию будет `text/plain;charset=UTF-8`.

Но, так как мы посылаем JSON, то используем параметр `headers` для отправки вместо этого `application/json`, правильный `Content-Type` для JSON.

Отправка изображения

Мы можем отправить бинарные данные при помощи `fetch`, используя объекты `Blob` или `BufferSource`.

В этом примере есть элемент `<canvas>`, на котором мы можем рисовать движением мыши. При нажатии на кнопку «Отправить» изображение отправляется на сервер:

```
<body style="margin:0">
  <canvas id="canvasElem" width="100" height="80" style="border:1px solid"></canvas

  <input type="button" value="Отправить" onclick="submit()">

  <script>
    canvasElem.onmousemove = function(e) {
      let ctx = canvasElem.getContext('2d');
      ctx.lineTo(e.clientX, e.clientY);
      ctx.stroke();
    };

    async function submit() {
      let blob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png
      let response = await fetch('/article/fetch/post/image', {
        method: 'POST',
        body: blob
      });

      // сервер ответит подтверждением и размером изображения
      let result = await response.json();
      alert(result.message);
    }

  </script>
</body>
```

Заметим, что здесь нам не нужно вручную устанавливать заголовок `Content-Type`, потому что объект `Blob` имеет встроенный тип (`image/png`, заданный в `toBlob`). При отправке объектов `Blob` он автоматически становится значением `Content-Type`.

Функция `submit()` может быть переписана без `async/await`, например, так:

```
function submit() {
  canvasElem.toBlob(function(blob) {
    fetch('/article/fetch/post/image', {
      method: 'POST',
      body: blob
    })
      .then(response => response.json())
      .then(result => alert(JSON.stringify(result, null, 2)))
  }, 'image/png');
}
```

Итого

Типичный запрос с помощью `fetch` состоит из двух операторов `await`:

```
let response = await fetch(url, options); // завершается с заголовками ответа
let result = await response.json(); // читать тело ответа в формате JSON
```

Или, без `await`:

```
fetch(url, options)
  .then(response => response.json())
  .then(result => /* обрабатываем результат */)
```

Параметры ответа:

- `response.status` – HTTP-код ответа,
- `response.ok` – `true`, если статус ответа в диапазоне 200-299.
- `response.headers` – похожий на `Map` объект с HTTP-заголовками.

Методы для получения тела ответа:

- `response.text()` – возвращает ответ как обычный текст,
- `response.json()` – преобразовывает ответ в JSON-объект,
- `response.formData()` – возвращает ответ как объект FormData (кодировка form/multipart, см. следующую главу),
- `response.blob()` – возвращает объект как Blob (бинарные данные с типом),
- `response.arrayBuffer()` – возвращает ответ как ArrayBuffer (низкоуровневые бинарные данные),

Опции `fetch`, которые мы изучили на данный момент:

- `method` – HTTP-метод,
- `headers` – объект с запрашиваемыми заголовками (не все заголовки разрешены),
- `body` – данные для отправки (тело запроса) в виде текста, `FormData`, `BufferSource`, `Blob` или `UrlSearchParams`.

В следующих главах мы рассмотрим больше параметров и вариантов использования `fetch`.

Задачи

Получите данные о пользователях GitHub

Создайте асинхронную функцию `getUsers(names)`, которая получает на вход массив логинов пользователей GitHub, запрашивает у GitHub информацию о них и возвращает массив объектов-пользователей.

Информация о пользователе GitHub с логином USERNAME доступна по ссылке: `https://api.github.com/users/USERNAME`.

В песочнице есть тестовый пример.

Важные детали:

1. На каждого пользователя должен приходиться один запрос `fetch`.
2. Запросы не должны ожидать завершения друг друга. Надо, чтобы данные приходили как можно быстрее.
3. Если какой-то запрос завершается ошибкой или оказалось, что данных о запрашиваемом пользователе нет, то функция должна возвращать `null` в массиве результатов.

Открыть песочницу с тестами для задачи.

К решению

FormData

В этой главе речь пойдёт об отправке HTML-форм: с файлами и без, с дополнительными полями и так далее. Объекты FormData помогут нам с этим. Как вы, наверняка, догадались по его названию, это объект, представляющий данные HTML формы.

Конструктор:

```
let formData = new FormData([form]);
```

Если передать в конструктор элемент HTML-формы `form`, то создаваемый объект автоматически прочитает из неё поля.

Его особенность заключается в том, что методы для работы с сетью, например `fetch`, позволяют указать объект `FormData` в свойстве тела запроса `body`.

Он будет соответствующим образом закодирован и отправлен с заголовком `Content-Type: form/multipart`.

То есть, для сервера это выглядит как обычная отправка формы.

Отправка простой формы

Давайте сначала отправим простую форму.

Как вы видите, код очень компактный:

```
<form id="formElem">
  <input type="text" name="name" value="John">
  <input type="text" name="surname" value="Smith">
  <input type="submit">
</form>

<script>
  formElem.onsubmit = async (e) => {
    e.preventDefault();

    let response = await fetch('/article/formdata/post/user', {
      method: 'POST',
      body: new FormData(formElem)
```

```
    });

    let result = await response.json();

    alert(result.message);
  };
</script>
```

В этом примере серверный код не представлен, он за рамками этой статьи, он принимает POST-запрос с данными формы и отвечает сообщением «Пользователь сохранён».

Методы объекта FormData

С помощью указанных ниже методов мы можем изменять поля в объекте `FormData`:

- `formData.append(name, value)` – добавляет к объекту поле с именем `name` и значением `value`,
- `formData.append(name, blob, fileName)` – добавляет поле, как будто в форме имеется элемент `<input type="file">`, третий аргумент `fileName` устанавливает имя файла (не имя поля формы), как будто это имя из файловой системы пользователя,
- `formData.delete(name)` – удаляет поле с заданным именем `name`,
- `formData.get(name)` – получает значение поля с именем `name`,
- `formData.has(name)` – если существует поле с именем `name`, то возвращает `true`, иначе `false`

Технически форма может иметь много полей с одним и тем же именем `name`, поэтому несколько вызовов `append` добавят несколько полей с одинаковыми именами.

Ещё существует метод `set`, его синтаксис такой же, как у `append`. Разница в том, что `.set` удаляет все уже имеющиеся поля с именем `name` и только затем добавляет новое. То есть этот метод гарантирует, что будет существовать только одно поле с именем `name`, в остальном он аналогичен `.append`:

- `formData.set(name, value)`,
- `formData.set(name, blob, fileName)`.

Поля объекта `formData` можно перебирать, используя цикл `for..of`:

```
let formData = new FormData();
formData.append('key1', 'value1');
formData.append('key2', 'value2');

// Список пар ключ/значение
for(let [name, value] of formData) {
  alert(`${name} = ${value}`); // key1=value1, потом key2=value2
}
```

Отправка формы с файлом

Объекты `FormData` всегда отсылаются с заголовком `Content-Type: form/multipart`, этот способ кодировки позволяет отсылать файлы. Таким образом, поля `<input type="file">` тоже отправляются, как это и происходит в случае обычной формы.

Пример такой формы:

```
<form id="formElem">
  <input type="text" name="firstName" value="John">
  Картинка: <input type="file" name="picture" accept="image/*">
  <input type="submit">
</form>

<script>
  formElem.onsubmit = async (e) => {
    e.preventDefault();

    let response = await fetch('/article/formdata/post/user-avatar', {
      method: 'POST',
      body: new FormData(formElem)
    });

    let result = await response.json();

    alert(result.message);
  };
</script>
```

John Картинка: Choose File No file chosen Submit

Отправка формы с Blob-данными

Ранее в главе Fetch мы видели, что очень легко отправить динамически сгенерированные бинарные данные в формате `Blob`. Мы можем явно передать её в параметр `body` запроса `fetch`.

Но на практике бывает удобнее отправлять изображение не отдельно, а в составе формы, добавив дополнительные поля для имени и другие метаданные.

Кроме того, серверы часто настроены на приём именно форм, а не просто бинарных данных.

В примере ниже в отсылается изображение из `<canvas>` и ещё несколько полей, как форма, используя `FormData`:

```
<body style="margin:0">
  <canvas id="canvasElem" width="100" height="80" style="border:1px solid"></canvas

  <input type="button" value="Отправить" onclick="submit()">

  <script>
    canvasElem.onmousemove = function(e) {
      let ctx = canvasElem.getContext('2d');
      ctx.lineTo(e.clientX, e.clientY);
      ctx.stroke();
    };

    async function submit() {
      let imageBlob = await new Promise(resolve => canvasElem.toBlob(resolve, 'imag

      let formData = new FormData();
      formData.append("firstName", "John");
      formData.append("image", imageBlob, "image.png");

      let response = await fetch('/article/formdata/post/image-form', {
        method: 'POST',
        body: formData
      });
      let result = await response.json();
      alert(result.message);
    }

  </script>
</body>
```

Пожалуйста, обратите внимание на то, как добавляется изображение `Blob`:

```
formData.append("image", imageBlob, "image.png");
```

Это как если бы в форме был элемент `<input type="file" name="image">` и пользователь прикрепил бы файл с именем `"image.png"` (3й аргумент) и данными `imageBlob` (2й аргумент) из своей файловой системы.

Сервер прочитает и данные и файл, точно так же, как если бы это была обычная отправка формы.

Итого

Объекты FormData используются, чтобы взять данные из HTML-формы и отправить их с помощью `fetch` или другого метода для работы с сетью.

Мы можем создать такой объект уже с данными, передав в конструктор HTML-форму – `new FormData(form)`, или же можно создать объект вообще без формы и затем добавить к нему поля с помощью методов:

- `formData.append(name, value)`
- `formData.append(name, blob, fileName)`
- `formData.set(name, value)`
- `formData.set(name, blob, fileName)`

Отметим две особенности:

1. Метод `set` удаляет предыдущие поля с таким же именем, а `append` – нет. В этом их единственное отличие.
2. Чтобы послать файл, нужно использовать синтаксис с тремя аргументами, в качестве третьего как раз указывается имя файла, которое обычно, при `<input type="file">`, берётся из файловой системы.

Другие методы:

- `formData.delete(name)`
- `formData.get(name)`
- `formData.has(name)`

Вот и всё!

Fetch: ход загрузки

Метод `fetch` позволяет отслеживать процесс *получения* данных.

Заметим, на данный момент в `fetch` нет способа отслеживать процесс *отправки*. Для этого используйте XMLHttpRequest, позже мы его рассмотрим.

Чтобы отслеживать ход загрузки данных с сервера, можно использовать свойство `response.body`. Это `ReadableStream` («поток для чтения») – особый объект, который предоставляет тело ответа по частям, по мере поступления. Потоки для чтения описаны в спецификации Streams API.

В отличие от `response.text()`, `response.json()` и других методов, `response.body` даёт полный контроль над процессом чтения, и мы можем подсчитать, сколько данных получено на каждый момент.

Вот примерный код, который читает ответ из `response.body`:

```
// вместо response.json() и других методов
const reader = response.body.getReader();

// бесконечный цикл, пока идёт загрузка
while(true) {
  // done становится true в последнем фрагменте
  // value - Uint8Array из байтов каждого фрагмента
  const {done, value} = await reader.read();

  if (done) {
    break;
  }

  console.log(`Получено ${value.length} байт`)
}
```

Результат вызова `await reader.read()` – это объект с двумя свойствами:

- **`done`** – `true`, когда чтение закончено, иначе `false`.
- **`value`** – типизированный массив данных ответа `Uint8Array`.

> **На заметку:**
>
> Streams API также описывает асинхронный перебор по `ReadableStream`, при помощи цикла `for await..of`, но он пока слабо поддерживается (см. задачи для браузеров), поэтому используем цикл `while`.

Мы получаем новые фрагменты данных в цикле, пока загрузка не завершится, то есть пока `done` не станет `true`.

Чтобы отслеживать процесс загрузки, нам нужно при получении очередного фрагмента прибавлять его длину `value` к счётчику.

Вот полный рабочий пример, который получает ответ сервера и в процессе получения выводит в консоли длину полученных данных:

```
// Шаг 1: начинаем загрузку fetch, получаем поток для чтения
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.jav

const reader = response.body.getReader();

// Шаг 2: получаем длину содержимого ответа
const contentLength = +response.headers.get('Content-Length');

// Шаг 3: считываем данные:
let receivedLength = 0; // количество байт, полученных на данный момент
let chunks = []; // массив полученных двоичных фрагментов (составляющих тело ответа
while(true) {
  const {done, value} = await reader.read();

  if (done) {
    break;
  }

  chunks.push(value);
  receivedLength += value.length;

  console.log(`Получено ${receivedLength} из ${contentLength}`)
}

// Шаг 4: соединим фрагменты в общий типизированный массив Uint8Array
let chunksAll = new Uint8Array(receivedLength); // (4.1)
let position = 0;
for(let chunk of chunks) {
  chunksAll.set(chunk, position); // (4.2)
  position += chunk.length;
}

// Шаг 5: декодируем Uint8Array обратно в строку
let result = new TextDecoder("utf-8").decode(chunksAll);

// Готово!
let commits = JSON.parse(result);
alert(commits[0].author.login);
```

Разберёмся, что здесь произошло:

1. Мы обращаемся к `fetch` как обычно, но вместо вызова `response.json()` мы получаем доступ к потоку чтения `response.body.getReader()`.

   Обратите внимание, что мы не можем использовать одновременно оба эти метода для чтения одного и того же ответа: либо обычный метод `response.json()`, либо чтение потока `response.body`.

2. Ещё до чтения потока мы можем вычислить полную длину ответа из заголовка `Content-Length`.

Он может быть нечитаемым при запросах на другой источник (подробнее в разделе Fetch: запросы на другие сайты) и, в общем-то, серверу необязательно его устанавливать. Тем не менее, обычно длина указана.

3. Вызываем `await reader.read()` до окончания загрузки.

   Всё, что получили, мы складываем по «кусочкам» в массив `chunks`. Это важно, потому что после того, как ответ получен, мы уже не сможем «перечитать» его, используя `response.json()` или любой другой способ (попробуйте – будет ошибка).

4. В самом конце у нас типизированный массив – `Uint8Array`. В нём находятся фрагменты данных. Нам нужно их склеить, чтобы получить строку. К сожалению, для этого нет специального метода, но можно сделать, например, так:

   1. Создаём `chunksAll = new Uint8Array(receivedLength)` – массив того же типа заданной длины.
   2. Используем `.set(chunk, position)` для копирования каждого фрагмента друг за другом в него.

5. Наш результат теперь хранится в `chunksAll`. Это не строка, а байтовый массив.

   Чтобы получить именно строку, надо декодировать байты. Встроенный объект TextDecoder как раз этим и занимается. Потом мы можем, если необходимо, преобразовать строку в данные с помощью `JSON.parse`.

   Что если результат нам нужен в бинарном виде вместо строки? Это ещё проще. Замените шаги 4 и 5 на создание единого `Blob` из всех фрагментов:

```
let blob = new Blob(chunks);
```

В итоге у нас есть результат (строки или `Blob`, смотря что удобно) и отслеживание прогресса получения.

На всякий случай повторимся, что здесь мы рассмотрели, как отслеживать процесс получения данных с сервера, а не их отправки на сервер. Для отслеживания отправки у `fetch` пока нет способа.

Fetch: прерывание запроса

Как мы знаем, метод `fetch` возвращает промис. А в JavaScript в целом нет понятия «отмены» промиса. Как же прервать запрос `fetch`?

Для таких целей существует специальный встроенный объект: `AbortController`, который можно использовать для отмены не только

`fetch`, но и других асинхронных задач.

Использовать его достаточно просто:

- Шаг 1: создаём контроллер:

```
let controller = new AbortController();
```

Контроллер `controller` – чрезвычайно простой объект.

- Он имеет единственный метод `abort()` и единственное свойство `signal`.
- При вызове `abort()`:
  - генерируется событие с именем `abort` на объекте `controller.signal`
  - свойство `controller.signal.aborted` становится равным `true`.

Все, кто хочет узнать о вызове `abort()`, ставят обработчики на `controller.signal`, чтобы отслеживать его.

Вот так (пока без `fetch`):

```
let controller = new AbortController();
let signal = controller.signal;

// срабатывает при вызове controller.abort()
signal.addEventListener('abort', () => alert("отмена!"));

controller.abort(); // отмена!

alert(signal.aborted); // true
```

- Шаг 2: передайте свойство `signal` опцией в метод `fetch`:

```
let controller = new AbortController();
fetch(url, {
  signal: controller.signal
});
```

Метод `fetch` умеет работать с `AbortController`, он слушает событие `abort` на `signal`.

- Шаг 3: чтобы прервать выполнение `fetch`, вызовите `controller.abort()`:

```
controller.abort();
```

Вот и всё: `fetch` получает событие из `signal` и прерывает запрос.

Когда `fetch` отменяется, его промис завершается с ошибкой `AbortError`, поэтому мы должны обработать её, например, в `try..catch`:

```
// прервать через 1 секунду
let controller = new AbortController();
setTimeout(() => controller.abort(), 1000);

try {
  let response = await fetch('/article/fetch-abort/demo/hang', {
    signal: controller.signal
  });
} catch(err) {
  if (err.name == 'AbortError') { // обработь ошибку от вызова abort()
    alert("Прервано!");
  } else {
    throw err;
  }
}
```

`AbortController` – масштабируемый, он позволяет отменить несколько вызовов `fetch` одновременно.

Например, здесь мы запрашиваем много URL параллельно, и контроллер прерывает их все:

```
let urls = [...]; // список URL для параллельных fetch

let controller = new AbortController();

let fetchJobs = urls.map(url => fetch(url, {
  signal: controller.signal
}));

let results = await Promise.all(fetchJobs);

// если откуда-то вызвать controller.abort(),
// то это прервёт все вызовы fetch
```

Если у нас есть собственные асинхронные задачи, отличные от `fetch`, мы можем использовать один `AbortController` для их остановки вместе с `fetch`.

Нужно лишь слушать его событие `abort`:

```
let urls = [...];
let controller = new AbortController();

let ourJob = new Promise((resolve, reject) => { // наша задача
  ...
  controller.signal.addEventListener('abort', reject);
});

let fetchJobs = urls.map(url => fetch(url, { // запросы fetch
  signal: controller.signal
}));

// ожидать выполнения нашей задачи и всех запросов
let results = await Promise.all([...fetchJobs, ourJob]);

// вызов откуда-нибудь ещё:
// controller.abort() прервёт все вызовы fetch и наши задачи
```

Так что `AbortController` существует не только для `fetch`, это универсальный объект для отмены асинхронных задач, в `fetch` встроена интеграция с ним.

Fetch: запросы на другие сайты

Если мы сделаем запрос `fetch` на другой веб-сайт, он, вероятно, завершится неудачей.

Например, давайте попробуем запросить `http://example.com`:

```
try {
  await fetch('http://example.com');
} catch(err) {
  alert(err); // Failed to fetch
}
```

Вызов `fetch` не удался, как и ожидалось.

Ключевым понятием здесь является *источник* (origin) – комбинация домен/порт/протокол.

Запросы на другой источник – отправленные на другой домен (или даже поддомен), или протокол, или порт – требуют специальных заголовков от удалённой стороны.

Эта политика называется «CORS»: Cross-Origin Resource Sharing («совместное использование ресурсов между разными источниками»).

Зачем нужен CORS? Экскурс в историю

CORS существует для защиты интернет от злых хакеров.

Серьёзно. Давайте сделаем краткое историческое отступление.

Многие годы скрипт с одного сайта не мог получить доступ к содержимому другого сайта.

Это простое, но могучее правило было основой интернет-безопасности. Например, хакерский скрипт с сайта `hacker.com` не мог получить доступ к почтовому ящику пользователя на сайте `gmail.com`. И люди чувствовали себя спокойно.

В то время в JavaScript не было методов для сетевых запросов. Это был «игрушечный» язык для украшения веб-страниц.

Но веб-разработчики жаждали большей власти. Чтобы обойти этот запрет и всё же получать данные с других сайтов, были придуманы разные хитрости.

Использование форм

Одним из способов общения с другим сервером была отправка туда формы `<form>`. Люди отправляли её в `<iframe>`, чтобы оставаться на текущей странице, вот так:

```
<!-- цель формы -->
<iframe name="iframe"></iframe>

<!-- форма могла быть динамически сгенерирована и отправлена с помощью JavaScript -
<form target="iframe" method="POST" action="http://another.com/…">
  ...
</form>
```

Таким способом было возможно сделать GET/POST запрос к другому сайту даже без сетевых методов, так как формы можно отправлять куда угодно. Но так как запрещено получать доступ к содержимому `<iframe>` с другого сайта, прочитать ответ было невозможно.

Если быть точным, были трюки и для этого, требующие специального кода на странице и в ифрейме, так что общение с ифреймом было технически возможно. Сейчас мы не будем вдаваться в подробности, пусть эти динозавры покоятся в мире.

Использование скриптов

Ещё один трюк заключался в использовании тега `script`. У него может быть любой `src`, с любым доменом, например `<script src="http://another.com/…">`. Это даёт возможность загрузить и выполнить скрипт откуда угодно.

Если сайт, например `another.com`, хотел предоставить данные для такого доступа, он предоставлял так называемый «протокол JSONP» (**JSON** with **P**adding)".

Вот как он работал.

Например, нам на нашем сайте нужны данные с сайта `http://another.com`, скажем, погода:

1. Сначала, заранее, объявляем глобальную функцию для обработки данных, например `gotWeather`.

```
// 1. Объявить функцию для обработки погодных данных
function gotWeather({ temperature, humidity }) {
  alert(`температура: ${temperature}, влажность: ${humidity}`);
}
```

2. Затем создаём тег `<script>` с `src="http://another.com/weather.json?callback=gotWeather"`, при этом имя нашей функции – в URL-параметре `callback`.

```
let script = document.createElement('script');
script.src = `http://another.com/weather.json?callback=gotWeather`;
document.body.append(script);
```

3. Удалённый сервер с `another.com` должен в ответ сгенерировать скрипт, который вызывает `gotWeather(...)` с данными, которые хочет передать.

```
// Ожидаемый ответ от сервера выглядит так:
gotWeather({
  temperature: 25,
  humidity: 78
});
```

4. Когда этот скрипт загрузится и выполнится, наша функция `gotWeather` получает данные.

Это работает и не нарушает безопасность, потому что обе стороны согласились передавать данные таким образом. А когда обе стороны согласны, то это

определённо не хак. Всё ещё существуют сервисы, которые предоставляют такой доступ, так как это работает даже для очень старых браузеров.

Спустя некоторое время в браузерном JavaScript появились методы для сетевых запросов.

Вначале запросы на другой источник были запрещены. Но в результате долгих дискуссий было решено разрешить их делать, но для использования новых возможностей требовать разрешение сервера, выраженное в специальных заголовках.

Простые запросы

Есть два вида запросов на другой источник:

1. Простые.
2. Все остальные.

Простые запросы будут попроще, поэтому давайте начнём с них.

Простой запрос – это запрос, удовлетворяющий следующим условиям:

1. Простой метод : GET, POST или HEAD
2. Простые заголовки – разрешены только:
   - `Accept`,
   - `Accept-Language`,
   - `Content-Language`,
   - `Content-Type` со значением `application/x-www-form-urlencoded`, `multipart/form-data` или `text/plain`.

Любой другой запрос считается «непростым». Например, запрос с методом `PUT` или с HTTP-заголовком `API-Key` не соответствует условиям.

Принципиальное отличие между ними состоит в том, что «простой запрос» может быть сделан через `<form>` или `<script>`, без каких-то специальных методов.

Таким образом, даже очень старый сервер должен быть способен принять простой запрос.

В противоположность этому, запросы с нестандартными заголовками или, например, методом `DELETE` нельзя создать таким способом. Долгое время JavaScript не мог делать такие запросы. Поэтому старый сервер может предположить, что такие запросы поступают от привилегированного источника, «просто потому, что веб-страница неспособна их посылать».

Когда мы пытаемся сделать непростой запрос, браузер посылает специальный предварительный запрос («предзапрос», по англ. «preflight»), который спрашивает у сервера – согласен ли он принять такой непростой запрос или нет?

И, если сервер явно не даёт согласие в заголовках, непростой запрос не посылается.

Далее мы разберём конкретные детали.

CORS для простых запросов

При запросе на другой источник браузер всегда ставит «от себя» заголовок `Origin`.

Например, если мы запрашиваем `https://anywhere.com/request` со страницы `https://javascript.info/page`, заголовки будут такими:

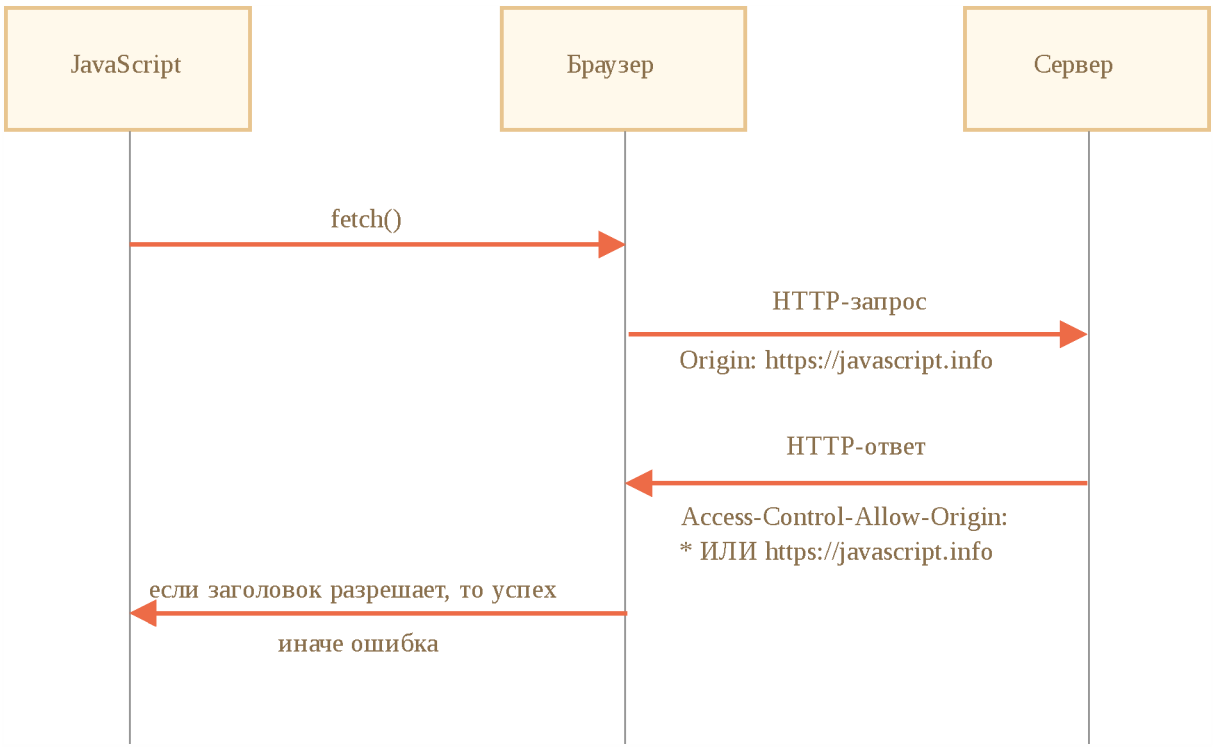
```
GET /request
Host: anywhere.com
Origin: https://javascript.info
...
```

Как вы можете видеть, заголовок `Origin` содержит именно источник (домен/протокол/порт), без пути.

Сервер может проверить `Origin` и, если он согласен принять такой запрос, добавить особый заголовок `Access-Control-Allow-Origin` к ответу. Этот заголовок должен содержать разрешённый источник (в нашем случае `https://javascript.info`) или звёздочку `*`. Тогда ответ успешен, в противном случае возникает ошибка.

Здесь браузер играет роль доверенного посредника:

1. Он гарантирует, что к запросу на другой источник добавляется правильный заголовок `Origin`.
2. Он проверяет наличие разрешающего заголовка `Access-Control-Allow-Origin` в ответе и, если всё хорошо, то JavaScript получает доступ к ответу сервера, в противном случае – доступ запрещается с ошибкой.



Вот пример ответа сервера, который разрешает доступ:

```
200 OK
Content-Type:text/html; charset=UTF-8
Access-Control-Allow-Origin: https://javascript.info
```

Заголовки ответа

По умолчанию при запросе к другому источнику JavaScript может получить доступ только к так называемым «простым» заголовкам ответа:

- `Cache-Control`
- `Content-Language`
- `Content-Type`
- `Expires`
- `Last-Modified`
- `Pragma`

При доступе к любому другому заголовку ответа будет ошибка.

Обратите внимание: нет `Content-Length`

Пожалуйста, обратите внимание: в списке нет заголовка `Content-Length`!

Этот заголовок содержит полную длину ответа. Поэтому если мы загружаем что-то и хотели бы отслеживать прогресс в процентах, то требуется дополнительное разрешение для доступа к этому заголовку (читайте ниже).

Чтобы разрешить JavaScript доступ к любому другому заголовку ответа, сервер должен указать заголовок `Access-Control-Expose-Headers`. Он содержит список, через запятую, заголовков, которые не являются простыми, но доступ к которым разрешён.

Например:

```
200 OK
Content-Type:text/html; charset=UTF-8
Content-Length: 12345
API-Key: 2c9de507f2c54aa1
Access-Control-Allow-Origin: https://javascript.info
Access-Control-Expose-Headers: Content-Length,API-Key
```

При таком заголовке `Access-Control-Expose-Headers`, скрипту разрешено получить заголовки `Content-Length` и `API-Key` ответа.

«Непростые» запросы

Мы можем использовать любой HTTP-метод: не только `GET/POST`, но и `PATCH`, `DELETE` и другие.

Некоторое время назад никто не мог даже предположить, что веб-страница способна делать такие запросы. Так что могут существовать веб-сервисы, которые рассматривают нестандартный метод как сигнал: «Это не браузер». Они могут учитывать это при проверке прав доступа.

Поэтому, чтобы избежать недопониманий, браузер не делает «непростые» запросы (которые нельзя было сделать в прошлом) сразу. Перед этим он посылает предварительный запрос, спрашивая разрешения.

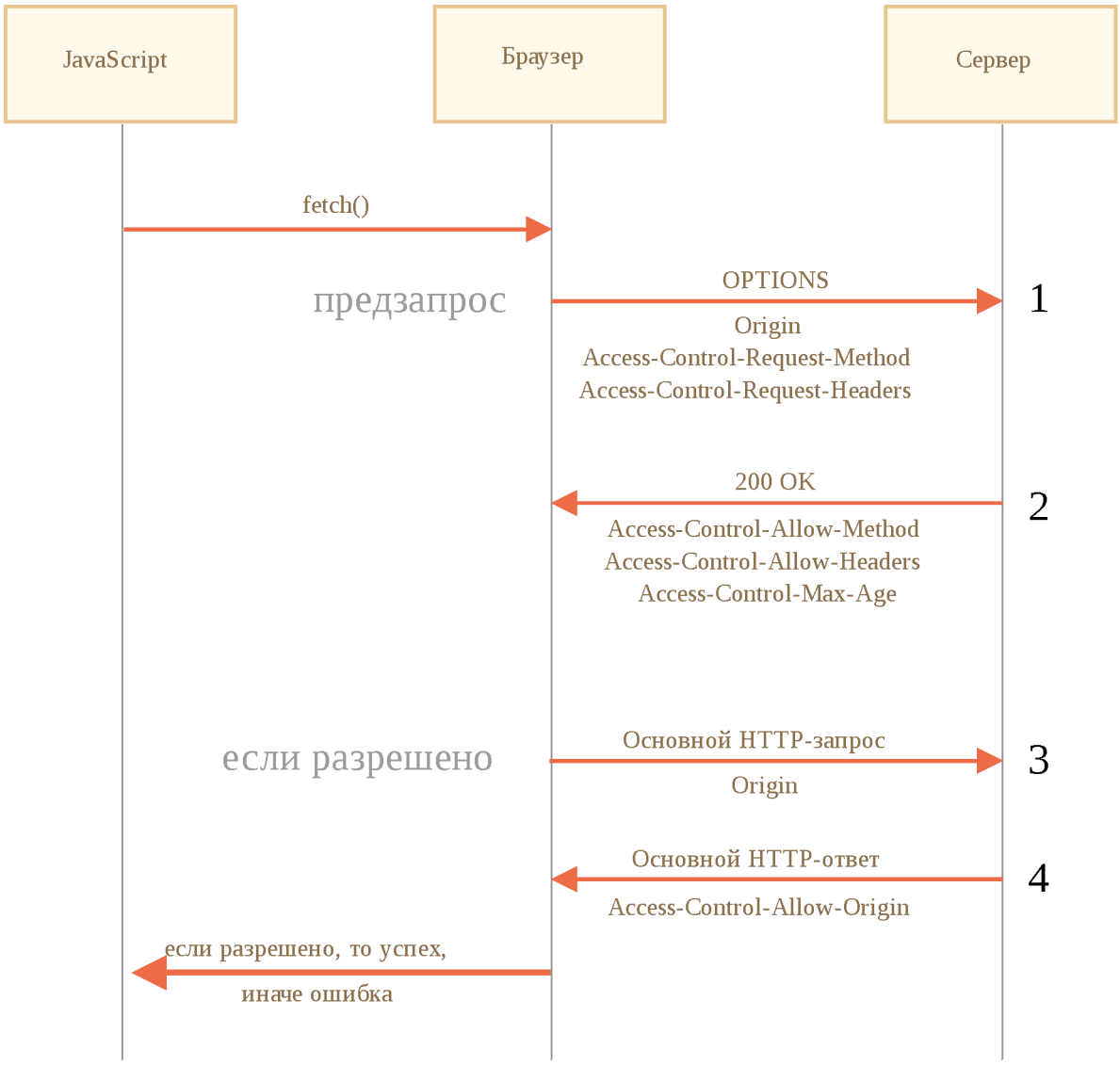
Предварительный запрос использует метод `OPTIONS`, у него нет тела, но есть два заголовка:

- `Access-Control-Request-Method` содержит HTTP-метод «непростого» запроса.

- `Access-Control-Request-Headers` предоставляет разделённый запятыми список его «непростых» HTTP-заголовков.

Если сервер согласен принимать такие запросы, то он должен ответить без тела, со статусом 200 и с заголовками:

- `Access-Control-Allow-Methods` должен содержать разрешённые методы.
- `Access-Control-Allow-Headers` должен содержать список разрешённых заголовков.
- Коме того, заголовок `Access-Control-Max-Age` может указывать количество секунд, на которое нужно кешировать разрешения. Так что браузеру не придётся посылать предзапрос для последующих запросов, удовлетворяющих данным разрешениям.



Давайте пошагово посмотрим, как это работает, на примере `PATCH` запроса (этот метод часто используется для обновления данных) на другой источник:

```
let response = await fetch('https://site.com/service.json', {
  method: 'PATCH',
  headers: {
    'Content-Type': 'application/json'
    'API-Key': 'secret'
  }
});
```

Этот запрос не является простым по трём причинам (достаточно одной):

- Метод `PATCH`
- `Content-Type` не один из: `application/x-www-form-urlencoded`, `multipart/form-data`, `text/plain`.
- Содержит «непростой» заголовок `API-Key`.

Шаг 1 (предзапрос)

Перед тем, как послать такой запрос, браузер самостоятельно генерирует и посылает предзапрос, который выглядит следующим образом:

```
OPTIONS /service.json
Host: site.com
Origin: https://javascript.info
Access-Control-Request-Method: PATCH
Access-Control-Request-Headers: Content-Type,API-Key
```

- Метод: `OPTIONS`.
- Путь – точно такой же, как в основном запросе: `/service.json`.
- Особые заголовки:
  - `Origin` – источник.
  - `Access-Control-Request-Method` – запрашиваемый метод.
  - `Access-Control-Request-Headers` – разделённый запятыми список «непростых» заголовков запроса.

Шаг 2 (ответ сервера на предзапрос)

Сервер должен ответить со статусом 200 и заголовками:

- `Access-Control-Allow-Methods: PATCH`
- `Access-Control-Allow-Headers: Content-Type,API-Key`.

Это разрешит будущую коммуникацию, в противном случае возникает ошибка.

Если сервер ожидает в будущем другие методы и заголовки, то он может в ответе перечислить их все сразу, разрешить заранее, например:

```
200 OK
Access-Control-Allow-Methods: PUT,PATCH,DELETE
Access-Control-Allow-Headers: API-Key,Content-Type,If-Modified-Since,Cache-Control
Access-Control-Max-Age: 86400
```

Теперь, когда браузер видит, что `PATCH` есть в `Access-Control-Allow-Methods`, а `Content-Type,API-Key` – в списке `Access-Control-Allow-Headers`, он посылает наш основной запрос.

Кроме того, ответ на предзапрос кешируется на время, указанное в заголовке `Access-Control-Max-Age` (86400 секунд, один день), так что последующие запросы не вызовут предзапрос. Они будут отосланы сразу при условии, что соответствуют закешированным разрешениям.

Шаг 3 (основной запрос)

Если предзапрос успешен, браузер делает основной запрос. Алгоритм здесь такой же, что и для простых запросов.

Основной запрос имеет заголовок `Origin` (потому что он идёт на другой источник):

```
PATCH /service.json
Host: site.com
Content-Type: application/json
API-Key: secret
Origin: https://javascript.info
```

Шаг 4 (основной ответ)

Сервер не должен забывать о добавлении `Access-Control-Allow-Origin` к ответу на основной запрос. Успешный предзапрос не освобождает от этого:

```
Access-Control-Allow-Origin: https://javascript.info
```

После этого JavaScript может прочитать ответ сервера.

На заметку:

Предзапрос осуществляется «за кулисами», невидимо для JavaScript.

JavaScript получает только ответ на основной запрос или ошибку, если со стороны сервера нет разрешения.

Авторизационные данные

Запрос на другой источник по умолчанию не содержит авторизационных данных (credentials), под которыми здесь понимаются куки и заголовки HTTP-аутентификации.

Это нетипично для HTTP-запросов. Обычно запрос к `http://site.com` сопровождается всеми куки с этого домена. Но запросы на другой источник, сделанные методами JavaScript – исключение.

Например, `fetch('http://another.com')` не посылает никаких куки, даже тех (!), которые принадлежат домену `another.com`.

Почему?

Потому что запрос с авторизационными данными даёт намного больше возможностей, чем без них. Если он разрешён, то это позволяет JavaScript действовать от имени пользователя и получать информацию, используя его авторизационные данные.

Действительно ли сервер настолько доверяет скрипту? Тогда он должен явно разрешить такие запросы при помощи дополнительного заголовка.

Чтобы включить отправку авторизационных данных в `fetch`, нам нужно добавить опцию `credentials: "include"`, вот так:

```
fetch('http://another.com', {
  credentials: "include"
});
```

Теперь `fetch` пошлёт куки с домена `another.com` вместе с нашим запросом на этот сайт.

Если сервер согласен принять запрос *с авторизационными данными*, он должен добавить заголовок `Access-Control-Allow-Credentials: true` к ответу, в дополнение к `Access-Control-Allow-Origin`.

Например:

```
200 OK
Access-Control-Allow-Origin: https://javascript.info
Access-Control-Allow-Credentials: true
```

Пожалуйста, обратите внимание: в `Access-Control-Allow-Origin` запрещено использовать звёздочку `*` для запросов с авторизационными данными. Там должен быть именно источник, как показано выше. Это дополнительная мера безопасности, чтобы гарантировать, что сервер действительно знает, кому он доверяет делать такие запросы.

Итого

С точки зрения браузера запросы к другому источнику бывают двух видов: «простые» и все остальные.

Простые запросы должны удовлетворять следующим условиям:

- Метод: GET, POST или HEAD.
- Заголовки – мы можем установить только:
  - `Accept`
  - `Accept-Language`
  - `Content-Language`
  - `Content-Type` со значением `application/x-www-form-urlencoded`, `multipart/form-data` или `text/plain`.

Основное их отличие заключается в том, что простые запросы с давних времён выполнялись с использованием тегов `<form>` или `<script>`, в то время как непростые долгое время были невозможны для браузеров.

Практическая разница состоит в том, что простые запросы отправляются сразу с заголовком `Origin`, а для других браузер делает предварительный запрос, спрашивая разрешения.

Для простых запросов:

- → Браузер посылает заголовок `Origin` с источником.
- ← Для запросов без авторизационных данных (не отправляются умолчанию) сервер должен установить:
  - `Access-Control-Allow-Origin` в `*` или то же значение, что и `Origin`
- ← Для запросов с авторизационными данными сервер должен установить:
  - `Access-Control-Allow-Origin` в то же значение, что и `Origin`
  - `Access-Control-Allow-Credentials` в `true`

Дополнительно, чтобы разрешить JavaScript доступ к любым заголовкам ответа, кроме `Cache-Control`, `Content-Language`, `Content-Type`, `Expires`, `Last-Modified` или `Pragma`, сервер должен перечислить разрешённые в заголовке `Access-Control-Expose-Headers`.

Для непростых запросов перед основным запросом отправляется предзапрос:

- → Браузер посылает запрос `OPTIONS` на тот же адрес с заголовками:
  - `Access-Control-Request-Method` – содержит запрашиваемый метод,

- `Access-Control-Request-Headers` – перечисляет непростые запрашиваемые заголовки.
- ← Сервер должен ответить со статусом 200 и заголовками:
  - `Access-Control-Allow-Methods` со списком разрешённых методов,
  - `Access-Control-Allow-Headers` со списком разрешённых заголовков,
  - `Access-Control-Max-Age` с количеством секунд для кеширования разрешений
- → Затем отправляется основной запрос, применяется предыдущая «простая» схема.

Задачи

Почему нам нужен Origin?

важность: 5

Как вы, вероятно, знаете, существует HTTP-заголовок `Referer`, который обычно содержит адрес страницы, инициировавшей сетевой запрос.

Например, при запросе (fetch) `http://google.com` с `http://javascript.info/some/url` заголовки выглядят так:

```
Accept: */*
Accept-Charset: utf-8
Accept-Encoding: gzip,deflate,sdch
Connection: keep-alive
Host: google.com
Origin: http://javascript.info
Referer: http://javascript.info/some/url
```

Как вы можете видеть, присутствуют и `Referer`, и `Origin`.

Вопросы:

1. Почему нужен `Origin`, если `Referer` содержит даже больше информации?
2. Возможно ли отсутствие `Referer` или `Origin`, или это неправильно?

К решению

Fetch API

На данный момент мы уже многое знаем про `fetch`.

Давайте рассмотрим оставшуюся часть API, чтобы охватить все возможности.

> **На заметку:**
>
> Заметим: большинство этих возможностей используются редко. Вы можете пропустить эту главу и, несмотря на это, нормально использовать `fetch`.
>
> Тем не менее, полезно знать, что вообще может `fetch`, чтобы, когда появится необходимость, вернуться и прочитать конкретные детали.

Нижеследующий список – это все возможные опции для `fetch` с соответствующими значениями по умолчанию (в комментариях указаны альтернативные значения):

```
let promise = fetch(url, {
  method: "GET", // POST, PUT, DELETE, etc.
  headers: {
    // значение этого заголовка обычно ставится автоматически,
    // в зависимости от тела запроса
    "Content-Type": "text/plain;charset=UTF-8"
  },
  body: undefined // string, FormData, Blob, BufferSource или URLSearchParams
  referrer: "about:client", // или "" для того, чтобы не послать заголовок Referer,
  // или URL с текущего источника
  referrerPolicy: "no-referrer-when-downgrade", // no-referrer, origin, same-origin
  mode: "cors", // same-origin, no-cors
  credentials: "same-origin", // omit, include
  cache: "default", // no-store, reload, no-cache, force-cache или only-if-cached
  redirect: "follow", // manual, error
  integrity: "", // контрольная сумма, например "sha256-abcdef1234567890"
  keepalive: false, // true
  signal: undefined, // AbortController, чтобы прервать запрос
  window: window // null
});
```

Довольно-таки внушительный список, не так ли?

В главе Fetch мы разобрали параметры `method`, `headers` и `body`.

Опция `signal` разъяснена в главе в Fetch: прерывание запроса.

Теперь давайте пройдёмся по оставшимся возможностям.

referrer, referrerPolicy

Данные опции определяют, как `fetch` устанавливает HTTP-заголовок `Referer`.

Обычно этот заголовок ставится автоматически и содержит URL-адрес страницы, с которой пришёл запрос. В большинстве случаев он совсем неважен, в некоторых случаях, с целью большей безопасности, имеет смысл убрать или укоротить его.

Опция `referrer` позволяет установить любой `Referer` в пределах текущего источника или же убрать его.

Чтобы не отправлять `Referer`, нужно указать значением пустую строку:

```
fetch('/page', {
  referrer: "" // не ставить заголовок Referer
});
```

Для того, чтобы установить другой URL-адрес (должен быть с текущего источника):

```
fetch('/page', {
  // предположим, что мы находимся на странице https://javascript.info
  // мы можем установить любое значение Referer при условии, что оно принадлежит те
  referrer: "https://javascript.info/anotherpage"
});
```

Опция `referrerPolicy` устанавливает общие правила для `Referer`.

Выделяется 3 типа запросов:

1. Запрос на тот же источник.
2. Запрос на другой источник.
3. Запрос с HTTPS to HTTP (с безопасного протокола на небезопасный).

В отличие от настройки `referrer`, которая позволяет задачать точное значение `Referer`, настройка `referrerPolicy` сообщает браузеру общие правила, что делать для каждого типа запроса.

Возможные значения описаны в спецификации Referrer Policy :

- **`"no-referrer-when-downgrade"`** – это значение по умолчанию: `Referer` отправляется всегда, если только мы не отправим запрос из HTTPS в HTTP (из более безопасного протокола в менее безопасный).
- **`"no-referrer"`** – никогда не отправлять `Referer`.
- **`"origin"`** – отправлять в `Referer` только текущий источник, а не полный URL-адрес страницы, например, посылать только `http://site.com` вместо `http://site.com/path`.

- **`"origin-when-cross-origin"`** – отправлять полный Referer для запросов в пределах текущего источника, но для запросов на другой источник отправлять только сам источник (как выше).
- **`"same-origin"`** – отправлять полный Referer для запросов в пределах текущего источника, а для запросов на другой источник не отправлять его вообще.
- **`"strict-origin"`** – отправлять только значение источника, не отправлять Referer для HTTPS→HTTP запросов.
- **`"strict-origin-when-cross-origin"`** – для запросов в пределах текущего источника отправлять полный Referer, для запросов на другой источник отправлять только значение источника, в случае HTTPS→HTTP запросов не отправлять ничего.
- **`"unsafe-url"`** – всегда отправлять полный URL-адрес в `Referer`, даже при запросах `HTTPS→HTTP`.

Вот таблица со всеми комбинациями:

Значение	На тот же источник	На другой источник	HTTPS→HTTP
`"no-referrer"`	-	-	-
`"no-referrer-when-downgrade"` или `""` (по умолчанию)	full	full	-
`"origin"`	origin	origin	origin
`"origin-when-cross-origin"`	full	origin	origin
`"same-origin"`	full	-	-
`"strict-origin"`	origin	origin	-
`"strict-origin-when-cross-origin"`	full	origin	-
`"unsafe-url"`	full	full	full

Допустим, у нас есть админка со структурой URL, которая не должна стать известной снаружи сайта.

Если мы отправляем запрос `fetch`, то по умолчанию он всегда отправляет заголовок `Referer` с полным URL-адресом нашей админки (исключение – это когда мы делаем запрос от HTTPS в HTTP, в таком случае `Referer` не будет отправляться).

Например, `Referer: https://javascript.info/admin/secret/paths`.

Если мы хотим, чтобы другие сайты получали только источник, но не URL-путь, это сделает такая настройка:

```
fetch('https://another.com/page', {
  // ...
  referrerPolicy: "origin-when-cross-origin" // Referer: https://javascript.info
});
```

Мы можем поставить её во все вызовы `fetch`, возможно, интегрировать в JavaScript-библиотеку нашего проекта, которая делает все запросы и внутри использует `fetch`.

Единственным отличием в поведении будет то, что для всех запросов на другой источник `fetch` будет посылать только источник в заголовке `Referer` (например, `https://javascript.info`, без пути). А для запросов на наш источник мы продолжим получать полный `Referer` (это может быть полезно для отладки).

Политика установки Referer (Referrer Policy) – не только для `fetch`

Политика установки Referer, описанная в спецификации Referrer Policy, существует не только для `fetch`, она более глобальная.

В частности, можно поставить политику по умолчанию для всей страницы, используя HTTP-заголовок `Referrer-Policy`, или на уровне ссылки `<a rel="noreferrer">`.

mode

Опция `mode` – это защита от нечаянной отправки запроса на другой источник:

- **`"cors"`** – стоит по умолчанию, позволяет делать такие запросы так, как описано в Fetch: запросы на другие сайты,
- **`"same-origin"`** – запросы на другой источник запрещены,
- **`"no-cors"`** – разрешены только простые запросы на другой источник.

Эта опция может пригодиться, если URL-адрес для `fetch` приходит от третьей стороны, и нам нужен своего рода «глобальный выключатель» для запросов на другие источники.

credentials

Опция `credentials` указывает, должен ли `fetch` отправлять куки и авторизационные заголовки HTTP вместе с запросом.

- **`"same-origin"`** – стоит по умолчанию, не отправлять для запросов на другой источник,

- `"include"` – отправлять всегда, но при этом необходим заголовок `Access-Control-Allow-Credentials` в ответе от сервера, чтобы JavaScript получил доступ к ответу сервера, об этом говорилось в главе Fetch: запросы на другие сайты,
- `"omit"` – не отправлять ни при каких обстоятельствах, даже для запросов, сделанных в пределах текущего источника.

cache

По умолчанию `fetch` делает запросы, используя стандартное HTTP-кеширование. То есть, учитывается заголовки `Expires`, `Cache-Control`, отправляется `If-Modified-Since` и так далее. Так же, как и обычные HTTP-запросы.

Настройка `cache` позволяет игнорировать HTTP-кеш или же настроить его использование:

- `"default"` – `fetch` будет использовать стандартные правила и заголовки HTTP кеширования,
- `"no-store"` – полностью игнорировать HTTP-кеш, этот режим становится режимом по умолчанию, если присутствуют такие заголовки как `If-Modified-Since`, `If-None-Match`, `If-Unmodified-Since`, `If-Match`, или `If-Range`,
- `"reload"` – не брать результат из HTTP-кеша (даже при его присутствии), но сохранить ответ в кеше (если это дозволено заголовками ответа);
- `"no-cache"` – в случае, если существует кешированный ответ – создать условный запрос, в противном же случае – обычный запрос. Сохранить ответ в HTTP-кеше,
- `"force-cache"` – использовать ответ из HTTP-кеша, даже если он устаревший. Если же ответ в HTTP-кеше отсутствует, сделать обычный HTTP-запрос, действовать как обычно,
- `"only-if-cached"` – использовать ответ из HTTP-кеша, даже если он устаревший. Если же ответ в HTTP-кеше отсутствует, то выдаётся ошибка. Это работает, только когда `mode` установлен в `"same-origin"`.

redirect

Обычно `fetch` прозрачно следует HTTP-редиректам, таким как 301, 302 и так далее.

Это можно поменять при помощи опции `redirect`:

- `"follow"` – стоит по умолчанию, следовать HTTP-редиректам,
- `"error"` – ошибка в случае HTTP-редиректа,

- `"manual"` – не следовать HTTP-редиректу, но установить адрес редиректа в `response.url`, а `response.redirected` будет иметь значение `true`, чтобы мы могли сделать перенаправление на новый адрес вручную.

integrity

Опция `integrity` позволяет проверить, соответствует ли ответ известной заранее контрольной сумме.

Как описано в спецификации, поддерживаемыми хеш-функциями являются SHA-256, SHA-384 и SHA-512. В зависимости от браузера, могут быть и другие.

Например, мы скачиваем файл, и мы точно знаем, что его контрольная сумма по алгоритму SHA-256 равна «abcdef» (разумеется, настоящая контрольная сумма будет длиннее).

Мы можем добавить это в настройку `integrity` вот так:

```
fetch('http://site.com/file', {
  integrity: 'sha256-abcdef'
});
```

Затем `fetch` самостоятельно вычислит SHA-256 и сравнит его с нашей строкой. В случае несоответствия будет ошибка.

keepalive

Опция `keepalive` указывает на то, что запрос может «пережить» страницу, которая его отправила.

Например, мы собираем статистические данные о том, как посетитель ведёт себя на нашей странице (на что он кликает, части страницы, которые он просматривает), для анализа и улучшения интерфейса.

Когда посетитель покидает нашу страницу – мы хотим сохранить собранные данные на нашем сервере.

Для этого мы можем использовать событие `window.onunload`:

```
window.onunload = function() {
  fetch('/analytics', {
    method: 'POST',
    body: "statistics",
    keepalive: true
  });
};
```

Обычно, когда документ выгружается, все связанные с ним сетевые запросы прерываются. Но настройка `keepalive` указывает браузеру выполнять запрос в фоновом режиме даже после того, как пользователь покидает страницу. Поэтому эта опция обязательна, чтобы такой запрос удался.

У неё есть ряд ограничений:

- Мы не можем посылать мегабайты: лимит тела для запроса с `keepalive` – 64кб.
  - Если мы собираем больше данных, можем отправлять их регулярно, «пакетами», тогда на момент последнего запроса в `onunload` их останется немного.
  - Этот лимит распространяется на все запросы с `keepalive`. То есть, мы не можем его обойти, послав 100 запросов одновременно – каждый по 64Кбайт.
- Мы не сможем обработать ответ от сервера, если запрос сделан при `onunload`: в тот момент документ уже выгружен, его функции не сработают.
  - Обычно сервер посылает пустой ответ на такие запросы, так что это не является проблемой.

Объекты URL

Встроенный класс URL предоставляет удобный интерфейс для создания и разбора URL-адресов.

Нет сетевых методов, которые требуют именно объект `URL`, обычные строки вполне подходят. Так что, технически, мы не обязаны использовать `URL`. Но иногда он может быть весьма удобным.

Создание URL

Синтаксис создания нового объекта `URL`:

```
new URL(url, [base])
```

- **`url`** – полный URL-адрес или только путь, если указан второй параметр,
- **`base`** – необязательный «базовый» URL: если указан и аргумент `url` содержит только путь, то адрес будет создан относительно него (пример ниже).

Например:

```
let url = new URL('https://javascript.info/profile/admin');
```

Эти два URL одинаковы:

```
let url1 = new URL('https://javascript.info/profile/admin');
let url2 = new URL('/profile/admin', 'https://javascript.info');

alert(url1); // https://javascript.info/profile/admin
alert(url2); // https://javascript.info/profile/admin
```

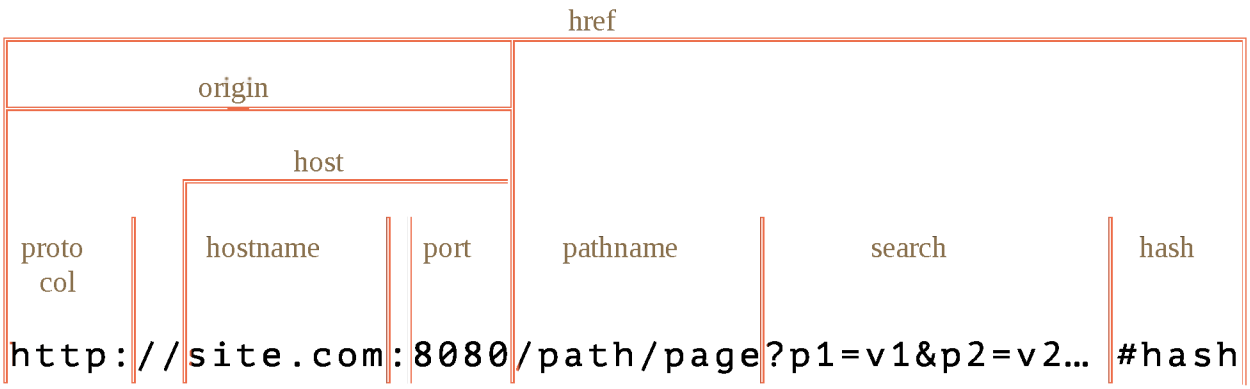
Можно легко создать новый URL по пути относительно существующего URL-адреса:

```
let url = new URL('https://javascript.info/profile/admin');
let newUrl = new URL('tester', url);

alert(newUrl); // https://javascript.info/profile/tester
```

Объект `URL` даёт доступ к компонентам URL, поэтому это отличный способ «разобрать» URL-адрес, например:

```
let url = new URL('https://javascript.info/url');

alert(url.protocol); // https:
alert(url.host);     // javascript.info
alert(url.pathname); // /url
```

Вот шпаргалка по компонентам URL:



- `href` это полный URL-адрес, то же самое, что `url.toString()`

- `protocol` – протокол, заканчивается символом двоеточия `:`
- `search` строка параметров, начинается с вопросительного знака `?`
- `hash` начинается с символа `#`
- также есть свойства `user` и `password`, если используется HTTP-аутентификация: `http://login:password@site.com` (не нарисованы сверху, так как редко используются).

> **Можно передавать объекты `URL` в сетевые методы (и большинство других) вместо строк**
>
> Мы можем использовать объект `URL` в методах `fetch` или `XMLHttpRequest` и почти во всех других, где ожидается URL-строка.
>
> Вообще, объект `URL` можно передавать почти куда угодно вместо строки, так как большинство методов сконвертируют объект в строку, при этом он станет строкой с полным URL-адресом.

SearchParams «?...»

Допустим, мы хотим создать URL-адрес с заданными параметрами, например, `https://google.com/search?query=JavaScript`.

Мы можем указать их в строке:

```
new URL('https://google.com/search?query=JavaScript')
```

...Но параметры должны быть правильно закодированы, чтобы они могли содержать не-латинские буквы, пробелы и т.п. (об этом подробнее далее).

Так что для этого есть свойство `url.searchParams` – объект типа URLSearchParams.

Он предоставляет удобные методы для работы с параметрами:

- **`append(name, value)`** – добавить параметр по имени,
- **`delete(name)`** – удалить параметр по имени,
- **`get(name)`** – получить параметр по имени,
- **`getAll(name)`** – получить все параметры с одинаковым именем `name` (такое возможно, например: `?user=John&user=Pete`),
- **`has(name)`** – проверить наличие параметра по имени,
- **`set(name, value)`** – задать/заменить параметр,
- **`sort()`** – отсортировать параметры по имени, используется редко,

- …и является перебираемым, аналогично `Map`.

Пример добавления параметров, содержащих пробелы и знаки препинания:

```
let url = new URL('https://google.com/search');
url.searchParams.set('q', 'test me!'); // добавим параметр, содержащий пробел и !

alert(url); // https://google.com/search?q=test+me%21

url.searchParams.set('tbs', 'qdr:y'); // параметр с двоеточием :

// параметры автоматически кодируются
alert(url); // https://google.com/search?query=test+me%21&tbs=qdr%3Ay

// перебрать параметры (в исходном виде)
for(let [name, value] of url.searchParams) {
  alert(`${name}=${value}`); // q=test me!, далее tbs=qdr:y
}
```

Кодирование

Существует стандарт RFC3986, который определяет список разрешённых и запрещённых символов в URL.

Запрещённые символы, например, нелатинские буквы и пробелы, должны быть закодированы – заменены соответствующими кодами UTF-8 с префиксом `%`, например: `%20` (исторически сложилось так, что пробел в URL-адресе можно также кодировать симоволом `+`, но это исключение).

К счастью, объекты `URL` делают всё это автоматически. Мы просто указываем параметры в обычном, незакодированном, виде, а затем конвертируем `URL` в строку:

```
let url = new URL('https://ru.wikipedia.org/wiki/Тест');

url.searchParams.set('key', 'ъ');
alert(url); //https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82?key=%D1%8A
```

Как видно, слово `Тест` в пути URL-адреса и буква `ъ` в параметре закодированы.

URL стал длиннее, так как каждая кириллическая буква представляется двумя байтами в кодировке UTF-8.

Кодирование в строках

Раньше, до того как появились объекты `URL`, люди использовали для URL-адресов обычные строки.

Сейчас `URL` часто удобнее, но строки всё ещё можно использовать. Во многих случаях код с ними короче.

Однако, если мы используем строку, то надо самим позаботиться о кодировании специальных символов.

Для этого есть встроенные функции:

- encodeURI – кодирует URL-адрес целиком.
- decodeURI – декодирует URL-адрес целиком.
- encodeURIComponent – кодирует компонент URL, например, параметр, хеш, имя пути и т.п.
- decodeURIComponent – декодирует компонент URL.

Возникает естественный вопрос: «Какая разница между `encodeURIComponent` и `encodeURI`? Когда использовать одну и другую функцию?»

Это легко понять, если мы посмотрим на URL-адрес, разбитый на компоненты на рисунке выше:

```
http://site.com:8080/path/page?p1=v1&p2=v2#hash
```

Как мы видим, в URL-адресе разрешены символы `:`, `?`, `=`, `&`, `#`.

…С другой стороны, если взглянуть на один компонент, например, URL-параметр, то в нём такие символы должны быть закодированы, чтобы не поломать форматирование.

- `encodeURI` кодирует только символы, полностью запрещённые в URL.
- `encodeURIComponent` кодирует эти же символы плюс, в дополнение к ним, символы `#`, `$`, `&`, `+`, `,`, `/`, `:`, `;`, `=`, `?` и `@`.

Так что для URL целиком можно использовать `encodeURI`:

```
let url = encodeURI('http://site.com/привет');

alert(url); // http://site.com/%D0%BF%D1%80%D0%B8%D0%B2%D0%B5%D1%82
```

…А для параметров лучше будет взять `encodeURIComponent`:

```
let music = encodeURIComponent('Rock&Roll');

let url = `https://google.com/search?q=${music}`;
alert(url); // https://google.com/search?q=Rock%26Roll
```

Сравните с `encodeURI`:

```
let music = encodeURI('Rock&Roll');

let url = `https://google.com/search?q=${music}`;
alert(url); // https://google.com/search?q=Rock&Roll
```

Как видим, функция `encodeURI` не закодировала символ `&`, который является разрешённым в составе полного URL-адреса.

Но внутри параметра поиска символ `&` должен быть закодирован, в противном случае мы получим `q=Rock&Roll`, что значит `q=Rock` плюс непонятный параметр `Roll`. Не то, что предполагалось.

Чтобы правильно вставить параметр поиска в строку URL, мы должны использовать для него только `encodeURIComponent`. Наиболее безопасно кодировать и имя, и значение, за исключением случаев, когда мы абсолютно уверены в том, что они содержат только разрешённые символы.

Разница в кодировании с `URL`

Классы URL и URLSearchParams базируются на последней спецификации URI, описывающей устройство адресов: RFC3986 , в то время как функции `encode*` – на устаревшей версии стандарта RFC2396 .

Различий мало, но они есть, например, по-разному кодируются адреса IPv6:

```
// допустимый URL-адрес IPv6
let url = 'http://[2607:f8b0:4005:802::1007]/';

alert(encodeURI(url)); // http://%5B2607:f8b0:4005:802::1007%5D/
alert(new URL(url)); // http://[2607:f8b0:4005:802::1007]/
```

Как мы видим, функция `encodeURI` заменила квадратные скобки `[...]`, сделав адрес некорректным. Причина: URL-адреса IPv6 не существовали в момент создания стандарта RFC2396 (август 1998).

Тем не менее, такие случаи редки. По большей части функции `encode*` работают хорошо.

XMLHttpRequest

`XMLHttpRequest` – это встроенный в браузер объект, который даёт возможность делать HTTP-запросы к серверу без перезагрузки страницы.

Несмотря на наличие слова «XML» в названии, XMLHttpRequest может работать с любыми данными, а не только с XML. Мы можем загружать/скачивать файлы, отслеживать прогресс и многое другое.

На сегодняшний день не обязательно использовать `XMLHttpRequest`, так как существует другой, более современный метод `fetch`.

В современной веб-разработке `XMLHttpRequest` используется по трём причинам:

1. По историческим причинам: существует много кода, использующего `XMLHttpRequest`, который нужно поддерживать.
2. Необходимость поддерживать старые браузеры и нежелание использовать полифилы (например, чтобы уменьшить количество кода).
3. Потребность в функционале, который `fetch` пока что не может предоставить, к примеру, отслеживание прогресса отправки на сервер.

Что-то из этого списка звучит знакомо? Если да, тогда вперёд, приятного знакомства с `XMLHttpRequest`. Если же нет, возможно, имеет смысл изучать сразу Fetch.

Основы

XMLHttpRequest имеет два режима работы: синхронный и асинхронный.

Сначала рассмотрим асинхронный, так как в большинстве случаев используется именно он.

Чтобы сделать запрос, нам нужно выполнить три шага:

1. Создать `XMLHttpRequest`.

```
let xhr = new XMLHttpRequest(); // у конструктора нет аргументов
```

2. Инициализировать его.

```
xhr.open(method, URL, [async, user, password])
```

   Этот метод обычно вызывается сразу после `new XMLHttpRequest`. В него передаются основные параметры запроса:

   - `method` – HTTP-метод. Обычно это `"GET"` или `"POST"`.
   - `URL` – URL, куда отправляется запрос: строка, может быть и объект URL.
   - `async` – если указать `false`, тогда запрос будет выполнен синхронно, это мы рассмотрим чуть позже.
   - `user`, `password` – логин и пароль для базовой HTTP-авторизации (если требуется).

   Заметим, что вызов `open`, вопреки своему названию, не открывает соединение. Он лишь конфигурирует запрос, но непосредственно отсылается запрос только лишь после вызова `send`.

3. Послать запрос.

```
xhr.send([body])
```

   Этот метод устанавливает соединение и отсылает запрос к серверу. Необязательный параметр `body` содержит тело запроса.

Некоторые типы запросов, такие как `GET`, не имеют тела. А некоторые, как, например, `POST`, используют `body`, чтобы отправлять данные на сервер. Мы позже увидим примеры.

4. Слушать события на `xhr`, чтобы получить ответ.

   Три наиболее используемых события:

   - `load` – происходит, когда получен какой-либо ответ, включая ответы с HTTP-ошибкой, например 404.
   - `error` – когда запрос не может быть выполнен, например, нет соединения или невалидный URL.
   - `progress` – происходит периодически во время загрузки ответа, сообщает о прогрессе.

```
xhr.onload = function() {
  alert(`Загружено: ${xhr.status} ${xhr.response}`);
};

xhr.onerror = function() { // происходит, только когда запрос совсем не получилос
  alert(`Ошибка соединения`);
};

xhr.onprogress = function(event) { // запускается периодически
  // event.loaded - количество загруженных байт
  // event.lengthComputable = равно true, если сервер присылает заголовок Content
  // event.total - количество байт всего (только если lengthComputable равно true
  alert(`Загружено ${event.loaded} из ${event.total}`);
};
```

Вот полный пример. Код ниже загружает `/article/xmlhttprequest/example/load` с сервера и сообщает о прогрессе:

```
// 1. Создаём новый XMLHttpRequest-объект
let xhr = new XMLHttpRequest();

// 2. Настраиваем его: GET-запрос по URL /article/.../load
xhr.open('GET', '/article/xmlhttprequest/example/load');

// 3. Отсылаем запрос
xhr.send();

// 4. Этот код сработает после того, как мы получим ответ сервера
xhr.onload = function() {
  if (xhr.status != 200) { // анализируем HTTP-статус ответа, если статус не 200, т
    alert(`Ошибка ${xhr.status}: ${xhr.statusText}`); // Например, 404: Not Found
  } else { // если всё прошло гладко, выводим результат
```

```
    alert(`Готово, получили ${xhr.response.length} байт`); // response -- это ответ
  }
};

xhr.onprogress = function(event) {
  if (event.lengthComputable) {
    alert(`Получено ${event.loaded} из ${event.total} байт`);
  } else {
    alert(`Получено ${event.loaded} байт`); // если в ответе нет заголовка Content-
  }

};

xhr.onerror = function() {
  alert("Запрос не удался");
};
```

После ответа сервера мы можем получить результат запроса в следующих свойствах `xhr`:

`status`

Код состояния HTTP (число): `200`, `404`, `403` и так далее, может быть `0` в случае, если ошибка не связана с HTTP.

`statusText`

Сообщение о состоянии ответа HTTP (строка): обычно `OK` для `200`, `Not Found` для `404`, `Forbidden` для `403`, и так далее.

`response` (в старом коде может встречаться как `responseText`)

Тело ответа сервера.

Мы можем также указать таймаут – промежуток времени, который мы готовы ждать ответ:

```
xhr.timeout = 10000; // таймаут указывается в миллисекундах, т.е. 10 секунд
```

Если запрос не успевает выполниться в установленное время, то он прерывается, и происходит событие `timeout`.

URL с параметрами

Чтобы добавить к URL параметры, вида `?name=value`, и корректно закодировать их, можно использовать объект URL:

```
let url = new URL('https://google.com/search');
url.searchParams.set('q', 'test me!');

// параметр 'q' закодирован
xhr.open('GET', url); // https://google.com/search?q=test+me%21
```

Тип ответа

Мы можем использовать свойство `xhr.responseType`, чтобы указать ожидаемый тип ответа:

- `""` (по умолчанию) – строка,
- `"text"` – строка,
- `"arraybuffer"` – `ArrayBuffer` (для бинарных данных, смотрите в ArrayBuffer, бинарные массивы),
- `"blob"` – `Blob` (для бинарных данных, смотрите в Blob),
- `"document"` – XML-документ (может использовать XPath и другие XML-методы),
- `"json"` – JSON (парсится автоматически).

К примеру, давайте получим ответ в формате JSON:

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/article/xmlhttprequest/example/json');

xhr.responseType = 'json';

xhr.send();

// тело ответа {"сообщение": "Привет, мир!"}
xhr.onload = function() {
  let responseObj = xhr.response;
  alert(responseObj.message); // Привет, мир!
};
```

На заметку:

В старом коде вы можете встретить свойства `xhr.responseText` и даже `xhr.responseXML`.

Они существуют по историческим причинам, раньше с их помощью получали строки или XML-документы. Сегодня следует устанавливать желаемый тип объекта в `xhr.responseType` и получать `xhr.response`, как показано выше.

Состояния запроса

У `XMLHttpRequest` есть состояния, которые меняются по мере выполнения запроса. Текущее состояние можно посмотреть в свойстве `xhr.readyState`.

Список всех состояний, указанных в спецификации :

```
UNSENT = 0; // исходное состояние
OPENED = 1; // вызван метод open
HEADERS_RECEIVED = 2; // получены заголовки ответа
LOADING = 3; // ответ в процессе передачи (данные частично получены)
DONE = 4; // запрос завершён
```

Состояния объекта `XMLHttpRequest` меняются в таком порядке: `0` → `1` → `2` → `3` → … → `3` → `4`. Состояние `3` повторяется каждый раз, когда получена часть данных.

Изменения в состоянии объекта запроса генерируют событие `readystatechange`:

```
xhr.onreadystatechange = function() {
  if (xhr.readyState == 3) {
    // загрузка
  }
  if (xhr.readyState == 4) {
    // запрос завершён
  }
};
```

Вы можете наткнуться на обработчики события `readystatechange` в очень старом коде, так уж сложилось исторически, когда-то не было событий `load` и других. Сегодня из-за существования событий `load/error/progress` можно сказать, что событие `readystatechange` «морально устарело».

Отмена запроса

Если мы передумали делать запрос, можно отменить его вызовом `xhr.abort()`:

```
xhr.abort(); // завершить запрос
```

При этом генерируется событие `abort`, а `xhr.status` устанавливается в `0`.

Синхронные запросы

Если в методе `open` третий параметр `async` установлен на `false`, запрос выполняется синхронно.

Другими словами, выполнение JavaScript останавливается на `send()` и возобновляется после получения ответа. Так ведут себя, например, функции `alert` или `prompt`.

Вот пример переписанный пример, с параметром `async`, равным `false`:

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/article/xmlhttprequest/hello.txt', false);

try {
  xhr.send();
  if (xhr.status != 200) {
    alert(`Ошибка ${xhr.status}: ${xhr.statusText}`);
  } else {
    alert(xhr.response);
  }
} catch(err) { // для отлова ошибок используем конструкцию try...catch вместо onerr
  alert("Запрос не удался");
}
```

Выглядит, может быть, и неплохо, но синхронные запросы используются редко, так как они блокируют выполнение JavaScript до тех пор, пока загрузка не завершена. В некоторых браузерах нельзя прокручивать страницу, пока идёт синхронный запрос. Ну а если же синхронный запрос по какой-то причине выполняется слишком долго, браузер предложит закрыть «зависшую» страницу.

Многие продвинутые возможности `XMLHttpRequest`, такие как выполнение запроса на другой домен или установка таймаута, недоступны для синхронных

запросов. Также, как вы могли заметить, ни о какой индикации прогресса речь тут не идёт.

Из-за всего этого синхронные запросы используют очень редко. Мы более не будем рассматривать их.

HTTP-заголовки

`XMLHttpRequest` умеет как указывать свои заголовки в запросе, так и читать присланные в ответ.

Для работы с HTTP-заголовками есть 3 метода:

`setRequestHeader(name, value)`

Устанавливает заголовок запроса с именем `name` и значением `value`.

Например:

```
xhr.setRequestHeader('Content-Type', 'application/json');
```

Ограничения на заголовки

Некоторые заголовки управляются исключительно браузером, например `Referer` или `Host`, а также ряд других. Полный список тут.

`XMLHttpRequest` не разрешено изменять их ради безопасности пользователей и для обеспечения корректности HTTP-запроса.

Поставленный заголовок нельзя снять

Ещё одной особенностью `XMLHttpRequest` является то, что отменить `setRequestHeader` невозможно.

Если заголовок определён, то его нельзя снять. Повторные вызовы лишь добавляют информацию к заголовку, а не перезаписывают его.

Например:

```
xhr.setRequestHeader('X-Auth', '123');
xhr.setRequestHeader('X-Auth', '456');

// заголовок получится такой:
// X-Auth: 123, 456
```

`getResponseHeader(name)`

Возвращает значение заголовка ответа `name` (кроме `Set-Cookie` и `Set-Cookie2`).

Например:

```
xhr.getResponseHeader('Content-Type')
```

`getAllResponseHeaders()`

Возвращает все заголовки ответа, кроме `Set-Cookie` и `Set-Cookie2`.

Заголовки возвращаются в виде единой строки, например:

```
Cache-Control: max-age=31536000
Content-Length: 4260
Content-Type: image/png
Date: Sat, 08 Sep 2012 16:53:16 GMT
```

Между заголовками всегда стоит перевод строки в два символа `"\r\n"` (независимо от ОС), так что мы может легко разделить их в отдельные заголовки. Значение заголовка всегда отделено двоеточием с пробелом `": "`. Этот формат задан стандартом.

Таким образом, если хочется получить объект с парами заголовок-значение, нам нужно задействовать немного JS.

Вот так (предполагается, что если два заголовка имеют одинаковое имя, то последний перезаписывает предыдущий):

```
let headers = xhr
  .getAllResponseHeaders()
  .split('\r\n')
  .reduce((result, current) => {
    let [name, value] = current.split(': ');
    result[name] = value;
    return result;
  }, {});

// headers['Content-Type'] = 'image/png'
```

POST, FormData

Чтобы сделать POST-запрос, мы можем использовать встроенный объект FormData .

Синтаксис:

```
let formData = new FormData([form]); // создаём объект, по желанию берём данные фор
formData.append(name, value); // добавляем поле
```

Мы создаём объект, при желании указываем, из какой формы `form` взять данные, затем, если нужно, с помощью метода `append` добавляем дополнительные поля, после чего:

1. `xhr.open('POST', ...)` – создаём `POST`-запрос.
2. `xhr.send(formData)` – отсылаем форму серверу.

Например:

```
<form name="person">
  <input name="name" value="Петя">
  <input name="surname" value="Васечкин">
</form>

<script>
  // заполним FormData данными из формы
  let formData = new FormData(document.forms.person);

  // добавим ещё одно поле
  formData.append("middle", "Иванович");

  // отправим данные
  let xhr = new XMLHttpRequest();
  xhr.open("POST", "/article/xmlhttprequest/post/user");
  xhr.send(formData);

  xhr.onload = () => alert(xhr.response);
</script>
```

Обычно форма отсылается в кодировке `multipart/form-data`.

Если нам больше нравится формат JSON, то используем `JSON.stringify` и отправляем данные как строку.

Важно не забыть поставить соответствующий заголовок `Content-Type: application/json`, многие серверные фреймворки автоматически декодируют JSON при его наличии:

```
let xhr = new XMLHttpRequest();

let json = JSON.stringify({
```

```
  name: "Вася",
  surname: "Петров"
});

xhr.open("POST", '/submit')
xhr.setRequestHeader('Content-type', 'application/json; charset=utf-8');

xhr.send(json);
```

Метод `.send(body)` весьма всеяден. Он может отправить практически что угодно в `body`, включая объекты типа `Blob` и `BufferSource`.

Прогресс отправки

Событие `progress` срабатывает только на стадии загрузки ответа с сервера.

А именно: если мы отправляем что-то через `POST`-запрос, `XMLHttpRequest` сперва отправит наши данные (тело запроса) на сервер, а потом загрузит ответ сервера. И событие `progress` будет срабатывать только во вовремя загрузки ответа.

Если мы отправляем что-то большое, то нас гораздо больше интересует прогресс отправки данных на сервер. Но `xhr.onprogress` тут не поможет.

Существует другой объект, без методов, только для отслеживания событий отправки: `xhr.upload`.

Он генерирует события, похожие на события `xhr`, но только во время отправки данных на сервер:

- `loadstart` – начало загрузки данных.
- `progress` – генерируется переодически во время отправки на сервер.
- `abort` – загрузка прервана.
- `error` – ошибка, не связанная с HTTP.
- `load` – загрузка успешно завершена.
- `timeout` – вышло время, отведённое на загрузку (при установленном свойстве `timeout`).
- `loadend` – загрузка завершена, вне зависимости от того, как – успешно или нет.

Примеры обработчиков для этих событий:

```
xhr.upload.onprogress = function(event) {
  alert(`Отправлено ${event.loaded} из ${event.total} байт`);
};
```

```
xhr.upload.onload = function() {
  alert(`Данные успешно отправлены.`);
};

xhr.upload.onerror = function() {
  alert(`Произошла ошибка во время отправки: ${xhr.status}`);
};
```

Пример из реальной жизни: загрузка файла на сервер с индикацией прогресса:

```
<input type="file" onchange="upload(this.files[0])">

<script>
function upload(file) {
  let xhr = new XMLHttpRequest();

  // отслеживаем процесс отправки
  xhr.upload.onprogress = function(event) {
    console.log(`Отправлено ${event.loaded} из ${event.total}`);
  };

  // Ждём завершения: неважно, успешного или нет
  xhr.onloadend = function() {
    if (xhr.status == 200) {
      console.log("Успех");
    } else {
      console.log("Ошибка " + this.status);
    }
  };

  xhr.open("POST", "/article/xmlhttprequest/post/upload");
  xhr.send(file);
}
</script>
```

Запросы на другой источник

`XMLHttpRequest` может осуществлять запросы на другие сайты, используя ту же политику CORS, что и fetch.

Точно так же, как и при работе с `fetch`, по умолчанию на другой источник не отсылаются куки и заголовки HTTP-авторизации. Чтобы это изменить, установите `xhr.withCredentials` в `true`:

```
let xhr = new XMLHttpRequest();
xhr.withCredentials = true;
```

```
xhr.open('POST', 'http://anywhere.com/request');
...
```

Детали по заголовкам, которые при этом необходимы, смотрите в главе fetch.

Итого

Типичный код GET-запроса с использованием `XMLHttpRequest`:

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/my/url');

xhr.send();

xhr.onload = function() {
  if (xhr.status != 200) { // HTTP ошибка?
    // обработаем ошибку
    alert( 'Ошибка: ' + xhr.status);
    return;
  }

  // получим ответ из xhr.response
};

xhr.onprogress = function(event) {
  // выведем прогресс
  alert(`Загружено ${event.loaded} из ${event.total}`);
};

xhr.onerror = function() {
  // обработаем ошибку, не связанную с HTTP (например, нет соединения)
};
```

Событий на самом деле больше, в современной спецификации они все перечислены в том порядке, в каком генерируются во время запроса:

- `loadstart` – начало запроса.
- `progress` – прибыла часть данных ответа, тело ответа полностью на данный момент можно получить из свойства `responseText`.
- `abort` – запрос был прерван вызовом `xhr.abort()`.
- `error` – произошла ошибка соединения, например неправильное доменное имя. Событие не генерируется для HTTP-ошибок как, например, 404.
- `load` – запрос успешно завершён.

- `timeout` – запрос был отменён по причине истечения отведённого для него времени (происходит, только если был установлен таймаут).
- `loadend` – срабатывает после `load`, `error`, `timeout` или `abort`.

События `error`, `abort`, `timeout` и `load` взаимно исключают друг друга – может произойти только одно из них.

Наиболее часто используют события завершения загрузки (`load`), ошибки загрузки (`error`), или мы можем использовать единый обработчик `loadend` для всего и смотреть в свойствах объекта запроса `xhr` детали произошедшего.

Также мы уже видели событие: `readystatechange`. Исторически оно появилось одним из первых, даже раньше, чем была составлена спецификация. Сегодня нет необходимости использовать его, так как оно может быть заменено современными событиями, но на него можно часто наткнуться в старом коде.

Если же нам нужно следить именно за процессом отправки данных на сервер, тогда можно использовать те же события, но для объекта `xhr.upload`.

Возобновляемая загрузка файлов

При помощи `fetch` достаточно просто отправить файл на сервер.

Но как возобновить загрузку, если соединение прервалось? Для этого нет готовой настройки, но у нас есть все средства, чтобы решить эту задачу самостоятельно.

Возобновляемая загрузка должна сопровождаться индикацией прогресса, ведь, скорее всего, нам нужно отправлять большие файлы. Поскольку `fetch` не позволяет отслеживать прогресс отправки, то мы будем использовать XMLHttpRequest.

Не очень полезное событие progress

Чтобы возобновить отправку, нам нужно знать, какая часть файла была успешно передана до того, как соединение прервалось.

Можно установить обработчик `xhr.upload.onprogress`, чтобы отслеживать процесс загрузки, но, к сожалению, это бесполезно, так как этот обработчик вызывается, только когда данные *отправляются*, но были ли они получены сервером? Браузер этого не знает.

Возможно, отправленные данные оказались в буфере прокси-сервера локальной сети или удалённый сервер просто отключился и не смог принять

их, или данные потерялись где-то по пути при разрыве соединения и так и не достигли пункта назначения.

В общем, событие `progress` подходит только для того, чтобы показывать красивый индикатор загрузки, не более.

Для возобновления же загрузки нужно *точно* знать, сколько байт было получено сервером. И только сам сервер может это сказать, поэтому будем делать для этого отдельный запрос.

Алгоритм

1. Во-первых, создадим уникальный идентификатор для файла, который собираемся загружать:

   ```
   let fileId = file.name + '-' + file.size + '-' + +file.lastModifiedDate;
   ```

   Это нужно, чтобы при возобновлении загрузки серверу было понятно, какой файл мы продолжаем загружать.

   Если имя или размер или дата модификация файла изменятся, то у него уже будет другой `fileId`.

2. Далее, посылаем запрос к серверу с просьбой указать количество уже полученных байтов:

   ```
   let response = await fetch('status', {
     headers: {
       'X-File-Id': fileId
     }
   });

   // сервер получил столько-то байтов
   let startByte = +await response.text();
   ```

   Предполагается, что сервер учитывает загружаемые файлы с помощью заголовка `X-File-Id`. Это на стороне сервера должно быть реализовано.

   Если файл серверу неизвестен, то он должен ответить `0`.

3. Затем мы можем использовать метод `slice` объекта `Blob`, чтобы отправить данные, начиная со `startByte` байта:

   ```
   xhr.open("POST", "upload", true);

   // Идентификатор файла, чтобы сервер знал, что мы загружаем
   ```

```
xhr.setRequestHeader('X-File-Id', fileId);

// Номер байта, начиная с которого мы будем отправлять данные.
// Таким образом, сервер поймёт, с какого момента мы возобновляем загрузку
xhr.setRequestHeader('X-Start-Byte', startByte);

xhr.upload.onprogress = (e) => {
  console.log(`Uploaded ${startByte + e.loaded} of ${startByte + e.total}`);
};

// файл file может быть взят из input.files[0] или другого источника
xhr.send(file.slice(startByte));
```

Здесь мы посылаем серверу и идентификатор файла в заголовке `X-File-Id`, чтобы он знал, что мы загружаем, и номер стартового байта в заголовке `X-Start-Byte`, чтобы он понял, что мы продолжаем отправку, а не начинаем её с нуля.

Сервер должен проверить информацию на своей стороне, и если обнаружится, что такой файл уже когда-то загружался, и его текущий размер равен значению из заголовка `X-Start-Byte`, то вновь принимаемые данные добавлять в этот файл.

Ниже представлен демо-код как для сервера (Node.js), так и для клиента.

Пример работает только частично на этом сайте, так как Node.js здесь располагается за другим веб-сервером Nginx, который сохраняет в своём буфере все загружаемые файлы и передаёт их дальше в Node.js только после завершения загрузки.

Но вы можете скачать код и запустить его локально, чтобы увидеть полный пример в действии:

https://plnkr.co/edit/Swk7ixpHThKt5pcRUDRc?p=preview

Как видим, современные методы работы с сетью очень близки по своим возможностям к файловым менеджерам – контроль заголовков, индикация прогресса загрузки, отправка данных по частям и так далее.

Можно реализовать и возобновляемую отправку и многое другое.

Длинные опросы

Длинные опросы – это самый простой способ поддерживать постоянное соединение с сервером, не используя при этом никаких специфических протоколов (типа WebSocket или Server Side Events).

Его очень легко реализовать, и он хорошо подходит для многих задач.

Частые опросы

Самый простой способ получать новую информацию от сервера – периодический опрос. То есть, регулярные запросы на сервер вида: «Привет, я здесь, у вас есть какая-нибудь информация для меня?». Например, раз в 10 секунд.

В ответ сервер, во-первых, помечает у себя, что клиент онлайн, а во-вторых посылает весь пакет сообщений, накопившихся к данному моменту.

Это работает, но есть и недостатки:

1. Сообщения передаются с задержкой до 10 секунд (между запросами).
2. Даже если сообщений нет, сервер «атакуется» запросами каждые 10 секунд, даже если пользователь переключился куда-нибудь или спит. С точки зрения производительности, это довольно большая нагрузка.

Так что, если речь идёт об очень маленьком сервисе, подход может оказаться жизнеспособным, но в целом он нуждается в улучшении.

Длинные опросы

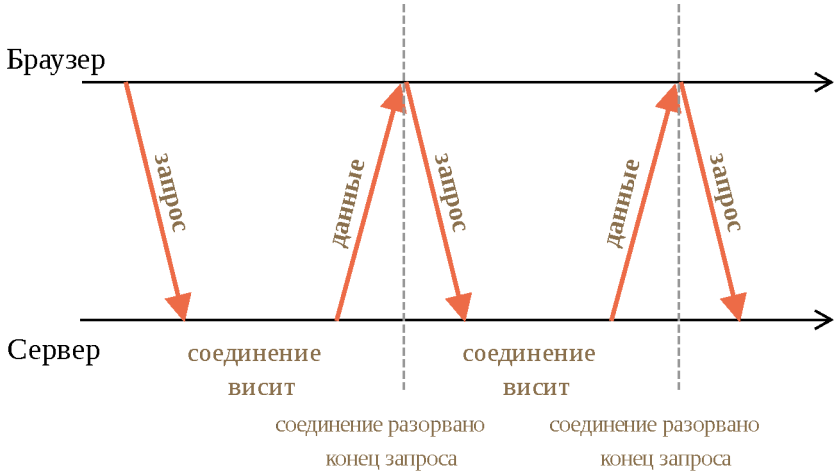
«Длинные опросы» – гораздо лучший способ взаимодействия с сервером.

Они также очень просты в реализации, и сообщения доставляются без задержек.

Как это происходит:

1. Запрос отправляется на сервер.
2. Сервер не закрывает соединение, пока у него не возникнет сообщение для отсылки.
3. Когда появляется сообщение – сервер отвечает на запрос, посылая его.
4. Браузер немедленно делает новый запрос.

Для данного метода ситуация, когда браузер отправил запрос и удерживает соединение с сервером, ожидании ответа, является стандартной. Соединение прерывается только доставкой сообщений.



Если соединение будет потеряно, скажем, из-за сетевой ошибки, браузер немедленно посылает новый запрос.

Примерный код клиентской функции `subscribe`, которая реализует длинные опросы:

```
async function subscribe() {
  let response = await fetch("/subscribe");

  if (response.status == 502) {
    // Статус 502 - это таймаут соединения;
    // возможен, когда соединение ожидало слишком долго
    // и сервер (или промежуточный прокси) закрыл его
    // давайте восстановим связь
    await subscribe();
  } else if (response.status != 200) {
    // Какая-то ошибка, покажем её
    showMessage(response.statusText);
    // Подключимся снова через секунду.
    await new Promise(resolve => setTimeout(resolve, 1000));
    await subscribe();
  } else {
    // Получим и покажем сообщение
    let message = await response.text();
    showMessage(message);
    // И снова вызовем subscribe() для получения следующего сообщения
    await subscribe();
  }
}

subscribe();
```

Функция `subscribe()` делает запрос, затем ожидает ответ, обрабатывает его и снова вызывает сама себя.

> ⚠ **Сервер должен поддерживать много ожидающих соединений.**
>
> Архитектура сервера должна быть способна работать со многими ожидающими подключениями.
>
> Некоторые серверные архитектуры запускают отдельный процесс для каждого соединения. Для большого количества соединений будет столько же процессов, и каждый процесс занимает значительный обьём памяти. Так много соединений просто поглотят всю память.
>
> Часто такая проблема возникает с бэкендом, написанными на PHP или Ruby, но технически дело не в языке, а в реализации. На большинстве современных языков можно написать подходящий сервер, но на некоторых это проще сделать.
>
> Бэкенды, написанные с помощью Node.js, обычно не имеют таких проблем.

Демо: чат

Вот демо-чат, вы также можете скачать его и запустить локально (если вам знаком Node.js и можете поставить модули):

https://plnkr.co/edit/MSr4xrLK6XaGPokBcNbf?p=preview

Браузерный код находится в `browser.js`.

Область применения

Длинные опросы прекрасно работают, когда сообщения приходят редко.

Если сообщения приходят очень часто, то схема приёма-отправки сообщений, приведённая выше, становится похожей на «пилу».

Каждое сообщение – это отдельный запрос, с заголовками, авторизацией и так далее.

Поэтому в этом случае предпочтительнее использовать другой метод, такой как Websocket или Server Sent Events.

WebSocket

Протокол `WebSocket` («вебсокет»), описанный в спецификации RFC 6455 , обеспечивает возможность обмена данными между браузером и сервером через постоянное соединение. Данные передаются по нему в обоих

направлениях в виде «пакетов», без разрыва соединения и дополнительных HTTP-запросов.

WebSocket особенно хорош для сервисов, которые нуждаются в постоянном обмене данными, например онлайн игры, торговые площадки, работающие в реальном времени, и т.д.

Простой пример

Чтобы открыть вебсокет-соединение, нам нужно создать объект `new WebSocket`, указав в url-адресе специальный протокол `ws`:

```
let socket = new WebSocket("ws://javascript.info");
```

Также существует протокол `wss://`, использующий шифрование. Это как HTTPS для вебсокетов.

> **Всегда предпочитайте `wss://`**
>
> Протокол `wss://` не только использует шифрование, но и обладает повышенной надёжностью.
>
> Это потому, что данные `ws://` не зашифрованы, видны для любого посредника. Старые прокси-серверы не знают о WebSocket, они могут увидеть «странные» заголовки и закрыть соединение.
>
> С другой стороны, `wss://` – это WebSocket поверх TLS (так же, как HTTPS – это HTTP поверх TLS), безопасный транспортный уровень шифрует данные от отправителя и расшифровывает на стороне получателя. Пакеты данных передаются в зашифрованном виде через прокси, которые не могут видеть, что внутри, и всегда пропускают их.

Как только объект `WebSocket` создан, мы должны слушать его события. Их всего 4:

- **`open`** – соединение установлено,
- **`message`** – получены данные,
- **`error`** – ошибка,
- **`close`** – соединение закрыто.

…А если мы хотим отправить что-нибудь, то вызов `socket.send(data)` сделает это.

Вот пример:

```
let socket = new WebSocket("wss://javascript.info/article/websocket/demo/hello");

socket.onopen = function(e) {
  alert("[open] Соединение установлено");
  alert("Отправляем данные на сервер");
  socket.send("Меня зовут Джон");
};

socket.onmessage = function(event) {
  alert(`[message] Данные получены с сервера: ${event.data}`);
};

socket.onclose = function(event) {
  if (event.wasClean) {
    alert(`[close] Соединение закрыто чисто, код=${event.code} причина=${event.reas
  } else {
    // например, сервер убил процесс или сеть недоступна
    // обычно в этом случае event.code 1006
    alert('[close] Соединение прервано');
  }
};

socket.onerror = function(error) {
  alert(`[error] ${error.message}`);
};
```

Для демонстрации есть небольшой пример сервера server.js, написанного на Node.js, для запуска примера выше. Он отвечает «Привет с сервера, Джон», после ожидает 5 секунд и закрывает соединение.

Так вы увидите события `open` → `message` → `close`.

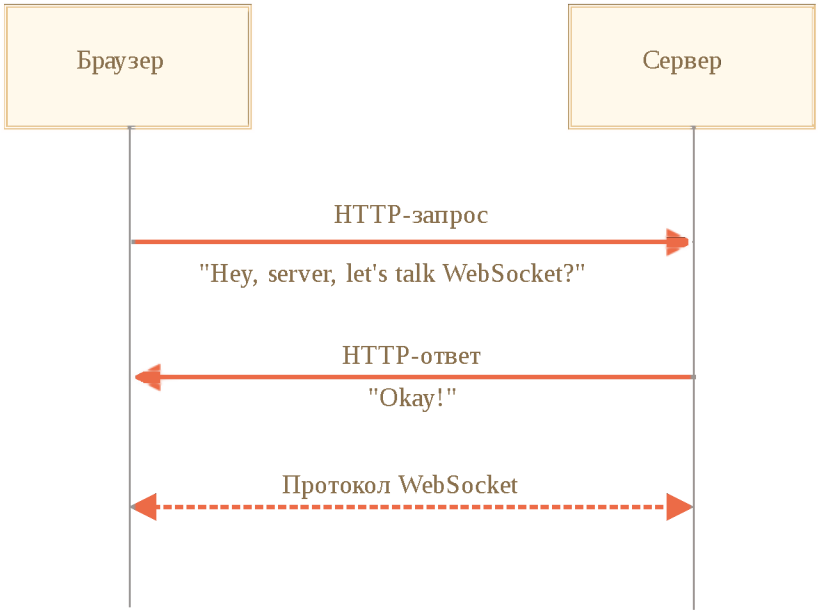
В общем-то, всё, мы уже можем общаться по протоколу WebSocket. Просто, не так ли?

Теперь давайте поговорим более подробно.

Открытие вебсокета

Когда `new WebSocket(url)` создан, он тут же сам начинает устанавливать соединение.

Браузер, при помощи специальных заголовков, спрашивает сервер: «Ты поддерживаешь Websocket?» и если сервер отвечает «да», они начинают работать по протоколу WebSocket, который уже не является HTTP.



Вот пример заголовков для запроса, который делает `new WebSocket("wss://javascript.info/chat")`.

```
GET /chat
Host: javascript.info
Origin: https://javascript.info
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
Sec-WebSocket-Version: 13
```

- `Origin` – источник текущей страницы (например `https://javascript.info`). Объект WebSocket по своей природе не завязан на текущий источник. Нет никаких специальных заголовков или других ограничений. Старые сервера все равно не могут работать с WebSocket, поэтому проблем с совместимостью нет. Но заголовок `Origin` важен, так как он позволяет серверу решать, использовать ли WebSocket с этим сайтом.
- `Connection: Upgrade` – сигнализирует, что клиент хотел бы изменить протокол.
- `Upgrade: websocket` – запрошен протокол «websocket».
- `Sec-WebSocket-Key` – случайный ключ, созданный браузером для обеспечения безопасности.
- `Sec-WebSocket-Version` – версия протокола WebSocket, текущая версия 13.

Запрос WebSocket нельзя эмулировать

Мы не можем использовать `XMLHttpRequest` или `fetch` для создания такого HTTP-запроса, потому что JavaScript не позволяет устанавливать такие заголовки.

Если сервер согласен переключиться на WebSocket, то он должен отправить в ответ код 101:

```
101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: hsBlbuDTkk24srzEOTBUlZAlC2g=
```

Здесь `Sec-WebSocket-Accept` – это `Sec-WebSocket-Key`, перекодированный с помощью специального алгоритма. Браузер использует его, чтобы убедиться, что ответ соответствует запросу.

После этого данные передаются по протоколу WebSocket, и вскоре мы увидим его структуру («фреймы»). И это вовсе не HTTP.

Расширения и подпротоколы

Могут быть дополнительные заголовки `Sec-WebSocket-Extensions` и `Sec-WebSocket-Protocol`, описывающие расширения и подпротоколы.

Например:

- `Sec-WebSocket-Extensions: deflate-frame` означает, что браузер поддерживает сжатие данных. Расширение – это что-то, связанное с передачей данных, расширяющее сам протокол WebSocket. Заголовок `Sec-WebSocket-Extensions` отправляется браузером автоматически со списком всевозможных расширений, которые он поддерживает.
- `Sec-WebSocket-Protocol: soap, wamp` означает, что мы будем передавать не только произвольные данные, но и данные в протоколах SOAP или WAMP (The WebSocket Application Messaging Protocol" – «протокол обмена сообщениями WebSocket приложений»). Подпротоколы WebSocket регистрируются в каталоге IANA.

  Этот необязательный заголовок ставим мы сами, чтобы сказать серверу, какие подпротоколы поддерживает наш код, при помощи второго (необязательного) параметра `new WebSocket`. Он содержит массив подпротоколов, например если мы хотим использовать SOAP или WAMP:

  ```
  let socket = new WebSocket("wss://javascript.info/chat", ["soap", "wamp"]);
  ```

Этот заголовок ставим мы сами,

А заголовок `Sec-WebSocket-Protocol` уже описывает, какого вида данные мы хотим отправлять и посылать, он зависит от нас. Второй необязательный параметр `new WebSocket` как раз для этого и предназначен – это массив подпротоколов, например для данных в форматах SOAP и WAMP:

```
let socket = new WebSocket("wss://javascript.info/chat", ["soap", "wamp"]);
```

Сервер должен ответить перечнем протоколов и расширений, которые он может использовать.

Например, запрос:

```
GET /chat
Host: javascript.info
Upgrade: websocket
Connection: Upgrade
Origin: https://javascript.info
Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
Sec-WebSocket-Version: 13
Sec-WebSocket-Extensions: deflate-frame
Sec-WebSocket-Protocol: soap, wamp
```

Ответ:

```
101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: hsBlbuDTkk24srzEOTBUlZAlC2g=
Sec-WebSocket-Extensions: deflate-frame
Sec-WebSocket-Protocol: soap
```

Здесь сервер отвечает, что поддерживает расширение – deflate-frame и может использовать только протокол SOAP из всего списка запрошенных подпротоколов.

Передача данных

Поток данных в WebSocket состоит из «фреймов», фрагментов данных, которые могут быть отправлены любой стороной, и которые могут быть следующих видов:

- «текстовые фреймы» – содержат текстовые данные, которые стороны отправляют друг другу.
- «бинарные фреймы» – содержат бинарные данные, которые стороны отправляют друг другу.
- «пинг-понг фреймы» используется для проверки соединения; отправляется с сервера, браузер реагирует на них автоматически.
- также есть «фрейм закрытия соединения» и некоторые другие служебные фреймы.

В браузере мы напрямую работаем только с текстовыми и бинарными фреймами.

Метод WebSocket `.send()` может отправлять и текстовые и бинарные данные.

Вызов `socket.send(body)` принимает `body` в виде строки или любом бинарном формате включая `Blob`, `ArrayBuffer` и другие. Дополнительных настроек не требуется, просто отправляем в любом формате.

При получении данных, текст всегда поступает в виде строки. А для бинарных данных мы можем выбрать один из двух форматов: `Blob` или `ArrayBuffer`.

Это задаётся свойством `socket.bufferType`, по умолчанию оно равно `"blob"`, так что бинарные данные поступают в виде `Blob`-объектов.

Blob – это высокоуровневый бинарный объект, он напрямую интегрируется с `<a>`, `<img>` и другими тегами, так что это вполне удобное значение по умолчанию. Но для обработки данных, если требуется доступ к отдельным байтам, мы можем изменить его на `"arraybuffer"`:

```
socket.bufferType = "arraybuffer";
socket.onmessage = (event) => {
  // event.data является строкой (если текст) или arraybuffer (если двоичные данные
};
```

Ограничение скорости

Представим, что наше приложение генерирует много данных для отправки. Но у пользователя медленное соединение, возможно, он в интернете с мобильного телефона и не из города.

Мы можем вызывать `socket.send(data)` снова и снова. Но данные будут буферизованы (сохранены) в памяти и отправлены лишь с той скоростью, которую позволяет сеть.

Свойство `socket.bufferedAmount` хранит количество байт буферизованных данных на текущий момент, ожидающих отправки по сети.

Мы можем изучить его, чтобы увидеть, действительно ли сокет доступен для передачи.

```
// каждые 100мс проверить сокет и отправить больше данных,
// только если все текущие отосланы
setInterval(() => {
  if (socket.bufferedAmount == 0) {
    socket.send(moreData());
  }
}, 100);
```

Закрытие подключения

Обычно, когда сторона хочет закрыть соединение (браузер и сервер имеют равные права), они отправляют «фрейм закрытия соединения» с кодом закрытия и указывают причину в виде текста.

Метод для этого:

```
socket.close([code], [reason]);
```

- `code` – специальный WebSocket-код закрытия (не обязателен).
- `reason` – строка с описанием причины закрытия (не обязательна).

Затем противоположная сторона в обработчике события `close` получит и код `code` и причину `reason`, например:

```
// закрывающая сторона:
socket.close(1000, "работа закончена");

// другая сторона:
socket.onclose = event => {
  // event.code === 1000
  // event.reason === "работа закончена"
  // event.wasClean === true (закрыто чисто)
};
```

`code` – это не любое число, а специальный код закрытия WebSocket.

Наиболее распространённые значения:

- `1000` – по умолчанию, нормальное закрытие,

- `1006` – невозможно установить такой код вручную, указывает, что соединение было потеряно (нет фрейма закрытия).

Есть и другие коды:

- `1001` – сторона отключилась, например сервер выключен или пользователь покинул страницу,
- `1009` – сообщение слишком большое для обработки,
- `1011` – непредвиденная ошибка на сервере,
- …и так далее.

Полный список находится в RFC6455, §7.4.1 .

Коды WebSocket чем-то похожи на коды HTTP, но они разные. В частности, любые коды меньше `1000` зарезервированы. Если мы попытаемся установить такой код, то получим ошибку.

```
// в случае, если соединение сброшено
socket.onclose = event => {
  // event.code === 1006
  // event.reason === ""
  // event.wasClean === false (нет закрывающего кадра)
};
```

Состояние соединения

Чтобы получить состояние соединения, существует дополнительное свойство `socket.readyState` со значениями:

- **`0`** – «CONNECTING»: соединение ещё не установлено,
- **`1`** – «OPEN»: обмен данными,
- **`2`** – «CLOSING»: соединение закрывается,
- **`3`** – «CLOSED»: соединение закрыто.

Пример чата

Давайте рассмотрим пример чата с использованием WebSocket API и модуля WebSocket сервера Node.js https://github.com/websockets/ws . Основное внимание мы, конечно, уделим клиентской части, но и серверная весьма проста.

HTML: нам нужна форма `<form>` для отправки данных и `<div>` для отображения сообщений:

```
<!-- форма сообщений -->
<form name="publish">
  <input type="text" name="message">
  <input type="submit" value="Отправить">
</form>

<!-- div с сообщениями -->
<div id="messages"></div>
```

От JavaScript мы хотим 3 вещи:

1. Открыть соединение.
2. При отправке формы пользователем – вызвать `socket.send(message)` для сообщения.
3. При получении входящего сообщения – добавить его в `div#messages`.

Вот код:

```
let socket = new WebSocket("wss://javascript.info/article/websocket/chat/ws");

// отправка сообщения из формы
document.forms.publish.onsubmit = function() {
  let outgoingMessage = this.message.value;

  socket.send(outgoingMessage);
  return false;
};

// получение сообщения - отобразить данные в div#messages
socket.onmessage = function(event) {
  let message = event.data;

  let messageElem = document.createElement('div');
  messageElem.textContent = message;
  document.getElementById('messages').prepend(messageElem);
}
```

Серверный код выходит за рамки этой главы. Здесь мы будем использовать Node.js, но вы не обязаны это делать. Другие платформы также поддерживают средства для работы с WebSocket.

Серверный алгоритм действий будет таким:

1. Создать `clients = new Set()` – набор сокетов.
2. Для каждого принятого вебсокета – добавить его в набор `clients.add(socket)` и поставить ему обработчик события `message` для приёма сообщений.

3. Когда сообщение получено: перебрать клиентов `clients` и отправить его всем.
4. Когда подключение закрыто: `clients.delete(socket)`.

```
const ws = new require('ws');
const wss = new ws.Server({noServer: true});

const clients = new Set();

http.createServer((req, res) => {
  // в реальном проекте здесь может также быть код для обработки отличных от websoc
  // здесь мы работаем с каждым запросом как с вебсокетом
  wss.handleUpgrade(req, req.socket, Buffer.alloc(0), onSocketConnect);
});

function onSocketConnect(ws) {
  clients.add(ws);

  ws.on('message', function(message) {
    message = message.slice(0, 50); // максимальный размер сообщения 50

    for(let client of clients) {
      client.send(message);
    }
  });

  ws.on('close', function() {
    clients.delete(ws);
  });
}
```

Вот рабочий пример:

Вы также можете скачать его (верхняя правая кнопка в ифрейме) и запустить локально. Только не забудьте установить Node.js и выполнить команду `npm install ws` до запуска.

Итого

WebSocket – это современный способ иметь постоянное соединение между браузером и сервером.

- Нет ограничений, связанных с кросс-доменными запросами.
- Имеют хорошую поддержку браузерами.
- Могут отправлять/получать как строки, так и бинарные данные.

API прост.

Методы:

- `socket.send(data)`,
- `socket.close([code], [reason])`.

События:

- `open`,
- `message`,
- `error`,
- `close`.

WebSocket сам по себе не содержит такие функции, как переподключение при обрыве соединения, аутентификацию пользователей и другие механизмы высокого уровня. Для этого есть клиентские и серверные библиотеки, а также можно реализовать это вручную.

Иногда, чтобы добавить WebSocket к уже существующему проекту, WebSocket-сервер запускают параллельно с основным сервером. Они совместно использует одну базу данных. Запросы к WebSocket отправляются на `wss://ws.site.com` – поддомен, который ведёт к WebSocket-серверу, в то время как `https://site.com` ведёт на основной HTTP-сервер.

Конечно, возможны и другие пути интеграции.

Server Sent Events

Спецификация Server-Sent Events описывает встроенный класс `EventSource`, который позволяет поддерживать соединение с сервером и получать от него события.

Как и в случае с `WebSocket`, соединение постоянно.

Но есть несколько важных различий:

`WebSocket`	`EventSource`
Двунаправленность: и сервер, и клиент могут обмениваться сообщениями	Однонаправленность: данные посылает только сервер
Бинарные и текстовые данные	Только текст

`WebSocket`	`EventSource`
Протокол WebSocket	Обычный HTTP

`EventSource` не настолько мощный способ коммуникации с сервером, как `WebSocket`.

Зачем нам его использовать?

Основная причина: он проще. Многим приложениям не требуется вся мощь `WebSocket`.

Если нам нужно получать поток данных с сервера: неважно, сообщения в чате или же цены для магазина – с этим легко справится `EventSource`. К тому же, он поддерживает автоматическое переподключение при потере соединения, которое, используя `WebSocket`, нам бы пришлось реализовывать самим. Кроме того, используется старый добрый HTTP, а не новый протокол.

Получение сообщений

Чтобы начать получать данные, нам нужно просто создать `new EventSource(url)`.

Браузер установит соединение с `url` и будет поддерживать его открытым, ожидая события.

Сервер должен ответить со статусом 200 и заголовком `Content-Type: text/event-stream`, затем он должен поддерживать соединение открытым и отправлять сообщения в особом формате:

```
data: Сообщение 1

data: Сообщение 2

data: Сообщение 3
data: в две строки
```

- Текст сообщения указывается после `data:`, пробел после двоеточия необязателен.
- Сообщения разделяются двойным переносом строки `\n\n`.
- Чтобы разделить сообщение на несколько строк, мы можем отправить несколько `data:` подряд (третье сообщение).

На практике сложные сообщения обычно отправляются в формате JSON, в котором перевод строки кодируется как `\n`, так что в разделении сообщения на несколько строк обычно нет нужды.

Например:

```
data: {"user":"Джон","message":"Первая строка\n Вторая строка"}
```

...Так что можно считать, что в каждом `data:` содержится ровно одно сообщение.

Для каждого сообщения генерируется событие `message`:

```
let eventSource = new EventSource("/events/subscribe");

eventSource.onmessage = function(event) {
  console.log("Новое сообщение", event.data);
  // этот код выведет в консоль 3 сообщения, из потока данных выше
};

// или eventSource.addEventListener('message', ...)
```

Кросс-доменные запросы

`EventSource`, как и `fetch`, поддерживает кросс-доменные запросы. Мы можем использовать любой URL:

```
let source = new EventSource("https://another-site.com/events");
```

Сервер получит заголовок `Origin` и должен будет ответить с заголовком `Access-Control-Allow-Origin`.

Чтобы послать авторизационные данные, следует установить дополнительную опцию `withCredentials`:

```
let source = new EventSource("https://another-site.com/events", {
  withCredentials: true
});
```

Более подробное описание кросс-доменных заголовков вы можете прочитать в главе Fetch: запросы на другие сайты.

Переподключение

После создания `new EventSource` подключается к серверу и, если соединение обрывается, – переподключается.

Это очень удобно, так как нам не приходится беспокоиться об этом.

По умолчанию между попытками возобновить соединение будет небольшая пауза в несколько секунд.

Сервер может выставить рекомендуемую задержку, указав в ответе `retry:` (в миллисекундах):

```
retry: 15000
data: Привет, я выставил задержку переподключения в 15 секунд
```

Поле `retry:` может посылаться как вместе с данными, так и отдельным сообщением.

Браузеру следует ждать именно столько миллисекунд перед новой попыткой подключения. Или дольше, например, если браузер знает (от операционной сисстемы) что соединения с сетью нет, то он может осуществить переподключение только когда оно появится.

- Если сервер хочет остановить попытки переподключения, он должен ответить со статусом 204.
- Если браузер хочет прекратить соединение, он может вызвать `eventSource.close()`:

```
let eventSource = new EventSource(...);

eventSource.close();
```

Также переподключение не произойдёт, если в ответе указан неверный `Content-Type` или его статус отличается от 301, 307, 200 и 204. Браузер создаст событие `"error"` и не будет восстанавливать соединение.

> **На заметку:**
>
> После того как соединение окончательно закрыто, «переоткрыть» его уже нельзя. Если необходимо снова подключиться, просто создайте новый `EventSource`.

Идентификатор сообщения

Когда соединение прерывается из-за проблем с сетью, ни сервер, ни клиент не могут быть уверены в том, какие сообщения были доставлены, а какие – нет.

Чтобы правильно возобновить подключение, каждое сообщение должно иметь поле `id`:

```
data: Сообщение 1
id: 1

data: Сообщение 2
id: 2

data: Сообщение 3
data: в две строки
id: 3
```

Получая сообщение с указанным `id:`, браузер:

- Установит его значение свойству `eventSource.lastEventId`.
- При переподключении отправит заголовок `Last-Event-ID` с этим `id`, чтобы сервер мог переслать последующие сообщения.

> **Указывайте `id:` после `data:`**
>
> Обратите внимание: `id` указывается сервером после данных `data` сообщения, чтобы обновление `lastEventId` произошло после того, как сообщение будет получено.

Статус подключения: readyState

У объекта `EventSource` есть свойство `readyState`, имеющее одно из трёх значений:

```
EventSource.CONNECTING = 0; // подключение или переподключение
EventSource.OPEN = 1;       // подключено
EventSource.CLOSED = 2;     // подключение закрыто
```

При создании объекта и разрыве соединения оно автоматически устанавливается в значение `EventSource.CONNECTING` (равно `0`).

Мы можем обратиться к этому свойству, чтобы узнать текущее состояние `EventSource`.

Типы событий

По умолчанию объект `EventSource` генерирует 3 события:

- `message` – получено сообщение, доступно как `event.data`.
- `open` – соединение открыто.
- `error` – не удалось установить соединение, например, сервер вернул статус 500.

Сервер может указать другой тип события с помощью `event: ...` в начале сообщения.

Например:

```
event: join
data: Боб

data: Привет

event: leave
data: Боб
```

Чтобы начать слушать пользовательские события, нужно использовать `addEventListener`, а не `onmessage`:

```
eventSource.addEventListener('join', event => {
  alert(`${event.data} зашёл`);
});

eventSource.addEventListener('message', event => {
  alert(`Сказал: ${event.data}`);
});

eventSource.addEventListener('leave', event => {
  alert(`${event.data} вышел`);
});
```

Полный пример

В этом примере сервер посылает сообщения `1`, `2`, `3`, затем `пока-пока` и разрывает соединение.

После этого браузер автоматически переподключается.

https://plnkr.co/edit/GFICkvPzjIne2LfCDfro?p=preview

Итого

Объект `EventSource` автоматически устанавливает постоянное соединение и позволяет серверу отправлять через него сообщения.

Он предоставляет:

- Автоматическое переподключение с настраиваемой `retry` задержкой.
- Идентификаторы сообщений для восстановления соединения. Последний полученный идентификатор посылается в заголовке `Last-Event-ID` при пересоединении.
- Текущее состояние, записанное в свойстве `readyState`.

Это делает `EventSource` достойной альтернативой протоколу `WebSocket`, который сравнительно низкоуровневый и не имеет таких встроенных возможностей (хотя их и можно реализовать).

Для многих приложений возможностей `EventSource` вполне достаточно.

Поддерживается во всех современных браузерах (кроме Internet Explorer).

Синтаксис:

```
let source = new EventSource(url, [credentials]);
```

Второй аргумент – необязательный объект с одним свойством: `{ withCredentials: true }`. Он позволяет отправлять авторизационные данные на другие домены.

В целом, кросс-доменная безопасность реализована так же как в `fetch` и других методов работы с сетью.

Свойства объекта `EventSource`

`readyState`

Текущее состояние подключения: `EventSource.CONNECTING (=0)`, `EventSource.OPEN (=1)` или `EventSource.CLOSED (=2)`.

`lastEventId`

`id` последнего полученного сообщения. При переподключении браузер посылает его в заголовке `Last-Event-ID`.

Методы

`close()`

Закрывает соединение.

События

`message`

Сообщение получено, переданные данные записаны в `event.data`.

`open`

Соединение установлено.

`error`

В случае ошибки, включая как потерю соединения так и другие ошибки в нём. Мы можем обратиться к свойству `readyState`, чтобы проверить, происходит ли переподключение.

Сервер может выставить собственное событие с помощью `event:`. Такие события должны быть обработаны с помощью `addEventListener`, а не `on<event>`.

Формат ответа сервера

Сервер посылает сообщения, разделённые двойным переносом строки `\n\n`.

Сообщение состоит из следующих полей:

- `data:` – тело сообщения, несколько `data` подряд интерпретируются как одно сообщение, разделённое переносами строк `\n`.
- `id:` – обновляет свойство `lastEventId`, отправляемое в `Last-Event-ID` при переподключении.
- `retry:` – рекомендованная задержка перед переподключением в миллисекундах. Не может быть установлена с помощью JavaScript.
- `event:` – имя пользовательского события, должно быть указано перед `data:`.

Сообщение может включать одно или несколько этих полей в любом порядке, но id обычно ставят в конце.

Хранение данных в браузере

Куки, document.cookie

Куки – это небольшие строки данных, которые хранятся непосредственно в браузере. Они являются частью HTTP-протокола, определённого в спецификации RFC 6265.

Куки обычно устанавливаются веб-сервером при помощи заголовка `Set-Cookie`. Затем браузер будет автоматически добавлять их в (почти) каждый запрос на тот же домен при помощи заголовка `Cookie`.

Один из наиболее частых случаев использования куки – это аутентификация:

1. При входе на сайт сервер отсылает в ответ HTTP-заголовок `Set-Cookie` для того, чтобы установить куки со специальным уникальным идентификатором сессии («session identifier»).
2. Во время следующего запроса к этому же домену браузер посылает на сервер HTTP-заголовок `Cookie`.
3. Таким образом, сервер понимает, кто сделал запрос.

Мы также можем получить доступ к куки непосредственно из браузера, используя свойство `document.cookie`.

Куки имеют множество особенностей и тонкостей в использовании, и в этой главе мы подробно с ними разберёмся.

Чтение из document.cookie

Если предположить, что вы зашли на сайт, то куки можно посмотреть вот так:

```
// На javascript.info мы используем сервис Google Analytics для сбора статистики,
// поэтому какие-то куки должны быть
alert( document.cookie ); // cookie1=value1; cookie2=value2;...
```

Значение `document.cookie` состоит из пар `ключ=значение`, разделённых `;`. Каждая пара представляет собой отдельное куки.

Чтобы найти определённое куки, достаточно разбить строку из `document.cookie` по `;`, и затем найти нужный ключ. Для этого мы можем использовать как регулярные выражения, так и функции для обработки массивов.

Оставим эту задачу читателю для самостоятельного выполнения. Кроме того, в конце этой главы вы найдёте полезные функции для работы с куки.

Запись в document.cookie

Мы можем писать в `document.cookie`. Но это не просто данные, а аксессор (геттер/сеттер). Присваивание обрабатывается особым образом.

Запись в `document.cookie` обновит только упомянутые в ней куки, но при этом не затронет все остальные.

Например, этот вызов установит куки с именем `user` и значением `John`:

```
document.cookie = "user=John"; // обновляем только куки с именем 'user'
alert(document.cookie); // показываем все куки
```

Если вы запустите этот код, то, скорее всего, увидите множество куки. Это происходит, потому что операция `document.cookie=` перезапишет не все куки, а лишь куки с вышеупомянутым именем `user`.

Технически, и имя и значение куки могут состоять из любых символов, для правильного форматирования следует использовать встроенную функцию `encodeURIComponent`:

```
// специальные символы (пробелы), требуется кодирование
let name = "my name";
let value = "John Smith"

// кодирует в my%20name=John%20Smith
document.cookie = encodeURIComponent(name) + '=' + encodeURIComponent(value);

alert(document.cookie); // ...; my%20name=John%20Smith
```

⚠ **Ограничения**

Существует несколько ограничений:

- После `encodeURIComponent` пара `name=value` не должна занимать более 4Кб. Таким образом, мы не можем хранить в куки большие данные.
- Общее количество куки на один домен ограничивается примерно 20+. Точное ограничение зависит от конкретного браузера.

У куки есть ряд настроек, многие из которых важны и должны быть установлены.

Эти настройки указываются после пары `ключ=значение` и отделены друг от друга разделителем `;`, вот так:

```
document.cookie = "user=John; path=/; expires=Tue, 19 Jan 2038 03:14:07 GMT"
```

path

- **`path=/mypath`**

URL-префикс пути, куки будут доступны для страниц под этим путём. Должен быть абсолютным. По умолчанию используется текущий путь.

Если куки установлено с `path=/admin`, то оно будет доступно на страницах `/admin` и `/admin/something`, но не на страницах `/home` или `/adminpage`.

Как правило, указывают в качестве пути корень `path=/`, чтобы наше куки было доступно на всех страницах сайта.

domain

- **`domain=site.com`**

Домен, на котором доступны наши куки. На практике, однако, есть ограничения – мы не можем указать здесь какой угодно домен.

По умолчанию куки доступно лишь тому домену, который его установил. Так что куки, которые были установлены сайтом `site.com`, не будут доступны на сайте `other.com`.

…Но что более интересно, мы не сможем получить эти куки на поддомене `forum.site.com`!

```
// на site.com
document.cookie = "user=John"

// на forum.site.com
alert(document.cookie); // нет user
```

Нет способа сделать куки доступным на другом домене 2-го уровня, так что `other.com` никогда не получит куки, установленное сайтом `site.com`.

Это ограничение безопасности, чтобы мы могли хранить в куки конфиденциальные данные, предназначенные только для одного сайта.

…Однако, если мы всё же хотим дать поддоменам типа `forum.site.com` доступ к куки, это можно сделать. Достаточно при установке куки на сайте `site.com` в качестве значения опции `domain` указать корневой домен: `domain=site.com`:

```
// находясь на странице site.com
// сделаем куки доступным для всех поддоменов *.site.com:
document.cookie = "user=John; domain=site.com"

// позже
```

```
// на forum.site.com
alert(document.cookie); // есть куки user=John
```

По историческим причинам установка `domain=.site.com` (с точкой перед `site.com`) также работает и разрешает доступ к куки для поддоменов. Это старая запись, но можно использовать и её, если нужно, чтобы поддерживались очень старые браузеры.

Таким образом, опция `domain` позволяет нам разрешить доступ к куки для поддоменов.

expires, max-age

По умолчанию, если куки не имеют ни одного из этих параметров, то они удалятся при закрытии браузера. Такие куки называются сессионными («session cookies»).

Чтобы помочь куки «пережить» закрытие браузера, мы можем установить значение опций `expires` или `max-age`.

- **`expires=Tue, 19 Jan 2038 03:14:07 GMT`**

Дата истечения срока действия куки, когда браузер удалит его автоматически.

Дата должна быть точно в этом формате, во временной зоне GMT. Мы можем использовать `date.toUTCString`, чтобы получить правильную дату. Например, мы можем установить срок действия куки на 1 день.

```
// +1 день от текущей даты
let date = new Date(Date.now() + 86400e3);
date = date.toUTCString();
document.cookie = "user=John; expires=" + date;
```

Если мы установим в `expires` прошедшую дату, то куки будет удалено.

- **`max-age=3600`**

Альтернатива `expires`, определяет срок действия куки в секундах с текущего момента.

Если задан ноль или отрицательное значение, то куки будет удалено:

```
// куки будет удалено через 1 час
document.cookie = "user=John; max-age=3600";
```

```
// удалим куки (срок действия истекает прямо сейчас)
document.cookie = "user=John; max-age=0";
```

secure

- `secure`

Куки следует передавать только по HTTPS-протоколу.

По умолчанию куки, установленные сайтом `http://site.com`, также будут доступны на сайте `https://site.com` и наоборот.

То есть, куки, по умолчанию, опираются на доменное имя, они не обращают внимания на протоколы.

С этой настройкой, если куки будет установлено на сайте `https://site.com`, то оно не будет доступно на том же сайте с протоколом HTTP, как `http://site.com`. Таким образом, если в куки хранится конфиденциальная информация, которую не следует передавать по незашифрованному протоколу HTTP, то нужно установить этот флаг.

```
// предполагается, что сейчас мы на https://
// установим опцию secure для куки (куки доступно только через HTTPS)
document.cookie = "user=John; secure";
```

samesite

Это ещё одна настройка безопасности, применяется для защиты от так называемой XSRF-атаки (межсайтовая подделка запроса).

Чтобы понять, как настройка работает и где может быть полезной, посмотрим на XSRF-атаки.

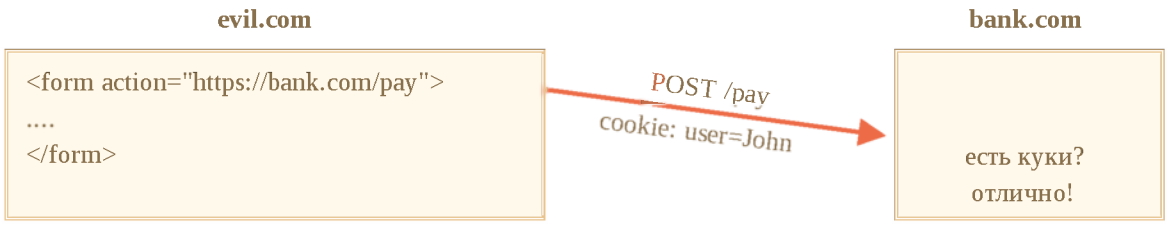
Атака XSRF

Представьте, вы авторизовались на сайте `bank.com`. То есть: у вас есть куки для аутентификации с этого сайта. Ваш браузер отправляет его на сайт `bank.com` с каждым запросом, чтобы сервер этого сайта узнавал вас и выполнял все конфиденциальные финансовые операции.

Теперь, просматривая веб-страницу в другом окне, вы случайно переходите на сайт `evil.com`, который автоматически отправляет форму `<form action="https://bank.com/pay">` на сайт `bank.com` с заполненными полями, которые инициируют транзакцию на счёт хакера.

Браузер посылает куки при каждом посещении `bank.com`, даже если форма была отправлена с `evil.com`. Таким образом, банк узнает вас и выполнит

платёж.



Такая атака называется межсайтовая подделка запроса (или Cross-Site Request Forgery, XSRF).

Конечно же, в реальной жизни банки защищены от такой атаки. Во всех сгенерированных сайтом `bank.com` формах есть специальное поле, так называемый «токен защиты от xsrf», который вредоносная страница не может ни сгенерировать, ни каким-либо образом извлечь из удалённой страницы (она может отправить форму туда, но не может получить данные обратно). И сайт `bank.com` при получении формы проверяет его наличие.

Но такая защита требует усилий на её реализацию: нам нужно убедиться, что в каждой форме есть поле с токеном, также мы должны проверить все запросы.

Настройка samesite

Параметр куки `samesite` предоставляет ещё один способ защиты от таких атак, который (теоретически) не должен требовать «токенов защиты xsrf».

У него есть два возможных значения:

- **`samesite=strict` (или, что то же самое, `samesite` без значения)**

Куки с `samesite=strict` никогда не отправятся, если пользователь пришёл не с этого же сайта.

Другими словами, если пользователь переходит по ссылке из почты, отправляет форму с `evil.com` или выполняет любую другую операцию, происходящую с другого домена, то куки не отправляется.

Если куки имеют настройку `samesite`, то атака XSRF не имеет шансов на успех, потому что отправка с сайта `evil.com` происходит без куки. Таким образом, сайт `bank.com` не распознает пользователя и не произведёт платёж.

Защита довольно надёжная. Куки с настройкой `samesite` будет отправлено только в том случае, если операции происходят с сайта `bank.com`, например отправка формы сделана со страницы на `bank.com`.

Хотя есть небольшие неудобства.

Когда пользователь перейдёт по ссылке на `bank.com`, например из своих заметок, он будет удивлён, что сайт `bank.com` не узнал его. Действительно, куки с `samesite=strict` в этом случае не отправляется.

Мы могли бы обойти это ограничение, используя два куки: одно куки для «общего узнавания», только для того, чтобы поздороваться: «Привет, Джон», и другое куки для операций изменения данных с `samesite=strict`. Тогда пользователь, пришедший на сайт, увидит приветствие, но платежи нужно инициировать с сайта банка, чтобы отправилось второе куки.

- **`samesite=lax`**

Это более мягкий вариант, который также защищает от XSRF и при этом не портит впечатление от использования сайта.

Режим Lax так же, как и `strict`, запрещает браузеру отправлять куки, когда запрос происходит не с сайта, но добавляет одно исключение.

Куки с `samesite=lax` отправляется, если два этих условия верны:

1. Используются безопасные HTTP-методы (например, GET, но не POST).

   Полный список безопасных HTTP-методов можно посмотреть в спецификации RFC7231. По сути, безопасными считаются методы, которые обычно используются для чтения, но не для записи данных. Они не должны выполнять никаких операций на изменение данных. Переход по ссылке является всегда GET-методом, то есть безопасным.

2. Операция осуществляет навигацию верхнего уровня (изменяет URL в адресной строке браузера).

   Обычно это так, но если навигация выполняется в `<iframe>`, то это не верхний уровень. Кроме того, JavaScript-методы для сетевых запросов не выполняют никакой навигации, поэтому они не подходят.

Таким образом, режим `samesite=lax`, позволяет самой распространённой операции «переход по ссылке» передавать куки. Например, открытие сайта из заметок удовлетворяет этим условиям.

Но что-то более сложное, например, сетевой запрос с другого сайта или отправка формы, теряет куки.

Если это вам походит, то добавление `samesite=lax`, скорее всего, не испортит впечатление пользователей от работы с сайтом и добавит защиту.

В целом, `samesite` отличная настройка, но у неё есть важный недостаток:

- `samesite` игнорируется (не поддерживается) старыми браузерами, выпущенными до 2017 года и ранее.

Так что, если мы будем полагаться исключительно на `samesite`, то старые браузеры будут уязвимы.

Но мы, безусловно, можем использовать `samesite` вместе с другими методами защиты, такими как XSRF-токены, чтобы добавить дополнительный слой защиты, а затем, в будущем, когда старые браузеры полностью исчезнут, мы, вероятно, сможем полностью удалить XSRF-токены.

httpOnly

Эта настройка не имеет ничего общего с JavaScript, но мы должны упомянуть её для полноты изложения.

Веб-сервер использует заголовок `Set-Cookie` для установки куки. И он может установить настройку `httpOnly`.

Эта настройка запрещает любой доступ к куки из JavaScript. Мы не можем видеть такое куки или манипулировать им с помощью `document.cookie`.

Эта настройка используется в качестве меры предосторожности от определённых атак, когда хакер внедряет свой собственный JavaScript-код в страницу и ждёт, когда пользователь посетит её. Это вообще не должно быть возможным, хакер не должен быть в состоянии внедрить свой код на ваш сайт, но могут быть ошибки, которые позволят хакеру сделать это.

Обычно, если такое происходит, и пользователь заходит на страницу с JavaScript-кодом хакера, то этот код выполняется и получает доступ к `document.cookie`, и тем самым к куки пользователя, которые содержат аутентификационную информацию. Это плохо.

Но если куки имеет настройку `httpOnly`, то `document.cookie` не видит его, поэтому такое куки защищено.

Приложение: Функции для работы с куки

Вот небольшой набор функций для работы с куки, более удобных, чем ручная модификация `document.cookie`.

Для этого существует множество библиотек, так что они, скорее, в демонстрационных целях. Но при этом полностью рабочие.

getCookie(name)

Самый короткий способ получить доступ к куки – это использовать регулярные выражения.

Функция `getCookie(name)` возвращает куки с указанным `name`:

```
// возвращает куки с указанным name,
// или undefined, если ничего не найдено
function getCookie(name) {
  let matches = document.cookie.match(new RegExp(
    "(?:^|; )" + name.replace(/([\.$?*|{}\(\)\[\]\\\/\+^])/g, '\\$1') + "=([^;]*)"
  ));
  return matches ? decodeURIComponent(matches[1]) : undefined;
}
```

Здесь `new RegExp` генерируется динамически, чтобы находить `; name=<value>`.

Обратите внимание, значение куки кодируется, поэтому `getCookie` использует встроенную функцию `decodeURIComponent` для декодирования.

setCookie(name, value, options)

Устанавливает куки с именем `name` и значением `value`, с настройкой `path=/` по умолчанию (можно изменить, чтобы добавить другие значения по умолчанию):

```
function setCookie(name, value, options = {}) {

  options = {
    path: '/',
    // при необходимости добавьте другие значения по умолчанию
    ...options
  };

  if (options.expires.toUTCString) {
    options.expires = options.expires.toUTCString();
  }

  let updatedCookie = encodeURIComponent(name) + "=" + encodeURIComponent(value);

  for (let optionKey in options) {
    updatedCookie += "; " + optionKey;
    let optionValue = options[optionKey];
    if (optionValue !== true) {
      updatedCookie += "=" + optionValue;
    }
  }

  document.cookie = updatedCookie;
}

// Пример использования:
setCookie('user', 'John', {secure: true, 'max-age': 3600});
```

deleteCookie(name)

Чтобы удалить куки, мы можем установить отрицательную дату истечения срока действия:

```
function deleteCookie(name) {
  setCookie(name, "", {
    'max-age': -1
  })
}
```

⚠ **Операции обновления или удаления куки должны использовать те же путь и домен**

Обратите внимание: когда мы обновляем или удаляем куки, нам следует использовать только такие же настройки пути и домена, как при установки куки.

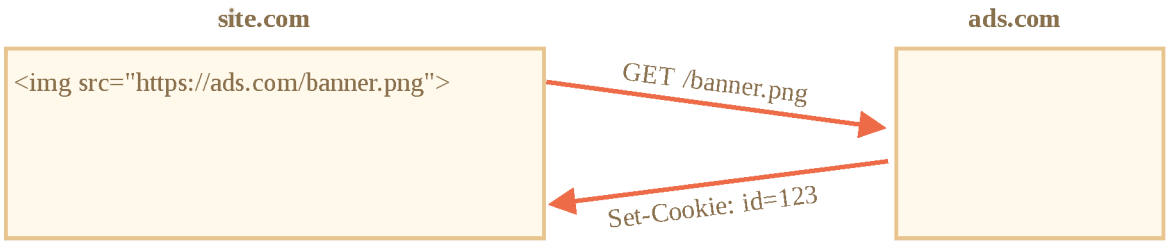
Всё вместе: cookie.js.

Приложение: Сторонние куки

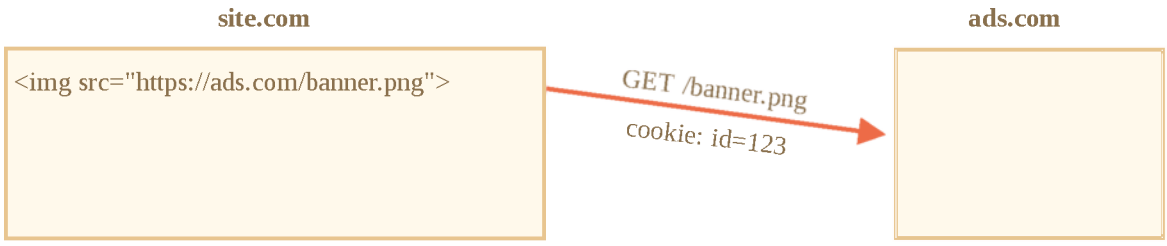
Куки называются сторонними, если они размещены с домена, отличающегося страницы, которую посещает пользователь.

Например:

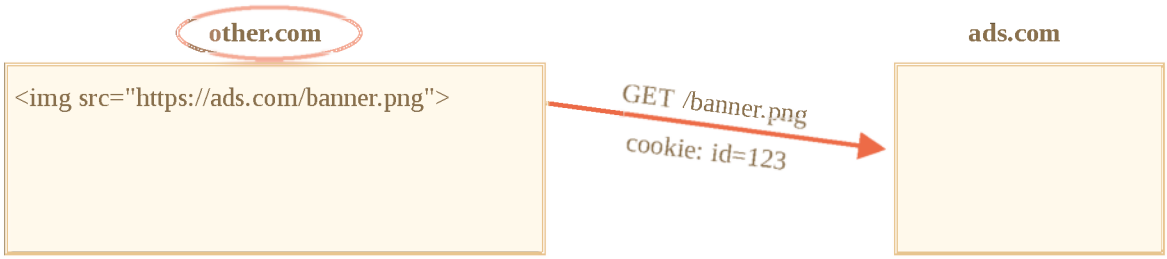
1. Страница `site.com` загружает баннер с другого сайта: `<img src="https://ads.com/banner.png">`.
2. Вместе с баннером удалённый сервер `ads.com` может установить заголовок `Set-Cookie` с куки, например, `id=1234`. Такие куки создаются с домена `ads.com` и будут видны только на сайте `ads.com`:



3. В следующий раз при доступе к `ads.com` удалённый сервер получит куки `id` и распознает пользователя:



4. Что ещё более важно, когда пользователь переходит с `site.com` на другой сайт `other.com`, на котором тоже есть баннер, то `ads.com` получит куки, так как они принадлежат `ads.com`, таким образом `ads.com` распознает пользователя и может отслеживать его перемещения между сайтами:



Сторонние куки в силу своей специфики обычно используются для целей отслеживания посещаемых пользователем страниц и показа рекламы. Они привязаны к исходному домену, поэтому ads.com может отслеживать одного и того же пользователя на разных сайтах, если оттуда идёт обращение к нему,

Естественно, некоторым пользователя не нравится, когда их отслеживают, поэтому браузеры позволяют отключать такие куки.

Кроме того, некторые современные браузеры используют специальные политики для таких куки:

- Safari вообще не разрешает сторонние куки.
- У Firefox есть «чёрный список» сторонних доменов, чьи сторонние куки он блокирует.

На заметку:

Если мы загружаем скрипт со стороннего домена, например `<script src="https://google-analytics.com/analytics.js">`, и этот скрипт использует `document.cookie`, чтобы установить куки, то такое куки не является сторонним.

Если скрипт устанавливает куки, то нет разницы откуда был загружен скрипт – куки принадлежит домену текущей веб-страницы.

Приложение: GDPR

Эта тема вообще не связана с JavaScript, но следует её иметь в виду при установке куки.

В Европе существует законодательство под названием GDPR, которое устанавливает для сайтов ряд правил, обеспечивающих конфиденциальность пользователей. И одним из таких правил является требование явного разрешения от пользователя на использование отслеживающих куки.

Обратите внимание, это относится только куки, используемым для отслеживания/идентификации/авторизации.

То есть, если мы установим куки, которые просто сохраняют некоторую информацию, но не отслеживают и не идентифицируют пользователя, то мы свободны от этого правила.

Но если мы собираемся установить куки с информацией об аутентификации или с идентификатором отслеживания, то пользователь должен явно разрешить это.

Есть два основных варианта как сайты следуют GDPR. Вы наверняка уже видели их в сети:

1. Если сайт хочет установить куки для отслеживания только для авторизованных пользователей.

   То в регистрационной форме должен быть установлен флажок «принять политику конфиденциальности» (которая определяет, как используются куки), пользователь должен установить его, и только тогда сайт сможет использовать авторизационные куки.

2. Если сайт хочет установить куки для отслеживания всем пользователям.

   Чтобы сделать это законно, сайт показывает модально окно для пользователей, которые зашли в первый раз, и требует от них согласие на использование куки. Затем сайт может установить такие куки и показать пользователю содержимое страницы. Хотя это создаёт неудобства для новых посетителей – никому не нравится наблюдать модальные окна вместо

контента. Но GDPR в данной ситуации требует явного согласия пользователя.

GDPR касается не только куки, но и других вопросов, связанных с конфиденциальностью, которые выходят за рамки материала этой главы.

Итого

`document.cookie` предоставляет доступ к куки

- операция записи изменяет только то куки, которое было указано.
- имя и значение куки должны быть закодированы.
- одно куки вмещает до 4kb данных, разрешается более 20 куки на сайт (зависит от браузера).

Настройки куки:

- `path=/`, по умолчанию устанавливается текущий путь, делает куки видимым только по указанному пути и ниже.
- `domain=site.com`, по умолчанию куки видно только на текущем домене, если явно указан домен, то куки видно и на поддоменах.
- `expires` или `max-age` устанавливает дату истечения срока действия, без них куки умрёт при закрытии браузера.
- `secure` делает куки доступным только при использовании HTTPS.
- `samesite` запрещает браузеру отправлять куки с запросами, поступающими извне, помогает предотвратить XSRF-атаки.

Дополнительно:

- Сторонние куки могут быть запрещены браузером, например Safari делает это по умолчанию.
- Установка отслеживающих куки пользователям из стран ЕС требует их явного согласия на это в соответствии с законодательством GDPR.

LocalStorage, sessionStorage

Объекты веб-хранилища `localStorage` и `sessionStorage` позволяют хранить пары ключ/значение в браузере.

Что в них важно – данные, которые в них записаны, сохраняются после обновления страницы (в случае `sessionStorage`) и даже после перезапуска браузера (при использовании `localStorage`). Скоро мы это увидим.

Но ведь у нас уже есть куки. Зачем тогда эти объекты?

- В отличие от куки, объекты веб-хранилища не отправляются на сервер при каждом запросе. Поэтому мы можем хранить гораздо больше данных. Большинство браузеров могут сохранить как минимум 2 мегабайта данных (или больше), и этот размер можно поменять в настройках.
- Ещё одно отличие от куки – сервер не может манипулировать объектами хранилища через HTTP-заголовки. Всё делается при помощи JavaScript.
- Хранилище привязано к источнику (домен/протокол/порт). Это значит, что разные протоколы или поддомены определяют разные объекты хранилища, и они не могут получить доступ к данным друг друга.

Объекты хранилища `localStorage` и `sessionStorage` предоставляют одинаковые методы и свойства:

- `setItem(key, value)` – сохранить пару ключ/значение.
- `getItem(key)` – получить данные по ключу `key`.
- `removeItem(key)` – удалить данные с ключом `key`.
- `clear()` – удалить всё.
- `key(index)` – получить ключ на заданной позиции.
- `length` – количество элементов в хранилище.

Как видим, интерфейс похож на `Map` (`setItem/getItem/removeItem`), но также запоминается порядок элементов, и можно получить доступ к элементу по индексу – `key(index)`.

Давайте посмотрим, как это работает.

Демо localStorage

Основные особенности `localStorage`:

- Этот объект один на все вкладки и окна в рамках источника (один и тот же домен/протокол/порт).
- Данные не имеют срока давности, по которому истекают и удаляются. Сохраняются после перезапуска браузера и даже ОС.

Например, если запустить этот код…

```
localStorage.setItem('test', 1);
```

…И закрыть/открыть браузер или открыть ту же страницу в другом окне, то можно получить данные следующим образом:

```
alert( localStorage.getItem('test') ); // 1
```

Нам достаточно находиться на том же источнике (домен/протокол/порт), при этом URL-путь может быть разным.

Объект `localStorage` доступен всем окнам из одного источника, поэтому, если мы устанавливаем данные в одном окне, изменения становятся видимыми в другом.

Доступ как к обычному объекту

Также можно получать/записывать данные, как в обычный объект:

```
// установить значение для ключа
localStorage.test = 2;

// получить значение по ключу
alert( localStorage.test ); // 2

// удалить ключ
delete localStorage.test;
```

Это возможно по историческим причинам и, как правило, работает, но обычно не рекомендуется, потому что:

1. Если ключ генерируется пользователем, то он может быть каким угодно, включая `length` или `toString` или другой встроенный метод `localStorage`. В этом случае `getItem/setItem` сработают нормально, а вот чтение/запись как свойства объекта не пройдут:

   ```
   let key = 'length';
   localStorage[key] = 5; // Ошибка, невозможно установить length
   ```

2. Когда мы модифицируем данные, то срабатывает событие `storage`. Но это событие не происходит при записи без `setItem`, как свойства объекта. Мы увидим это позже в этой главе.

Перебор ключей

Методы, которые мы видим, позволяют читать/писать/удалять данные. А как получить все значения или ключи?

К сожалению, объекты веб-хранилища нельзя перебрать в цикле, они не итерируемы.

Но можно пройти по ним, как по обычным массивам:

```
for(let i=0; i<localStorage.length; i++) {
  let key = localStorage.key(i);
  alert(`${key}: ${localStorage.getItem(key)}`);
}
```

Другой способ – использовать цикл, как по обычному объекту `for key in localStorage`.

Здесь перебираются ключи, но вместе с этим выводятся несколько встроенных полей, которые нам не нужны:

```
// bad try
for(let key in localStorage) {
  alert(key); // покажет getItem, setItem и другие встроенные свойства
}
```

…Поэтому нам нужно либо отфильтровать поля из прототипа проверкой `hasOwnProperty`:

```
for(let key in localStorage) {
  if (!localStorage.hasOwnProperty(key)) {
    continue; // пропустит такие ключи, как "setItem", "getItem" и так далее
  }
  alert(`${key}: ${localStorage.getItem(key)}`);
}
```

…Либо просто получить «собственные» ключи с помощью Object.keys, а затем при необходимости вывести их при помощи цикла:

```
let keys = Object.keys(localStorage);
for(let key of keys) {
  alert(`${key}: ${localStorage.getItem(key)}`);
}
```

Последнее работает, потому что `Object.keys` возвращает только ключи, принадлежащие объекту, игнорируя прототип.

Только строки

Обратите внимание, что ключ и значение должны быть строками.

Если мы используем любой другой тип, например число или объект, то он автоматически преобразуется в строку:

```
sessionStorage.user = {name: "John"};
alert(sessionStorage.user); // [object Object]
```

Мы можем использовать `JSON` для хранения объектов:

```
sessionStorage.user = JSON.stringify({name: "John"});

// немного позже
let user = JSON.parse( sessionStorage.user );
alert( user.name ); // John
```

Также возможно привести к строке весь объект хранилища, например для отладки:

```
// для JSON.stringify добавлены параметры форматирования, чтобы объект выглядел лучш
alert( JSON.stringify(localStorage, null, 2) );
```

sessionStorage

Объект `sessionStorage` используется гораздо реже, чем `localStorage`.

Свойства и методы такие же, но есть существенные ограничения:

- `sessionStorage` существует только в рамках текущей вкладки браузера.
  - Другая вкладка с той же страницей будет иметь другое хранилище.
  - Но оно разделяется между ифреймами на той же вкладке (при условии, что они из одного и того же источника).
- Данные продолжают существовать после перезагрузки страницы, но не после закрытия/открытия вкладки.

Давайте посмотрим на это в действии.

Запустите этот код…

```
sessionStorage.setItem('test', 1);
```

…И обновите страницу. Вы всё ещё можете получить данные:

```
alert( sessionStorage.getItem('test') ); // после обновления: 1
```

…Но если вы откроете ту же страницу в другой вкладке и попробуете получить данные снова, то код выше вернёт `null`, что значит «ничего не найдено».

Так получилось, потому что `sessionStorage` привязан не только к источнику, но и к вкладке браузера. Поэтому `sessionStorage` используется нечасто.

Событие storage

Когда обновляются данные в `localStorage` или `sessionStorage`, генерируется событие storage со следующими свойствами:

- `key` – ключ, который обновился (`null`, если вызван `.clear()`).
- `oldValue` – старое значение (`null`, если ключ добавлен впервые).
- `newValue` – новое значение (`null`, если ключ был удалён).
- `url` – url документа, где произошло обновление.
- `storageArea` – объект `localStorage` или `sessionStorage`, где произошло обновление.

Важно: событие срабатывает на всех остальных объектах `window`, где доступно хранилище, кроме того окна, которое его вызвало.

Давайте уточним.

Представьте, что у вас есть два окна с одним и тем же сайтом. Хранилище `localStorage` разделяется между ними.

Теперь, если оба окна слушают `window.onstorage`, то каждое из них будет реагировать на обновления, произошедшие в другом окне.

```
// срабатывает при обновлениях, сделанных в том же хранилище из других документов
window.onstorage = event => {
  if (event.key != 'now') return;
  alert(event.key + ':' + event.newValue + " at " + event.url);
};

localStorage.setItem('now', Date.now());
```

Обратите внимание, что событие также содержит: `event.url` – url-адрес документа, в котором данные обновились.

Также `event.storageArea` содержит объект хранилища – событие одно и то же для `sessionStorage` и `localStorage`, поэтому `event.storageArea` ссылается на то хранилище, которое было изменено. Мы можем захотеть что-то записать в ответ на изменения.

Это позволяет разным окнам одного источника обмениваться сообщениями.

Современные браузеры также поддерживают Broadcast channel API специальный API для связи между окнами одного источника, он более полнофункциональный, но менее поддерживаемый. Существуют библиотеки (полифилы), которые эмулируют это API на основе `localStorage` и делают его доступным везде.

Итого

Объекты веб-хранилища `localStorage` и `sessionStorage` позволяют хранить пары ключ/значение в браузере.

- `key` и `value` должны быть строками.
- Лимит 2 Мб+, зависит от браузера.
- Данные не имеют «времени истечения».
- Данные привязаны к источнику (домен/протокол/порт).

`localStorage`	`sessionStorage`
Совместно используется между всеми вкладками и окнами с одинаковым источником	Разделяется в рамках вкладки браузера, среди ифреймов из того же источника
«Переживает» перезапуск браузера	«Переживает» перезагрузку страницы (но не закрытие вкладки)

API:

- `setItem(key, value)` – сохранить пару ключ/значение.
- `getItem(key)` – получить данные по ключу `key`.
- `removeItem(key)` – удалить значение по ключу `key`.
- `clear()` – удалить всё.
- `key(index)` – получить ключ на заданной позиции.
- `length` – количество элементов в хранилище.
- Используйте `Object.keys` для получения всех ключей.
- Можно обращаться к ключам как к обычным свойствам объекта, в этом случае событие `storage` не срабатывает.

Событие storage:

- Срабатывает при вызове `setItem`, `removeItem`, `clear`.
- Содержит все данные об произошедшем обновлении (`key/oldValue/newValue`), `url` документа и объект хранилища `storageArea`.
- Срабатывает на всех объектах `window`, которые имеют доступ к хранилищу, кроме того, где оно было сгенерировано (внутри вкладки для `sessionStorage`, глобально для `localStorage`).

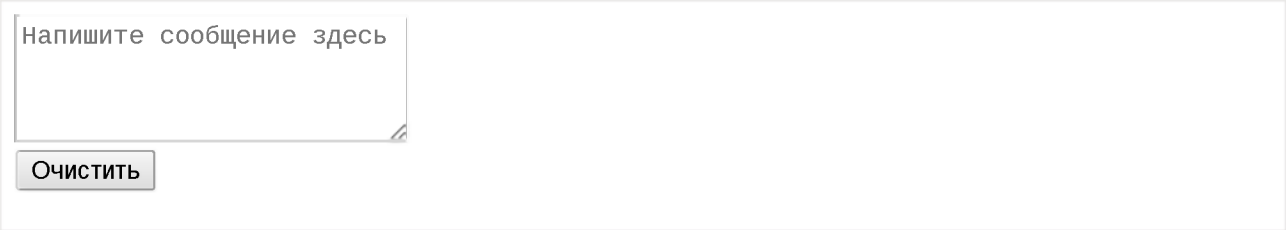
Задачи

Автосохранение поля формы

Создайте поле `textarea`, значение которого будет автоматически сохраняться при каждом его изменении.

Когда пользователь закроет страницу и потом откроет её заново он должен увидеть последнее введённое значение.

Вот пример:



Открыть песочницу для задачи.

К решению

IndexedDB

IndexedDB – это встроенная база данных, более мощная, чем `localStorage`.

- Хранилище ключей/значений: доступны несколько типов ключей, а значения могут быть (почти) любыми.
- Поддерживает транзакции для надёжности.
- Поддерживает запросы в диапазоне ключей и индексы.
- Позволяет хранить больше данных, чем `localStorage`.

Для традиционных клиент-серверных приложений эта мощность обычно чрезмерна. IndexedDB предназначена для оффлайн приложений, можно совмещать с ServiceWorkers и другими технологиями.

Интерфейс для IndexedDB, описанный в спецификации https://www.w3.org/TR/IndexedDB , основан на событиях.

Мы также можем использовать `async/await` с помощью обёртки, которая основана на промисах, например https://github.com/jakearchibald/idb . Это очень удобно, но обёртка не идеальна, она не может полностью заменить события. Поэтому мы начнём с событий, а затем, когда разберёмся в IndexedDB, рассмотрим и обёртку.

Открыть базу данных

Для начала работы с IndexedDB нужно открыть базу данных.

Синтаксис:

```
let openRequest = indexedDB.open(name, version);
```

- `name` – название базы данных, строка.
- `version` – версия базы данных, положительное целое число, по умолчанию `1` (объясняется ниже).

У нас может быть множество баз данных с различными именами, но все они существуют в контексте текущего источника (домен/протокол/порт). Разные сайты не могут получить доступ к базам данных друг друга.

После этого вызова необходимо назначить обработчик событий для объекта `openRequest`:

- `success`: база данных готова к работе, готов «объект базы данных» `openRequest.result`, его следует использовать для дальнейших вызовов.
- `error`: не удалось открыть базу данных.
- `upgradeneeded`: база открыта, но её схема устарела (см. ниже).

IndexedDB имеет встроенный механизм «версионирования схемы», который отсутствует в серверных базах данных.

В отличие от серверных баз данных, IndexedDB работает на стороне клиента, в браузере, и у нас нет прямого доступа к данным. Но когда мы публикуем новую версию нашего приложения, возможно, нам понадобиться обновить базу данных.

Если локальная версия базы данных меньше, чем версия, определённая в `open`, то сработает специальное событие `upgradeneeded`, и мы сможем сравнить версии и обновить структуры данных по мере необходимости.

Это событие также сработает, если базы данных ещё не существует, так что в этом обработчике мы можем выполнить инициализацию.

Например, когда мы впервые публикуем наше приложение, мы открываем базу данных с версией `1` и выполняем инициализацию в обработчике `upgradeneeded`:

```
let openRequest = indexedDB.open("store", 1);

openRequest.onupgradeneeded = function() {
  // срабатывает, если на клиенте нет базы данных
  // ...выполнить инициализацию...
};

openRequest.onerror = function() {
  console.error("Error", openRequest.error);
};

openRequest.onsuccess = function() {
  let db = openRequest.result;
  // продолжить работу с базой данных, используя объект db
};
```

Когда мы публикуем вторую версию:

```
let openRequest = indexedDB.open("store", 2);

// проверить существование указанной версии базы данных, обновить по мере необходим
openRequest.onupgradeneeded = function() {
  // the existing database version is less than 2 (or it doesn't exist)
  let db = openRequest.result;
  switch(db.version) { // существующая (старая) версия базы данных
    case 0:
      // версия 0 означает, что на клиенте нет базы данных
      // выполнить инициализацию
    case 1:
      // на клиенте версия базы данных 1
      // обновить
  }
};
```

Таким образом, в `openRequest.onupgradeneeded` мы обновляем базу данных. Скоро подробно увидим, как это делается. А после того, как этот

обработчик завершится без ошибок, сработает `openRequest.onsuccess`.

После `openRequest.onsuccess` у нас есть объект базы данных в `openRequest.result`, который мы будем использовать для дальнейших операций.

Удалить базу данных:

```
let deleteRequest = indexedDB.deleteDatabase(name)
// deleteRequest.onsuccess/onerror отслеживает результат
```

⚠️ **А что, если открыть предыдущую версию?**

Что если мы попробуем открыть базу с более низкой версией, чем текущая? Например, на клиенте база версии 3, а мы вызывает `open(...2)`.

Возникнет ошибка, сработает `openRequest.onerror`.

Такое может произойти, если посетитель загрузил устаревший код, например, из кеша прокси. Нам следует проверить `db.version` и предложить ему перезагрузить страницу. А также проверить наши кеширующие заголовки, убедиться, что посетитель никогда не получит устаревший код.

Проблема параллельного обновления

Раз уж мы говорим про версионирование, рассмотрим связанную с этим небольшую проблему.

Допустим, посетитель открыл наш сайт во вкладке браузера, с базой версии 1.

Затем мы выкатили обновление, и тот же посетитель открыл наш сайт в другой вкладке. Так что есть две вкладки, на которых открыт наш сайт, но в одной открыто соединение с базой версии 1, а другая пытается обновить версию базы в обработчике `upgradeneeded`.

Проблема заключается в том, что база данных всего одна на две вкладки, так как это один и тот же сайт, один источник. И она не может быть одновременно версии 1 и 2. Чтобы обновить на версию 2, все соединения к версии 1 должны быть закрыты.

Чтобы это можно было организовать, при попытке обновления на объекте базы возникает событие `versionchange`. Нам нужно слушать его и закрыть соединение к базе (а также, возможно, предложить пользователю перезагрузить страницу, чтобы получить обновлённый код).

Если мы его не закроем, то второе, новое соединение будет заблокировано с событием `blocked` вместо `success`.

Код, который это делает:

```
let openRequest = indexedDB.open("store", 2);

openRequest.onupgradeneeded = ...;
openRequest.onerror = ...;

openRequest.onsuccess = function() {
  let db = openRequest.result;

  db.onversionchange = function() {
    db.close();
    alert("База данных устарела, пожалуста, перезагрузите страницу.")
  };

  // ...база данных доступна как объект db...
};

openRequest.onblocked = function() {
  // есть другое соединение к той же базе
  // и оно не было закрыто после срабатывания на нём db.onversionchange
};
```

Здесь мы делаем две вещи:

1. Добавляем обработчик `db.onversionchange` после успешного открытия базы, чтобы узнать о попытке параллельного обновления.
2. Добавляем обработчик `openRequest.onblocked` для ситуаций, когда старое соединение не было закрыто. Такого не произойдёт, если мы закрываем его в `db.onversionchange`.

Есть и другие варианты. Например, мы можем более «мягко» закрыть соединение в `db.onversionchange`, предложить пользователю сохранить данные перед этим. Новое обновляющее соединение будет заблокировано сразу после того как обработчик `db.onversionchange` завершится, не закрыв соединение, и мы можем в новой вкладке попросить посетителя закрыть старые для обновления.

Такой конфликт при обновлении происходит редко, но мы должны как-то его обрабатывать, хотя бы поставить обработчик `onblocked`, чтобы наш скрипт не «умирал» молча, удивляя посетителя.

Хранилище объектов

Чтобы сохранить что-то в IndexedDB, нам нужно *хранилище объектов*.

Хранилище объектов – это основная концепция IndexedDB. В других базах данных это «таблицы» или «коллекции». Здесь хранятся данные. В базе данных может быть множество хранилищ: одно для пользователей, другое для товаров и так далее.

Несмотря на то, что название – «хранилище объектов», примитивы тоже могут там храниться.

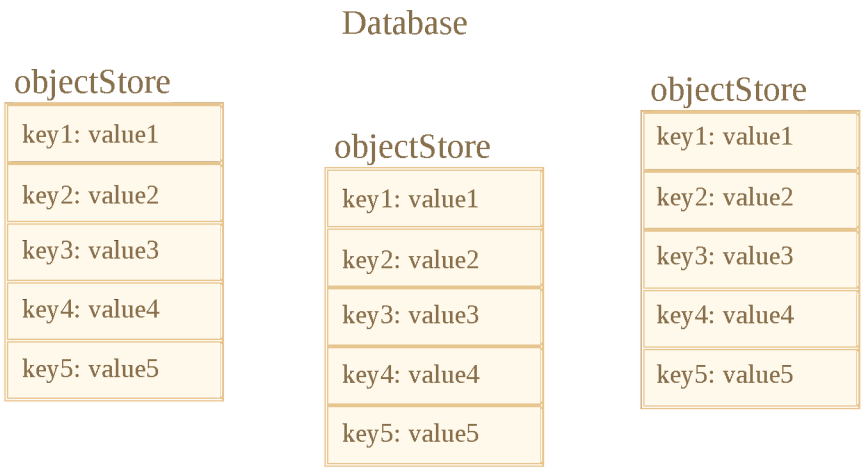
Мы можем хранить почти любое значение, в том числе сложные объекты.

IndexedDB использует стандартный алгоритм сериализации для клонирования и хранения объекта. Это как `JSON.stringify`, но более мощный, способный хранить гораздо больше типов данных.

Пример объекта, который нельзя сохранить: объект с циклическими ссылками. Такие объекты не сериализуемы. `JSON.stringify` также выдаст ошибку при сериализации.

Каждому значению в хранилище должен соответствовать уникальный ключ.

Ключ должен быть одним из следующих типов: number, date, string, binary или array. Это уникальный идентификатор: по ключу мы можем искать/удалять/обновлять значения.



Как мы видим, можно указать ключ при добавлении значения в хранилище, аналогично `localStorage`. Но когда мы храним объекты, IndexedDB позволяет установить свойство объекта в качестве ключа, что гораздо удобнее. Или мы можем автоматически сгенерировать ключи.

Но для начала нужно создать хранилище.

Синтаксис для создания хранилища объектов:

```
db.createObjectStore(name[, keyOptions]);
```

Обратите внимание, что операция является синхронной, использование `await` не требуется.

- `name` – это название хранилища, например `"books"` для книг,
- `keyOptions` – это необязательный объект с одним или двумя свойствами:
  - `keyPath` – путь к свойству объекта, которое IndexedDB будет использовать в качестве ключа, например `id`.
  - `autoIncrement` – если `true`, то ключ будет формироваться автоматически для новых объектов, как постоянно увеличивающееся число.

Если при создании хранилища не указать `keyOptions`, то нам потребуется явно указать ключ позже, при сохранении объекта.

Например, это хранилище объектов использует свойство `id` как ключ:

```
db.createObjectStore('books', {keyPath: 'id'});
```

Хранилище объектов можно создавать/изменять только при обновлении версии базы данных в обработчике `upgradeneeded`.

Это техническое ограничение. Вне обработчика мы сможем добавлять/удалять/обновлять данные, но хранилища объектов могут быть созданы/удалены/изменены только во время обновления версии базы данных.

Для обновления версии базы есть два основных подхода:

1. Мы можем реализовать функции обновления по версиям: с 1 на 2, с 2 на 3 и т.д. Потом в `upgradeneeded` сравнить версии (например, была 2, сейчас 4) и запустить операции обновления для каждой промежуточной версии (2 на 3, затем 3 на 4).
2. Или мы можем взять список существующих хранилищ объектов, используя `db.objectStoreNames`. Этот объект является DOMStringList, в нём есть метод `contains(name)`, используя который можно проверить существование хранилища. Посмотреть, какие хранилища есть и создать те, которых нет.

Для простых баз данных второй подход может быть проще и предпочтительнее.

Вот демонстрация второго способа:

```
let openRequest = indexedDB.open("db", 2);

// создаём хранилище объектов для books, если ещё не существует
```

```
openRequest.onupgradeneeded = function() {
  let db = openRequest.result;
  if (!db.objectStoreNames.contains('books')) { // if there's no "books" store
    db.createObjectStore('books', {keyPath: 'id'}); // create it
  }
};
```

Чтобы удалить хранилище объектов:

```
db.deleteObjectStore('books')
```

Транзакции

Термин «транзакция» является общеизвестным, транзакции используются во многих видах баз данных.

Транзакция – это группа операций, которые должны быть или все выполнены, или все не выполнены (всё или ничего).

Например, когда пользователь что-то покупает, нам нужно:

1. Вычесть деньги с его счёта.
2. Отправить ему покупку.

Будет очень плохо, если мы успеем завершить первую операцию, а затем что-то пойдёт не так, например отключат электричество, и мы не сможем завершить вторую операцию. Обе операции должны быть успешно завершены (покупка сделана, отлично!) или необходимо отменить обе операции (в этом случае пользователь сохранит свои деньги и может попытаться купить ещё раз).

Транзакции гарантируют это.

Все операции с данными в IndexedDB могут быть сделаны только внутри транзакций.

Для начала транзакции:

```
db.transaction(store[, type]);
```

- `store` – это название хранилища, к которому транзакция получит доступ, например, `"books"`. Может быть массивом названий, если нам нужно предоставить доступ к нескольким хранилищам.
- `type` – тип транзакции, один из:

- `readonly` – только чтение, по умолчанию.
- `readwrite` – только чтение и запись данных, создание/удаление самих хранилищ объектов недоступно.

Есть ещё один тип транзакций: `versionchange`. Такие транзакции могут делать любые операции, но мы не можем создать их вручную. IndexedDB автоматически создаёт транзакцию типа `versionchange`, когда открывает базу данных, для обработчика `updateneeded`. Вот почему это единственное место, где мы можем обновлять структуру базы данных, создавать/удалять хранилища объектов.

Почему существует несколько типов транзакций?

Производительность является причиной, почему транзакции необходимо помечать как `readonly` или `readwrite`.

Несколько readonly транзакций могут одновременно работать с одним и тем же хранилищем объектов, а readwrite транзакций – не могут. Транзакции типа readwrite «блокируют» хранилище для записи. Следующая такая транзакция должна дождаться выполнения предыдущей, перед тем как получит доступ к тому же самому хранилищу.

После того, как транзакция будет создана, мы можем добавить элемент в хранилище, вот так:

```
let transaction = db.transaction("books", "readwrite"); // (1)

// получить хранилище объектов для работы с ним
let books = transaction.objectStore("books"); // (2)

let book = {
  id: 'js',
  price: 10,
  created: new Date()
};

let request = books.add(book); // (3)

request.onsuccess = function() { // (4)
  console.log("Книга добавлена в хранилище", request.result);
};

request.onerror = function() {
  console.log("Ошибка", request.error);
};
```

Мы сделали четыре шага:

1. Создать транзакцию и указать все хранилища, к которым необходим доступ, строка `(1)`.
2. Получить хранилище объектов, используя `transaction.objectStore(name)`, строка `(2)`.
3. Выполнить запрос на добавление элемента в хранилище объектов `books.add(book)`, строка `(3)`.
4. …Обработать результат запроса `(4)`, затем мы можем выполнить другие запросы и так далее.

Хранилища объектов поддерживают два метода для добавления значений:

- **put(value, [key])** Добавляет значение `value` в хранилище. Ключ `key` необходимо указать, если при создании хранилища объектов не было указано свойство `keyPath` или `autoIncrement`. Если уже есть значение с таким же ключом, то оно будет заменено.
- **add(value, [key])** То же, что `put`, но если уже существует значение с таким ключом, то запрос не выполнится, будет сгенерирована ошибка с названием `"ConstraintError"`.

Аналогично открытию базы, мы отправляем запрос: `books.add(book)` и после ожидаем события `success/error`.

- `request.result` для `add` является ключом нового объекта.
- Ошибка находится в `request.error` (если есть).

Автоматическая фиксация транзакций

В примере выше мы запустили транзакцию и выполнили запрос `add`. Но, как говорилось ранее, транзакция может включать в себя несколько запросов, которые все вместе должны либо успешно завершиться, либо нет. Как нам закончить транзакцию, обозначить, что больше запросов в ней не будет?

Короткий ответ: этого не требуется.

В следующей 3.0 версии спецификации, вероятно, будет возможность вручную завершить транзакцию, но сейчас, в версии 2.0, такой возможности нет.

Когда все запросы завершены и очередь микрозадач пуста, тогда транзакция завершится автоматически.

Как правило, это означает, что транзакция автоматически завершается, когда выполнились все её запросы и завершился текущий код.

Таким образом, в приведённом выше примере не требуется никакой специальный вызов, чтобы завершить транзакцию.

Такое автозавершение транзакций имеет важный побочный эффект. Мы не можем вставить асинхронную операцию, такую как `fetch` или `setTimeout` в середину транзакции. IndexedDB никак не заставит транзакцию «висеть» и ждать их выполнения.

В приведённом ниже коде в запросе `request2` в строке с `(*)` будет ошибка, потому что транзакция уже завершена, больше нельзя выполнить в ней запрос:

```
let request1 = books.add(book);

request1.onsuccess = function() {
  fetch('/').then(response => {
    let request2 = books.add(anotherBook); // (*)
    request2.onerror = function() {
      console.log(request2.error.name); // TransactionInactiveError
    };
  });
};
```

Всё потому, что `fetch` является асинхронной операцией, макрозадачей. Транзакции завершаются раньше, чем браузер приступает к выполнению макрозадач.

Авторы спецификации IndexedDB из соображений производительности считают, что транзакции должны завершаться быстро.

В частности, `readwrite` транзакции «блокируют» хранилища от записи. Таким образом, если одна часть приложения инициирует `readwrite` транзакцию в хранилище объектов `books`, то другая часть приложения, которая хочет сделать то же самое, должна ждать: новая транзакция «зависает» до завершения первой. Это может привести к странным задержкам, если транзакции слишком долго выполняются.

Что же делать?

В приведённом выше примере мы могли бы запустить новую транзакцию `db.transaction` перед новым запросом `(*)`.

Но ещё лучше выполнять операции вместе, в рамках одной транзакции: отделить транзакции IndexedDB от других асинхронных операций.

Сначала сделаем `fetch`, подготовим данные, если нужно, затем создадим транзакцию и выполним все запросы к базе данных.

Чтобы поймать момент успешного выполнения, мы можем повесить обработчик на событие `transaction.oncomplete`:

```
let transaction = db.transaction("books", "readwrite");

// ...выполнить операции...

transaction.oncomplete = function() {
  console.log("Транзакция выполнена");
};
```

Только `complete` гарантирует, что транзакция сохранена целиком. По отдельности запросы могут выполниться, но при финальной записи что-то может пойти не так (ошибка ввода-вывода, проблема с диском, например).

Чтобы вручную отменить транзакцию, выполните:

```
transaction.abort();
```

Это отменит все изменения, сделанные запросами в транзакции, и сгенерирует событие `transaction.onabort`.

Обработка ошибок

Запросы на запись могут выполниться неудачно.

Мы должны быть готовы к этому, не только из-за возможных ошибок на нашей стороне, но и по причинам, которые не связаны с транзакцией. Например, размер хранилища может быть превышен. И мы должны быть готовы обработать такую ситуацию.

При ошибке в запросе соответствующая транзакция отменяется полностью, включая изменения, сделанные другими её запросами.

Если мы хотим продолжить транзакцию (например, попробовать другой запрос без отмены изменений), это также возможно. Для этого в обработчике `request.onerror` следует вызвать `event.preventDefault()`.

В примере ниже новая книга добавляется с тем же ключом (`id`), что и существующая. Метод `store.add` генерирует в этом случае ошибку `"ConstraintError"`. Мы обрабатываем её без отмены транзакции:

```
let transaction = db.transaction("books", "readwrite");

let book = { id: 'js', price: 10 };

let request = transaction.objectStore("books").add(book);

request.onerror = function(event) {
```

```
  // ConstraintError возникает при попытке добавить объект с ключом, который уже су
  if (request.error.name == "ConstraintError") {
    console.log("Книга с таким id уже существует"); // обрабатываем ошибку
    event.preventDefault(); // предотвращаем отмену транзакции
    // ...можно попробовать использовать другой ключ...
  } else {
    // неизвестная ошибка
    // транзакция будет отменена
  }
};

transaction.onabort = function() {
  console.log("Ошибка", transaction.error);
};
```

Делегирование событий

Нужны ли обработчики onerror/onsuccess для каждого запроса? Не всегда. Мы можем использовать делегирование событий.

События IndexedDB всплывают: `запрос` → `транзакция` → `база данных`.

Все события являются DOM-событиями с фазами перехвата и всплытия, но обычно используется только всплытие.

Поэтому мы можем перехватить все ошибки, используя обработчик `db.onerror`, для оповещения пользователя или других целей:

```
db.onerror = function(event) {
  let request = event.target; // запрос, в котором произошла ошибка

  console.log("Ошибка", request.error);
};
```

…А если мы полностью обработали ошибку? В этом случае мы не хотим сообщать об этом.

Мы можем остановить всплытие и, следовательно, `db.onerror`, используя `event.stopPropagation()` в `request.onerror`.

```
request.onerror = function(event) {
  if (request.error.name == "ConstraintError") {
    console.log("Книга с таким id уже существует"); // обрабатываем ошибку
    event.preventDefault(); // предотвращаем отмену транзакции
    event.stopPropagation(); // предотвращаем всплытие ошибки
  } else {
    // ничего не делаем
    // транзакция будет отменена
```

```
    // мы можем обработать ошибку в transaction.onabort
  }
};
```

Поиск по ключам

Есть два основных вида поиска в хранилище объектов:

1. По ключу или по диапазону ключей. То есть: по `book.id` в хранилище «books».
2. По полям объекта, например, `book.price`.

Сначала давайте разберёмся с ключами и диапазоном ключей `(1)`.

Методы поиска поддерживают либо точные ключи, либо так называемые «запросы с диапазоном» – IDBKeyRange объекты, которые задают «диапазон ключей».

Диапазоны создаются с помощью следующих вызовов:

- `IDBKeyRange.lowerBound(lower, [open])` означает: `>lower` (или `≥lower`, если `open` это true)
- `IDBKeyRange.upperBound(upper, [open])` означает: `<upper` (или `≤upper`, если `open` это true)
- `IDBKeyRange.bound(lower, upper, [lowerOpen], [upperOpen])` означает: между `lower` и `upper`, включительно, если соответствующий `open` равен `true`.
- `IDBKeyRange.only(key)` – диапазон, который состоит только из одного ключа `key`, редко используется.

Все методы поиска принимают аргумент `query`, который может быть либо точным ключом, либо диапазоном ключей:

- `store.get(query)` – поиск первого значения по ключу или по диапазону.
- `store.getAll([query], [count])` – поиск всех значений, можно ограничить, передав `count`.
- `store.getKey(query)` – поиск первого ключа, который удовлетворяет запросу, обычно передаётся диапазон.
- `store.getAllKeys([query], [count])` – поиск всех ключей, которые удовлетворяют запросу, обычно передаётся диапазон, возможно ограничить поиск, передав `count`.
- `store.count([query])` – получить общее количество ключей, которые удовлетворяют запросу, обычно передаётся диапазон.

Например, в хранилище у нас есть множество книг. Помните, поле `id` является ключом, поэтому все эти методы могут искать по ключу `id`.

Примеры запросов:

```
// получить одну книгу
books.get('js')

// получить все книги с 'css' < id < 'html'
books.getAll(IDBKeyRange.bound('css', 'html'))

// получить книги с 'html' <= id
books.getAll(IDBKeyRange.lowerBound('html', true))

// получить все книги
books.getAll()

// получить все ключи: id >= 'js'
books.getAllKeys(IDBKeyRange.lowerBound('js', true))
```

Хранилище объектов всегда отсортировано

Хранилище объектов внутренне сортирует значения по ключам.

Поэтому запросы, которые возвращают много значений, всегда возвращают их в порядке сортировки по ключу.

Поиск по индексированному полю

Для поиска по другим полям объекта нам нужно создать дополнительную структуру данных, называемую «индекс» (index).

Индекс является «расширением» к хранилищу, которое отслеживает данное поле объекта. Для каждого значения этого поля хранится список ключей для объектов, которые имеют это значение. Ниже будет более подробная картина.

Синтаксис:

```
objectStore.createIndex(name, keyPath, [options]);
```

- **`name`** – название индекса,
- **`keyPath`** – путь к полю объекта, которое индекс должен отслеживать (мы собираемся сделать поиск по этому полю),
- **`option`** – необязательный объект со свойствами:

- **`unique`** – если true, тогда в хранилище может быть только один объект с заданным значением в `keyPath`. Если мы попытаемся добавить дубликат, то индекс сгенерирует ошибку.
- **`multiEntry`** – используется только, если `keyPath` является массивом. В этом случае, по умолчанию, индекс обрабатывает весь массив как ключ. Но если мы укажем true в `multiEntry`, тогда индекс будет хранить список объектов хранилища для каждого значения в этом массиве. Таким образом, элементы массива становятся ключами индекса.

В нашем примере мы храним книги с ключом `id`.

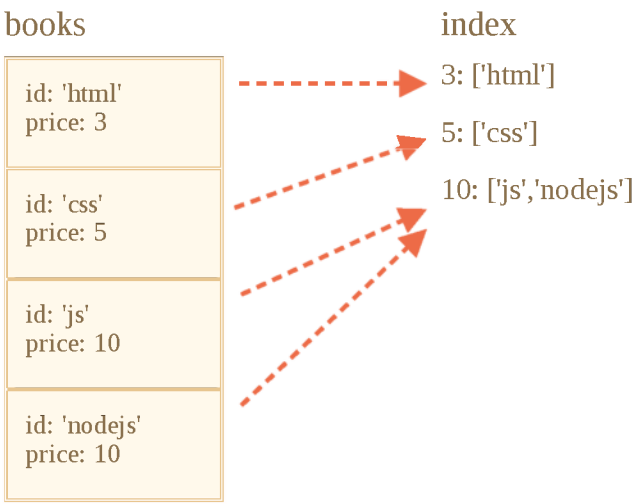
Допустим, мы хотим сделать поиск по полю `price`.

Сначала нам нужно создать индекс. Индексы должны создаваться в `upgradeneeded`, как и хранилище объектов:

```
openRequest.onupgradeneeded = function() {
  // мы должны создать индекс здесь, в versionchange транзакции
  let books = db.createObjectStore('books', {keyPath: 'id'});
  let index = inventory.createIndex('price_idx', 'price');
};
```

- Индекс будет отслеживать поле `price`.
- Поле price не уникальное, у нас может быть несколько книг с одинаковой ценой, поэтому мы не устанавливаем опцию `unique`.
- Поле price не является массивом, поэтому флаг `multiEntry` не применим.

Представим, что в нашем `inventory` есть 4 книги. Вот картинка, которая показывает, что такое «индекс».



Как уже говорилось, индекс для каждого значения `price` (второй аргумент) хранит список ключей, имеющих эту цену.

Индексы автоматически обновляются, нам не нужно об этом заботиться.

Сейчас, когда мы хотим найти объект по цене, мы просто применяем те же методы поиска к индексу:

```
let transaction = db.transaction("books"); // readonly
let books = transaction.objectStore("books");
let priceIndex = books.index("price_idx");

let request = priceIndex.getAll(10);

request.onsuccess = function() {
  if (request.result !== undefined) {
    console.log("Книги", request.result); // массив книг с ценой 10
  } else {
    console.log("Нет таких книг");
  }
};
```

Мы также можем использовать `IDBKeyRange`, чтобы создать диапазон и найти дешёвые/дорогие книги:

```
// найдём книги, где цена < 5
let request = priceIndex.getAll(IDBKeyRange.upperBound(5));
```

Индексы внутренне отсортированы по полю отслеживаемого объекта, в нашем случае по `price`. Поэтому результат поиска будет уже отсортированный по полю `price`.

Удаление из хранилища

Метод `delete` удаляет значения по запросу, формат вызова такой же как в `getAll`:

- **`delete(query)`** – производит удаление соответствующих запросу значений.

Например:

```
// удалить книгу с id='js'
books.delete('js');
```

Если нам нужно удалить книги, основываясь на цене или на любом другом поле, сначала нам надо найти ключ в индексе, а затем выполнить `delete`:

```
// найдём ключ, где цена = 5
let request = priceIndex.getKey(5);

request.onsuccess = function() {
  let id = request.result;
  let deleteRequest = books.delete(id);
};
```

Чтобы удалить всё:

```
books.clear(); // очищаем хранилище.
```

Курсоры

Такие методы как `getAll/getAllKeys` возвращают массив ключей/значений.

Но хранилище объектов может быть огромным, больше, чем доступно памяти.

Тогда метод `getAll` вернёт ошибку при попытке получить все записи в массиве.

Что делать?

Курсоры предоставляют возможности для работы в таких ситуациях.

Объект *cursor* идёт по хранилищу объектов с заданным запросом (query) и возвращает пары ключ/значение по очереди, а не все сразу. Это позволяет экономить память.

Так как хранилище объектов внутренне отсортировано по ключу, курсор проходит по хранилищу в порядке хранения ключей (по возрастанию по умолчанию).

Синтаксис:

```
// как getAll, но с использованием курсора:
let request = store.openCursor(query, [direction]);

// чтобы получить ключи, не значения (как getAllKeys): store.openKeyCursor
```

- **`query`** ключ или диапазон ключей, как для `getAll`.
- **`direction`** необязательный аргумент, доступные значения:

- `"next"` – по умолчанию, курсор будет проходить от самого маленького ключа к большему.
- `"prev"` – обратный порядок: от самого большого ключа к меньшему.
- `"nextunique"`, `"prevunique"` – то же самое, но курсор пропускает записи с тем же ключом, что уже был (только для курсоров по индексам, например, для нескольких книг с price=5, будет возвращена только первая).

Основным отличием курсора является то, что `request.onsuccess` генерируется многократно: один раз для каждого результата.

Вот пример того, как использовать курсор:

```
let transaction = db.transaction("books");
let books = transaction.objectStore("books");

let request = books.openCursor();

// вызывается для каждой найденной курсором книги
request.onsuccess = function() {
  let cursor = request.result;
  if (cursor) {
    let key = cursor.key; // ключ книги (поле id)
    let value = cursor.value; // объект книги
    console.log(key, value);
    cursor.continue();
  } else {
    console.log("Книг больше нет");
  }
};
```

Основные методы курсора:

- `advance(count)` – продвинуть курсор на `count` позиций, пропустив значения.
- `continue([key])` – продвинуть курсор к следующему значению в диапазоне соответсвия (или до позиции сразу после ключа key, если указан).

Независимо от того, есть ли ещё значения, соответствующие курсору или нет – вызывается `onsuccess`, затем в `result` мы можем получить курсор, указывающий на следующую запись или равный `undefined`.

В приведённом выше примере курсор был создан для хранилища объектов.

Но мы также можем создать курсор для индексов. Как мы помним, индексы позволяют искать по полю объекта. Курсоры для индексов работают так же, как

для хранилищ объектов – они позволяют экономить память, возвращая одно значение в единицу времени.

Для курсоров по индексам `cursor.key` является ключом индекса (например price), нам следует использовать свойство `cursor.primaryKey` как ключ объекта:

```
let request = priceIdx.openCursor(IDBKeyRange.upperBound(5));

// вызывается для каждой записи
request.onsuccess = function() {
  let cursor = request.result;
  if (cursor) {
    let key = cursor.primaryKey; // следующий ключ в хранилище объектов (поле id)
    let value = cursor.value; // следующее значение в хранилище объектов (объект "к
    let key = cursor.key; // следующий ключ индекса (price)
    console.log(key, value);
    cursor.continue();
  } else {
    console.log("Книг больше нет");
  }
};
```

Обёртка для промисов

Добавлять к каждому запросу `onsuccess/onerror` немного громоздко. Мы можем сделать нашу жизнь проще, используя делегирование событий, например, установить обработчики на все транзакции, но использовать `async/await` намного удобнее.

Давайте далее в главе использовать небольшую обёртку над промисами https://github.com/jakearchibald/idb . Она создаёт глобальный `idb` объект с промисифицированными IndexedDB методами.

Тогда вместо `onsuccess/onerror` мы можем писать примерно так:

```
let db = await idb.openDb('store', 1, db => {
  if (db.oldVersion == 0) {
    // выполняем инициализацию
    db.createObjectStore('books', {keyPath: 'id'});
  }
});

let transaction = db.transaction('books', 'readwrite');
let books = transaction.objectStore('books');

try {
  await books.add(...);
```

```
  await books.add(...);

  await transaction.complete;

  console.log('сохранено');
} catch(err) {
  console.log('ошибка', err.message);
}
```

Теперь у нас красивый «плоский асинхронный» код и, конечно, будет работать `try..catch`.

Обработка ошибок

Если мы не перехватим ошибку, то она «вывалится» наружу, вверх по стеку вызовов, до ближайшего внешнего `try..catch`.

Необработанная ошибка становится событием «unhandled promise rejection» в объекте `window`.

Мы можем обработать такие ошибки вот так:

```
window.addEventListener('unhandledrejection', event => {
  let request = event.target; // объект запроса IndexedDB
  let error = event.reason; //  Необработанный объект ошибки, как request.error
  ...сообщить об ошибке...
});
```

Подводный камень: «Inactive transaction»

Как мы уже знаем, транзакции автоматически завершаются, как только браузер завершает работу с текущим кодом и макрозадачу. Поэтому, если мы поместим *макрозадачу* наподобие `fetch` в середину транзакции, транзакция не будет ожидать её завершения. Произойдёт автозавершение транзакции. Поэтому при следующем запросе возникнет ошибка.

Для промисифицирующей обёртки и `async/await` поведение такое же.

Вот пример `fetch` в середине транзакции:

```
let transaction = db.transaction("inventory", "readwrite");
let inventory = transaction.objectStore("inventory");

await inventory.add({ id: 'js', price: 10, created: new Date() });

await fetch(...); // (*)

await inventory.add({ id: 'js', price: 10, created: new Date() }); // Ошибка
```

Следующий `inventory.add` после `fetch (*)` не сработает, сгенерируется ошибка «inactive transaction», потому что транзакция уже завершена и закрыта к этому времени.

Решение такое же, как при работе с обычным IndexedDB: либо создать новую транзакцию, либо разделить задачу на части.

1. Подготовить данные и получить всё, что необходимо.
2. Затем сохранить в базу данных.

Получение встроенных объектов

Внутренне обёртка выполняет встроенные IndexedDB запросы, добавляя к ним `onerror/onsuccess`, и возвращает промисы, которые отклоняются или выполняются с переданным результатом.

Это работает в большинстве случаев. Примеры можно увидеть на странице библиотеки https://github.com/jakearchibald/idb .

В некоторых редких случаях, когда нам нужен оригинальный объект `request`, мы можем получить в нему доступ, используя свойство `promise.request`:

```
let promise = books.add(book); // получаем промис (без await, не ждём результата)

let request = promise.request; // встроенный объект запроса
let transaction = request.transaction; // встроенный объект транзакции

// ...работаем с IndexedDB...

let result = await promise; // если ещё нужно
```

Итого

IndexedDB можно рассматривать как «localStorage на стероидах». Это простая база данных типа ключ-значение, достаточно мощная для оффлайн приложений, но простая в использовании.

Лучшим руководством является спецификация, текущая версия 2.0 , но также поддерживаются несколько методов из 3.0 (не так много отличий) версии.

Использование можно описать в нескольких фразах:

1. Подключить обёртку над промисами, например idb .
2. Открыть базу данных: `idb.openDb(name, version, onupgradeneeded)`
   - Создание хранилищ объектов и индексов происходит в обработчике `onupgradeneeded`.

- Обновление версии – либо сравнивая номера версий, либо можно проверить что существует, а что нет.

3. Для запросов:
   - Создать транзакцию `db.transaction('books')` (можно указать readwrite, если надо).
   - Получить хранилище объектов `transaction.objectStore('books')`.
4. Затем для поиска по ключу вызываем методы непосредственно у хранилища объектов.
   - Для поиска по любому полю объекта создайте индекс.
5. Если данные не помещаются в памяти, то используйте курсор.

Демо-приложение:

https://plnkr.co/edit/u6nmQ2jTFAH2KDySbo4R?p=preview

Анимация

Анимации на CSS и JavaScript.

Кривые Безье

Кривые Безье используются в компьютерной графике для рисования плавных изгибов, в CSS-анимации и много где ещё.

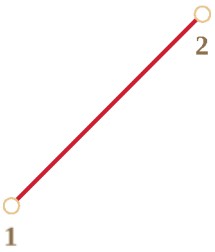
Это очень простая вещь, которую стоит изучить один раз, а затем чувствовать себя комфортно в мире векторной графики и продвинутых анимаций.

Опорные точки

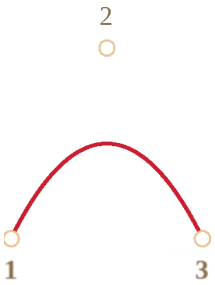
Кривая Безье задаётся опорными точками.

Их может быть две, три, четыре или больше. Например:

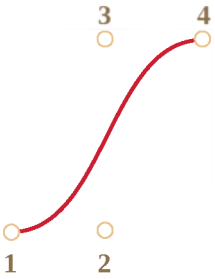
По двум точкам:



По трём точкам:

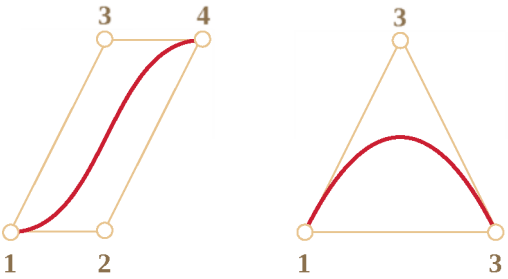


По четырём точкам:



Если вы посмотрите внимательно на эти кривые, то «на глазок» заметите:

1. **Точки не всегда на кривой.** Это совершенно нормально, как именно строится кривая мы рассмотрим чуть позже.
2. **Степень кривой равна числу точек минус один.** Для двух точек – это линейная кривая (т.е. прямая), для трёх точек – квадратическая кривая (парабола), для четырёх – кубическая.
3. **Кривая всегда находится внутри выпуклой оболочки , образованной опорными точками:**

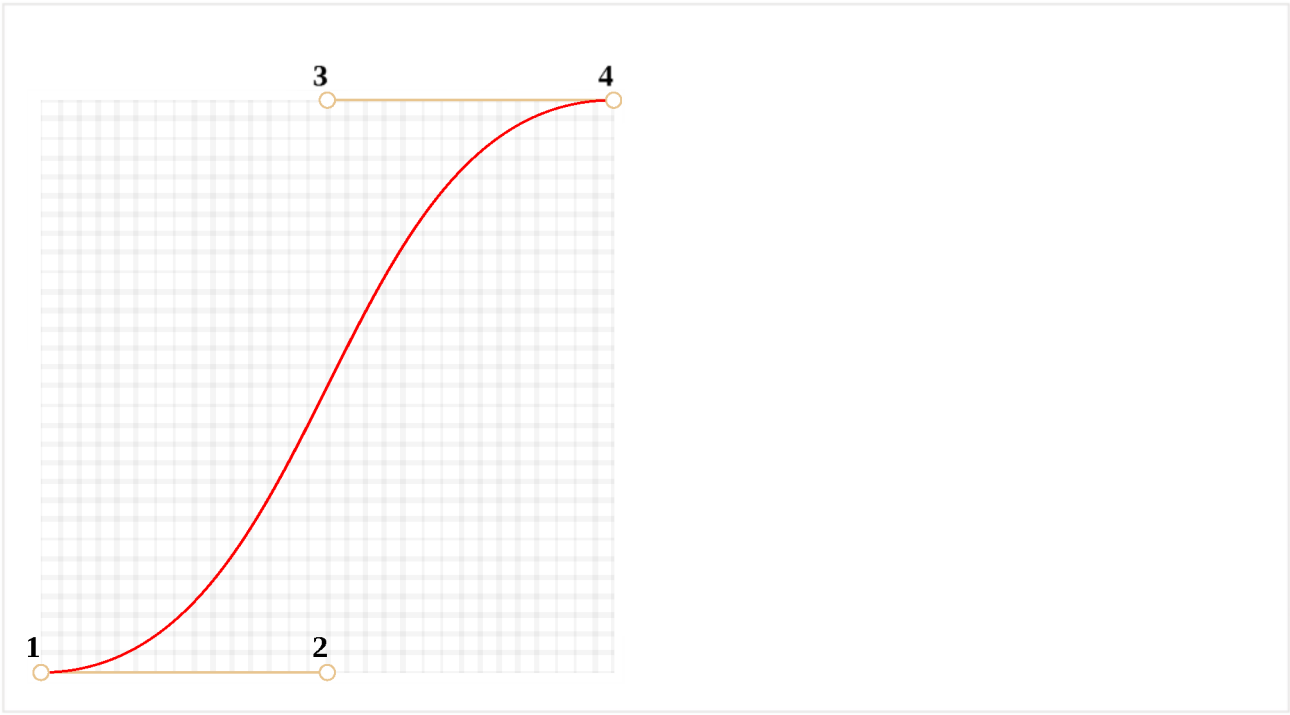


Благодаря последнему свойству в компьютерной графике можно оптимизировать проверку пересечения двух кривых. Если их выпуклые оболочки не пересекаются, то и кривые тоже не пересекутся. Таким образом, проверка пересечения выпуклых оболочек в первую очередь может дать быстрый ответ на вопрос о наличии пересечения. Проверить пересечение или выпуклые оболочки гораздо проще, потому что это прямоугольники,

треугольники и т.д. (см. рисунок выше), гораздо более простые фигуры, чем кривая.

Основная ценность кривых Безье для рисования в том, что, двигая точки, кривую можно менять, причём кривая при этом меняется интуитивно понятным образом.

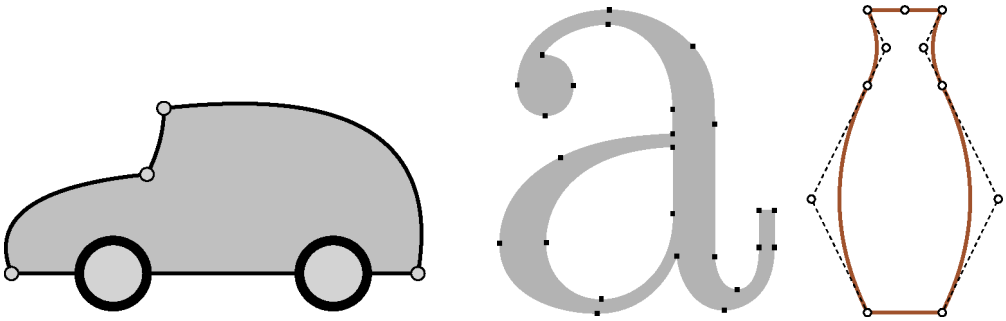
Попробуйте двигать точки мышью в примере ниже:



Как можно заметить, кривая натянута по касательным 1 → 2 и 3 → 4.

После небольшой практики становится понятно, как расположить точки, чтобы получить нужную форму. А, соединяя несколько кривых, можно получить практически что угодно.

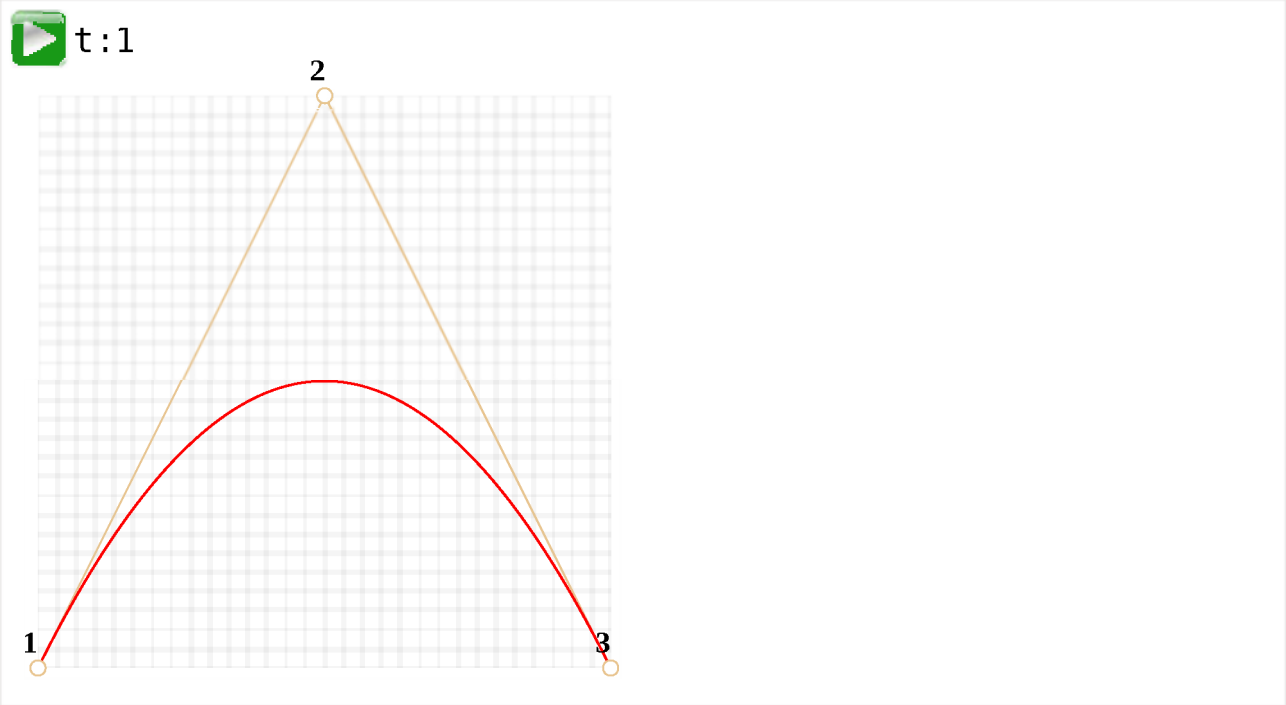
Вот некоторые примеры:



Алгоритм «де Кастельжо»

Есть математическая формула для кривых Безье, но давайте рассмотрим её чуть позже, потому что Алгоритм де Кастельжо идентичен математическому определению кривой и наглядно показывает, как она строится.

Рассмотрим его на примере трёх точек (точки 1,2 и 3 можно двигать). Нажатие на кнопку «play» запустит демонстрацию.



Построение кривой Безье с 3 точками по «алгоритму де Кастельжо»:

1. Рисуются опорные точки. В примере это: `1`, `2`, `3`.
2. Строятся отрезки между опорными точками в следующем порядке 1 → 2 → 3. На рисунке они коричневые.
3. Параметр `t` «пробегает» значения от `0` до `1`. В примере использован шаг `0.05`, т.е. в цикле `0, 0.05, 0.1, 0.15, ... 0.95, 1`.

   Для каждого из этих значений `t`:

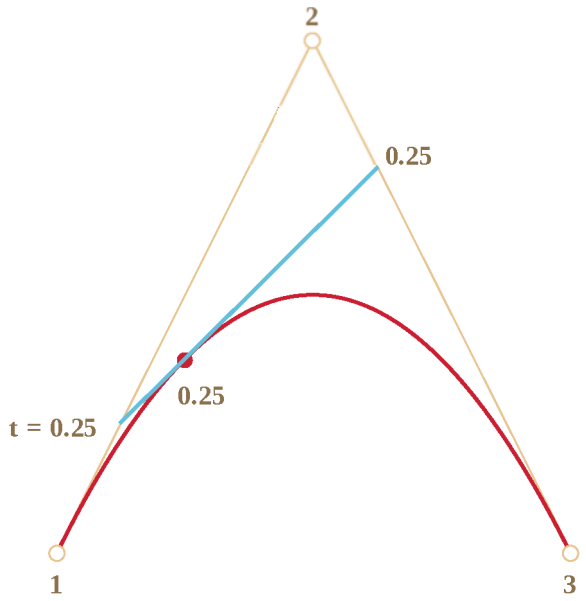
   - На каждом из коричневых отрезков берётся точка, находящаяся на расстоянии, пропорциональном `t`, от его начала. Так как отрезков два, то и точек две.

     Например, при `t=0` – точки будут в начале, при `t=0.25` – на расстоянии в 25% от начала отрезка, при `t=0.5` – 50% (на середине), при `t=1` – в конце отрезков.

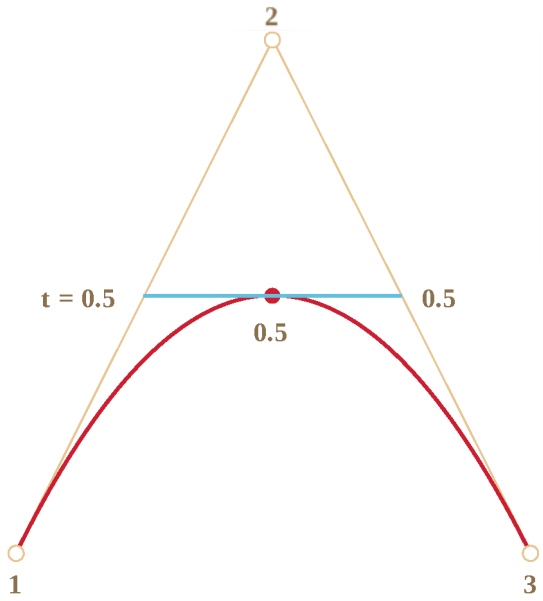
   - Эти точки соединяются. На рисунке ниже соединяющий их отрезок изображён синим.

При `t=0.25` **При `t=0.5`**

При `t=0.25`



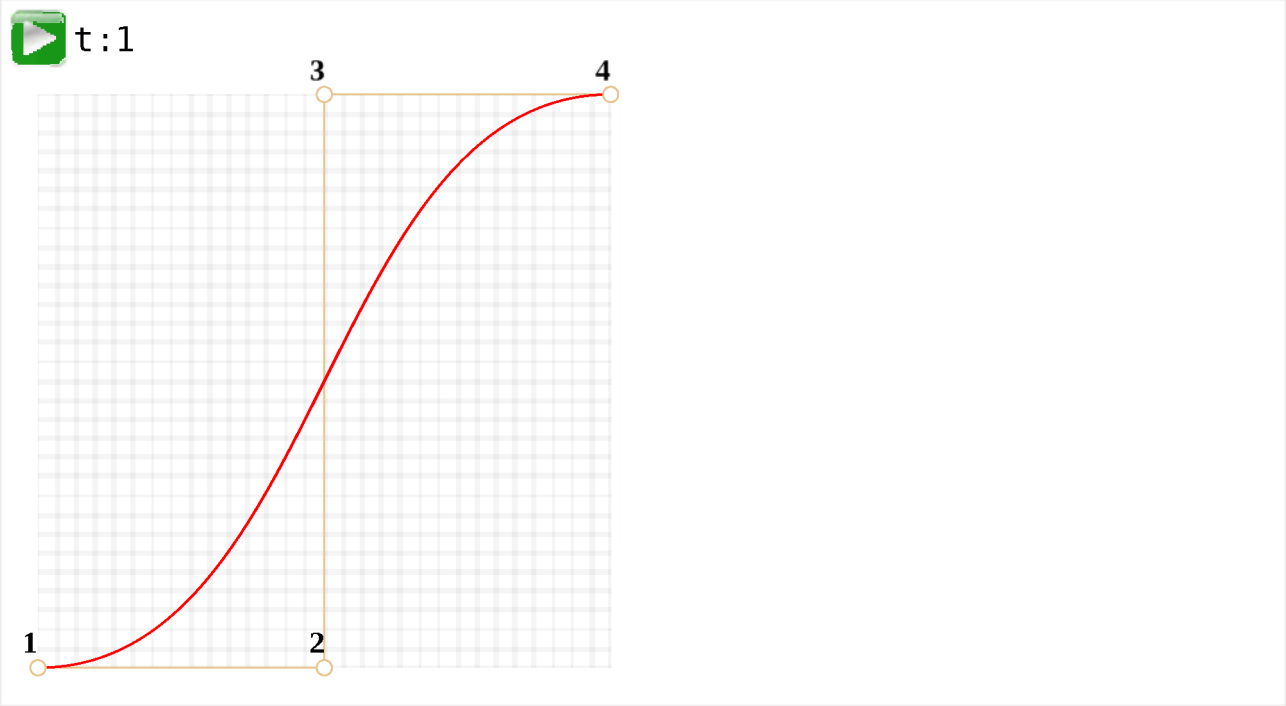
При `t=0.5`



4. На получившемся синем отрезке берётся точка на расстоянии, соответствующем `t`. То есть, для `t=0.25` (левый рисунок) получаем точку в конце первой четверти отрезка, для `t=0.5` (правый рисунок) – в середине отрезка. На рисунках выше эта точка отмечена красным.
5. По мере того, как `t` «пробегает» последовательность от `0` до `1`, каждое значение `t` добавляет к кривой точку. Совокупность таких точек для всех значений образует кривую Безье. Она красная и имеет параболическую форму на картинках выше.

Был описан процесс для построения по трём точкам. Но то же самое происходит и с четырьмя точками.

Демо для четырёх точек (точки можно двигать):



Алгоритм для 4 точек:

- Точки по порядку соединяются отрезками: 1 → 2, 2 → 3, 3 → 4. Получается три коричневых отрезка.
- Для `t` на отрезке от `0` до `1`:
  - На отрезках берутся точки, соответствующие текущему `t`, соединяются. Получается два зелёных отрезка .
  - На этих отрезках берутся точки, соответствующие текущему `t`, соединяются. Получается один синий отрезок.
  - На синем отрезке берётся точка, соответствующая текущему `t`. При запуске примера выше она красная.
- Эти точки вместе описывают кривую.

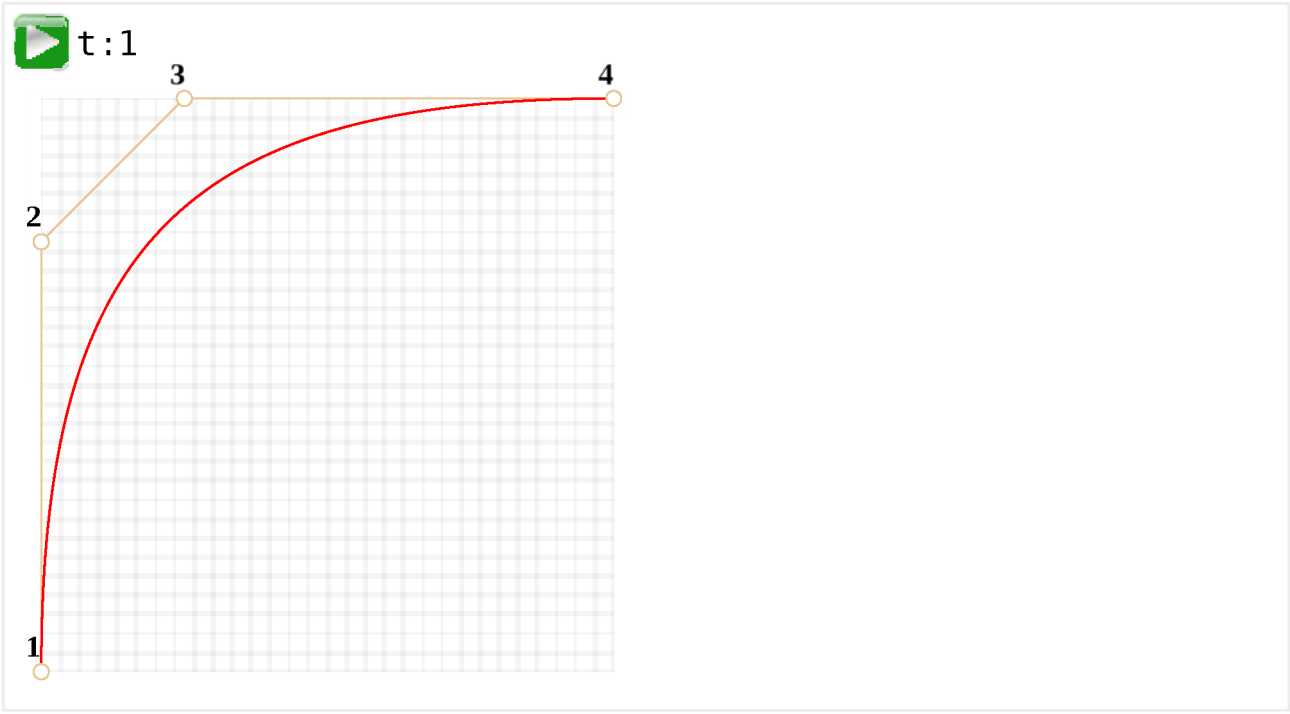
Алгоритм является рекурсивным и может быть обобщён на любое количество контрольных точек.

Дано N контрольных точек:

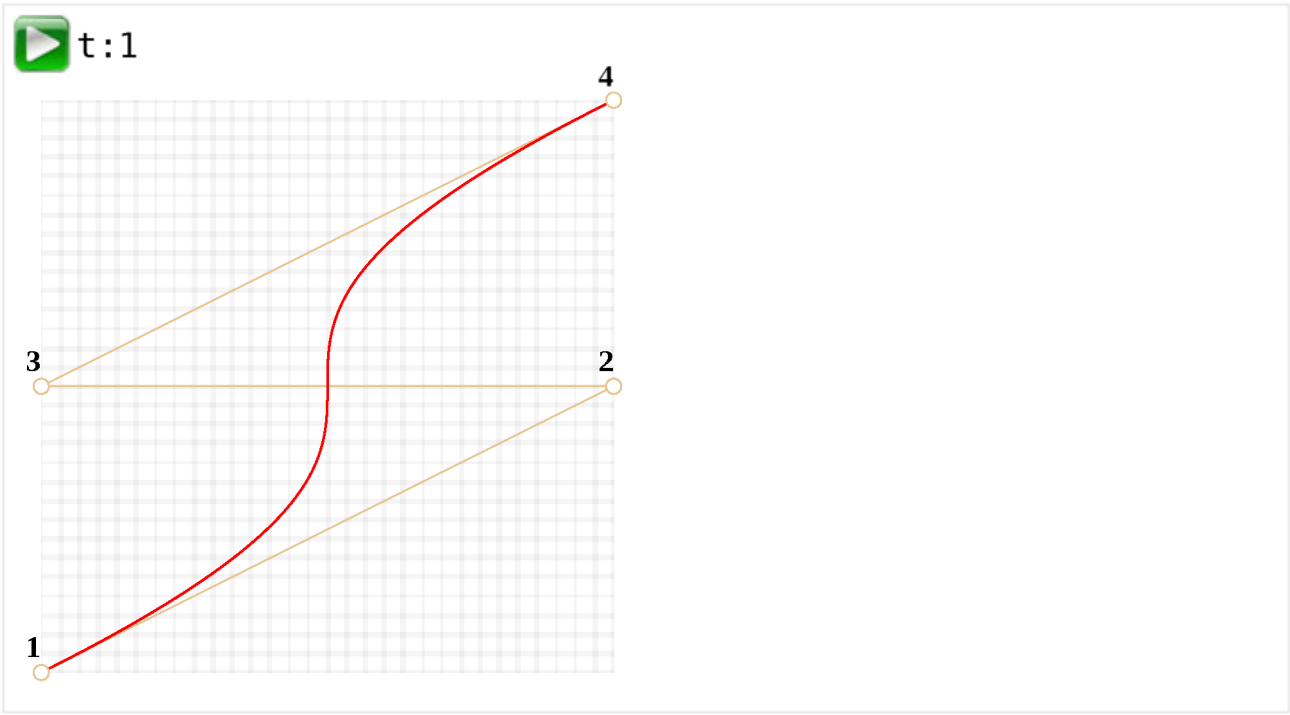
1. Соединяем их, чтобы получить N-1 отрезков.
2. Затем для каждого `t` от `0` до `1` берём точку на каждом отрезке на расстоянии пропорциональном `t` и соединяем их. Там будет N-2 отрезков.
3. Повторяем 2 шаг, пока не останется одна точка.

Эти точки образуют кривую.

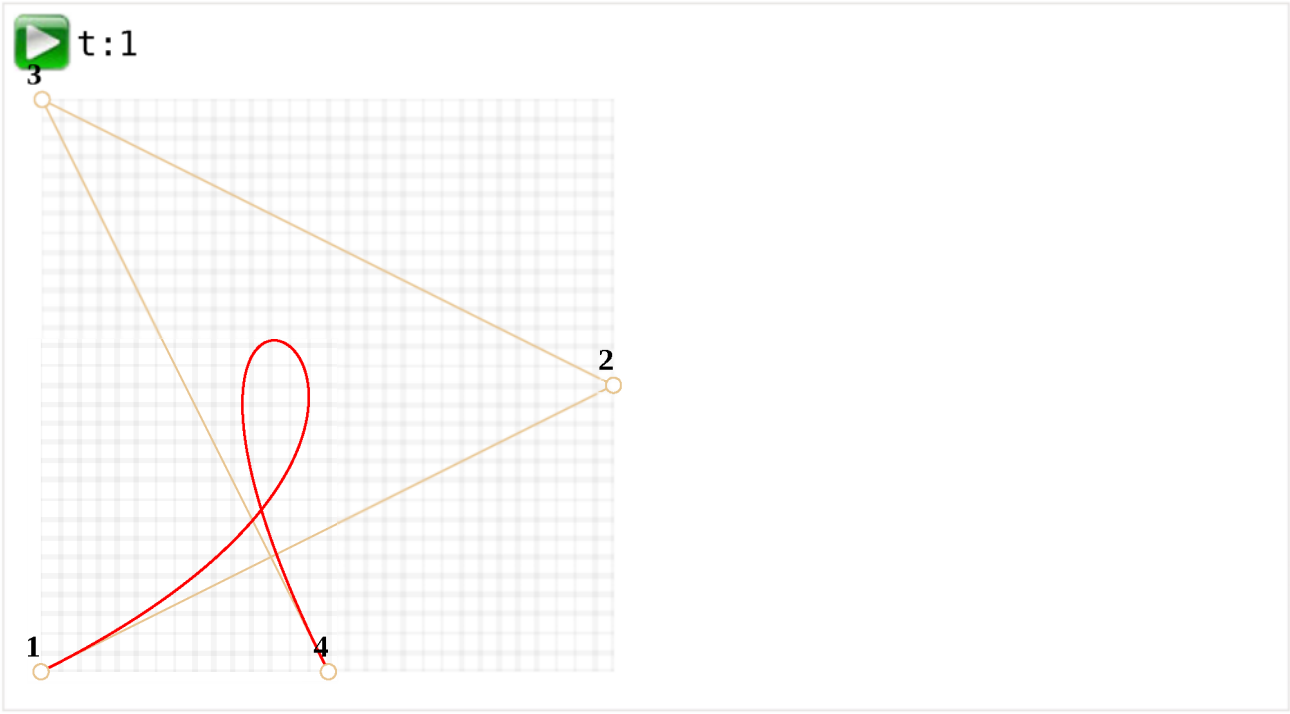
Кривая, которая выглядит как `y=1/t`:



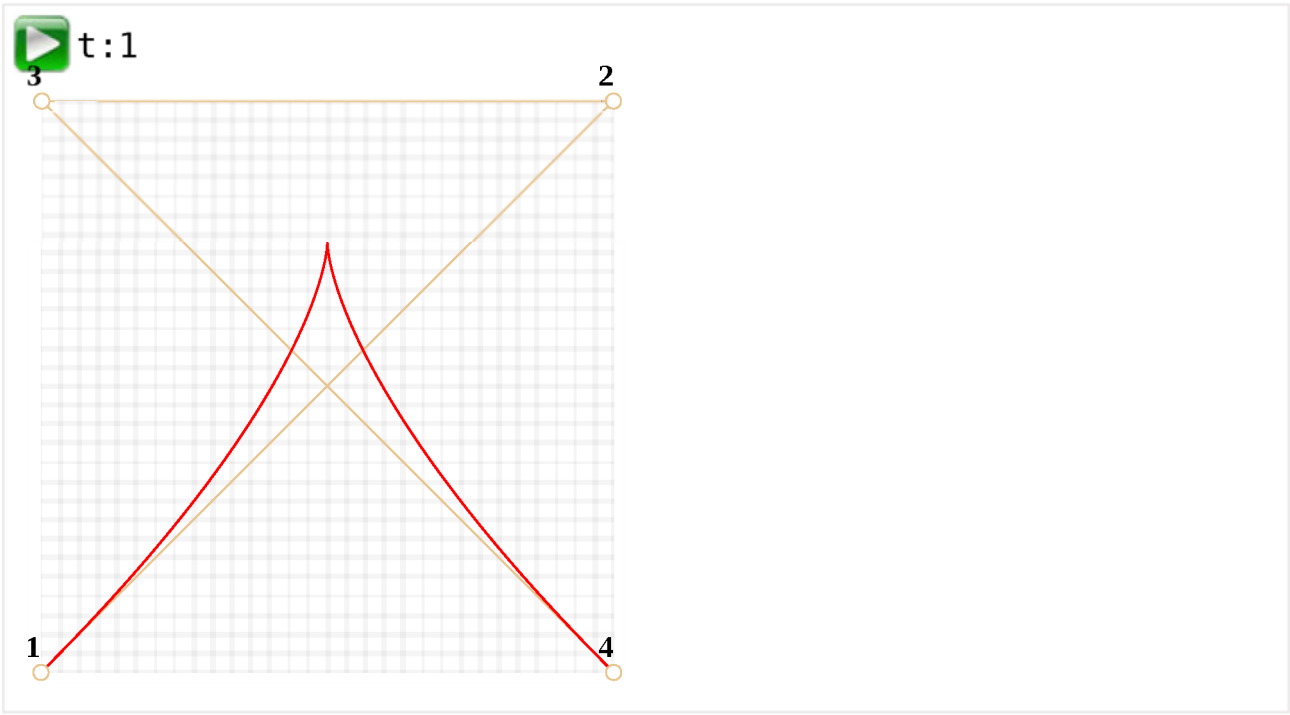
Зигзагообразные контрольные точки тоже работают нормально:



Создание петли возможно:



Негладкая кривая Безье (да, это тоже возможно):



Поскольку алгоритм является рекурсивным, мы можем построить кривые Безье любого порядка, используя 5, 6 или более контрольных точек. Но на практике много точек не так полезны. Обычно мы берём 2-3 точки, а для сложных линий склеиваем несколько кривых. Это проще для разработки и расчёта.

Как нарисовать кривую *через* заданные точки?

Для задания кривой Безье используются контрольные точки. Как видим, они не находятся на кривой, кроме первой и последней.

Иногда перед нами стоит другая задача: нарисовать кривую *через* несколько точек, чтобы все они были на одной гладкой кривой. Эта задача называется интерполяцией , и она за рамками нашего изложения.

Для таких кривых существуют математические формулы, например, полином Лагранжа . В компьютерной графике сплайн-интерполяция часто используется для построения плавных кривых, соединяющих множество точек.

Математика

Кривая Безье может быть описана с помощью математической формулы.

Как мы видели, на самом деле нет необходимости её знать, большинство людей просто рисуют кривую, перемещая точки с помощью мыши. Но если вы увлекаетесь математикой – вот она.

Координаты кривой с контрольными точками P_i: первая контрольная точка имеет координаты $P_1 = (x_1, y_1)$, вторая: $P_2 = (x_2, y_2)$ и т.д., описываются уравнением, зависящим от параметра t на отрезке $[0,1]$.

- Формула для 2-х точечной кривой:

  $$P = (1-t)P_1 + tP_2$$

- Для 3 контрольных точек:

  $$P = (1-t)^2P_1 + 2(1-t)tP_2 + t^2P_3$$

- Для 4 контрольных точек:

  $$P = (1-t)^3P_1 + 3(1-t)^2tP_2 +3(1-t)t^2P_3 + t^3P_4$$

Это векторные уравнения. Другими словами, мы можем поставить x и y вместо P, чтобы получить соответствующие координаты.

Например, 3-точечная кривая образована точками (x,y), рассчитанными как:

- $x = (1-t)^2x_1 + 2(1-t)tx_2 + t^2x_3$
- $y = (1-t)^2y_1 + 2(1-t)ty_2 + t^2y_3$

Вместо $x_1, y_1, x_2, y_2, x_3, y_3$ мы должны поместить координаты 3 контрольных точек, а затем при перемещении `t` от `0` до `1` для каждого значения `t` мы получим `(x,y)` кривой.

Например, если контрольными точками являются `(0,0)`, `(0.5,1)` и `(1,0)`, уравнения становятся:

- $x = (1-t)^2 * 0 + 2(1-t)t * 0.5 + t^2 * 1 = (1-t)t + t^2 = t$
- $y = (1-t)^2 * 0 + 2(1-t)t * 1 + t^2 * 0 = 2(1-t)t = -t^2 + 2t$

Теперь, в то время как `t` «пробегает» от `0` до `1`, набор значений `(x, y)` для каждого `t` образует кривую для таких контрольных точек.

Итого

Кривые Безье задаются опорными точками.

Мы рассмотрели два определения кривых:

1. Через математическую формулу.
2. Использование процесса рисования: алгоритм де Кастельжо.

Их удобство в том, что:

- Можно рисовать плавные линии с помощью мыши, перемещая контрольные точки.
- Сложные формы могут быть сделаны из нескольких кривых Безье.

Применение:

- В компьютерной графике, моделировании, в графических редакторах. Шрифты описываются с помощью кривых Безье.
- В веб-разработке – для графики на Canvas или в формате SVG. Кстати, все живые примеры выше написаны на SVG. Фактически, это один SVG-документ, к которому точки передаются параметрами. Вы можете открыть его в отдельном окне и посмотреть исходник: demo.svg.
- В CSS-анимации для задания траектории или скорости передвижения.

CSS-анимации

CSS позволяет создавать простые анимации без использования JavaScript.

JavaScript может быть использован для управления такими CSS-анимациями. Это позволяет делать более сложные анимации, используя небольшие кусочки кода.

CSS-переходы

Идея CSS-переходов проста: мы указываем, что некоторое свойство должно быть анимировано, и как оно должно быть анимировано. А когда свойство меняется, браузер сам обработает это изменение и отрисует анимацию.

Всё что нам нужно, чтобы начать анимацию – это изменить свойство, а дальше браузер сделает плавный переход сам.

Например, CSS-код ниже анимирует трёх-секундное изменение `background-color`:

```
.animated {
  transition-property: background-color;
  transition-duration: 3s;
}
```

Теперь, если элементу присвоен класс `.animated`, любое изменение свойства `background-color` будет анимироваться в течение трёх секунд.

Нажмите кнопку ниже, чтобы анимировать фон:

```
<button id="color">Нажми меня</button>

<style>
  #color {
    transition-property: background-color;
    transition-duration: 3s;
  }
</style>

<script>
  color.onclick = function() {
    this.style.backgroundColor = 'red';
  };
</script>
```

Нажми меня

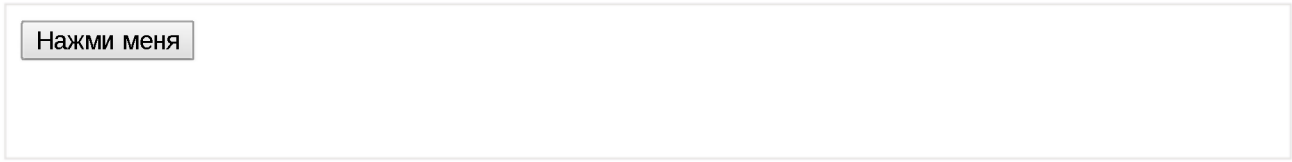
Существует 4 свойства для описания CSS-переходов:

- `transition-property` – свойство перехода
- `transition-duration` – продолжительность перехода
- `transition-timing-function` – временная функция перехода
- `transition-delay` – задержка начала перехода

Далее мы рассмотрим их все, а сейчас ещё заметим, что есть также общее свойство `transition`, которое позволяет задать их одновременно в последовательности: `property duration timing-function delay`, а также анимировать несколько свойств одновременно.

Например, у этой кнопки анимируются два свойства `color` и `font-size` одновременно:

```
<button id="growing">Нажми меня</button>

<style>
#growing {
  transition: font-size 3s, color 2s;
}
</style>

<script>
growing.onclick = function() {
  this.style.fontSize = '36px';
  this.style.color = 'red';
};
</script>
```



Теперь рассмотрим каждое свойство анимации по отдельности.

transition-property

В `transition-property` записывается список свойств, изменения которых необходимо анимировать, например: `left`, `margin-left`, `height`, `color`.

Анимировать можно не все свойства, но многие из них. Значение свойства `all` означает «анимируй все свойства».

transition-duration

В `transition-duration` можно определить, сколько времени займёт анимация. Время должно быть задано в формате времени CSS: в секундах `s` или миллисекундах `ms`.

transition-delay

В `transition-delay` можно определить задержку *перед* началом анимации. Например, если `transition-delay: 1s`, тогда анимация начнётся через 1 секунду после изменения свойства.

Отрицательные значения также допустимы. В таком случае анимация начнётся с середины. Например, если `transition-duration` равно `2s`, а `transition-delay` – `-1s`, тогда анимация займёт одну секунду и начнётся с середины.

Здесь приведён пример анимации, сдвигающей цифры от `0` до `9` с использованием CSS-свойства `transform` со значением `translate`:

https://plnkr.co/edit/nAOI7V3zf5BKhM6oe931?p=preview

Свойство `transform` анимируется следующим образом:

```
#stripe.animate {
  transform: translate(-90%);
  transition-property: transform;
  transition-duration: 9s;
}
```

В примере выше JavaScript-код добавляет класс `.animate` к элементу, после чего начинается анимация:

```
stripe.classList.add('animate');
```

Можно начать анимацию «с середины», с определённого числа, например, используя отрицательное значение `transition-delay`, соответствующие необходимому числу.

Если вы нажмёте на цифру ниже, то анимация начнётся с последней секунды:

https://plnkr.co/edit/Iit6aU7T7uYtc7vRISve?p=preview

JavaScript делает это с помощью нескольких строк кода:

```
stripe.onclick = function() {
  let sec = new Date().getSeconds() % 10;
  // например, значение -3s здесь начнут анимацию с третьей секунды
  stripe.style.transitionDelay = '-' + sec + 's';
  stripe.classList.add('animate');
};
```

transition-timing-function

Временная функция описывает, как процесс анимации будет распределён во времени. Будет ли она начата медленно и затем ускорится или наоборот.

На первый взгляд это очень сложное свойство, но оно становится понятным, если уделить ему немного времени.

Это свойство может принимать два вида значений: кривую Безье или количество шагов. Давайте начнём с кривой Безье, как с наиболее часто используемой.

Кривая Безье

Временная функция может быть задана, как кривая Безье с 4 контрольными точками, удовлетворяющими условиям:

1. Первая контрольная точка: `(0,0)`.
2. Последняя контрольная точка: `(1,1)`.
3. Для промежуточных точек значение `x` должно быть `0..1`, значение `y` может принимать любое значение.

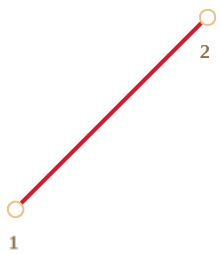
Синтаксис для кривых Безье в CSS: `cubic-bezier(x2, y2, x3, y3)`. Нам необходимо задать только вторую и третью контрольные точки, потому что первая зафиксирована со значением `(0,0)` и четвёртая – `(1,1)`.

Временная функция описывает то, насколько быстро происходит анимации во времени.

- Ось `x` – это время: `0` – начальный момент, `1` – последний момент `transition-duration`.
- Ось `y` указывает на завершение процесса: `0` – начальное значение свойства, `1` – конечное значение.

Самым простым примером анимации является равномерная анимация с линейной скоростью. Она может быть задана с помощью кривой `cubic-bezier(0, 0, 1, 1)`.

Вот как выглядит эта «кривая»:



…Как мы видим, это прямая линия. Значению времени (`x`) соответствует значение завершённости анимации (`y`), которое равномерно изменяется от `0` к `1`.

В примере ниже поезд «едет» слева направо с одинаковой скоростью (нажмите на поезд):

https://plnkr.co/edit/5l1dSNaTDMxtPsZ8wzKs?p=preview

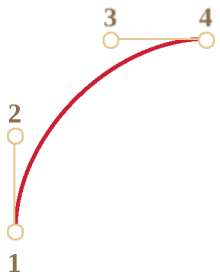
В свойстве `transition` указана следующая кривая Безье:

```
.train {
  left: 0;
  transition: left 5s cubic-bezier(0, 0, 1, 1);
  /* JavaScript устанавливает свойство left равным 450px */
}
```

…А как показать замедляющийся поезд?

Мы можем использовать другую кривую Безье: `cubic-bezier(0.0, 0.5, 0.5 ,1.0)`.

Её график:



Как видим, анимация начинается быстро: кривая быстро поднимается вверх, и затем все медленнее и медленнее.

Ниже временная функция в действии (нажмите на поезд):

https://plnkr.co/edit/sJLWo135qIfHLJ9VJ8VP?p=preview

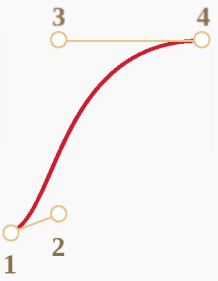
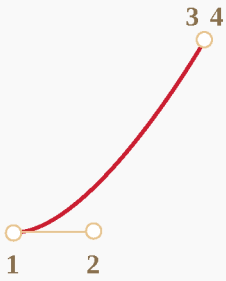
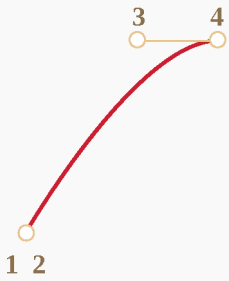
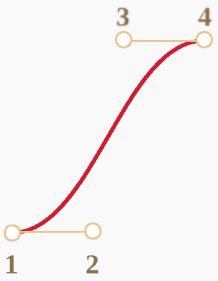
CSS:

```
.train {
  left: 0;
  transition: left 5s cubic-bezier(0, .5, .5, 1);
  /* JavaScript устанавливает свойство left равным 450px */
}
```

Есть несколько встроенных обозначений кривых Безье: `linear`, `ease`, `ease-in`, `ease-out` и `ease-in-out`.

`linear` это короткая запись для `cubic-bezier(0, 0, 1, 1)` – прямой линии, которую мы видели раньше.

Другие названия – это также сокращения для других `cubic-bezier`:

`ease`*	`ease-in`	`ease-out`	`ease-in-out`
`(0.25, 0.1, 0.25, 1.0)`	`(0.42, 0, 1.0, 1.0)`	`(0, 0, 0.58, 1.0)`	`(0.42, 0, 0.58, 1.0)`
			

`*` – используется по умолчанию, если не задана другая временная функция.

Для того, чтобы замедлить поезд, мы можем использовать `ease-out`:

```
.train {
  left: 0;
  transition: left 5s ease-out;
  /* transition: left 5s cubic-bezier(0, .5, .5, 1); */
}
```

Но получившийся результат немного отличается.

Кривая Безье может заставить анимацию «выпрыгивать» за пределы диапазона.

Контрольные точки могут иметь любые значения по оси `y`: отрицательные или сколь угодно большие. В таком случае кривая Безье будет скакать очень высоко или очень низко, заставляя анимацию выходить за её нормальные пределы.

В приведённом ниже примере код анимации:

```
.train {
  left: 100px;
  transition: left 5s cubic-bezier(.5, -1, .5, 2);
```

```
  /* JavaScript sets left to 400px */
}
```

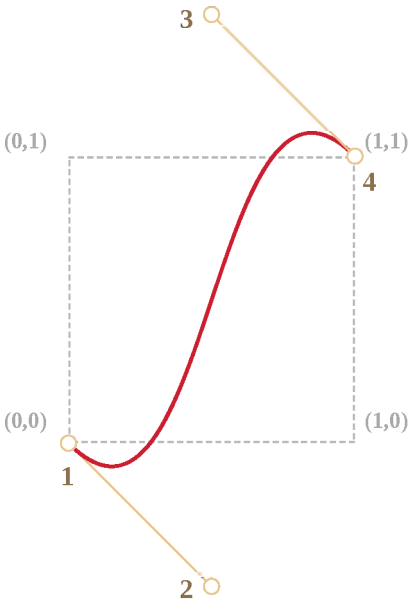
Свойство `left` будет анимироваться от `100px` до `400px`.

Но когда вы нажмёте на поезд, вы увидите следующее:

- Сначала, поезд поедет *назад*: `left` станет меньше, чем `100px`.
- Затем он поедет вперёд, немного дальше, чем `400px`.
- И затем вернётся назад в значение `400px`.

https://plnkr.co/edit/fJY48J8jEN9jULiZIBPk?p=preview

Если мы взглянем на кривую Безье из примера, становится понятно поведение поезда.



Мы вынесли координату `y` для первой опорной точки ниже нуля и выше единицы для третьей опорной точки, поэтому кривая вышла за пределы «обычного» квадрата. Значения `y` вышли из «стандартного» диапазона `0..1`.

Как мы знаем, ось `y` измеряет «завершённость процесса анимации». Значение `y = 0` соответствует начальному значению анимируемого свойства и `y = 1` – конечному значению. Таким образом, `y<0` делает значение свойства `left` меньше начального значения и `y>1` – больше конечного.

Это, конечно, «мягкий» вариант. Если значение `y` будут `-99` и `99`, то поезд будет гораздо сильнее «выпрыгивать» за пределы.

Как сделать кривую Безье необходимую для конкретной задачи? Существует множество инструментов, например можно использовать с сайта http://cubic-bezier.com/ .

Шаги

Временная функция `steps(количество шагов[, start/end])` позволяет разделить анимацию на шаги.

Давайте рассмотрим это на уже знакомом нам примере с цифрами.

Ниже представлен список цифр, без какой-либо анимации, который мы будем использовать в качестве основы:

https://plnkr.co/edit/l1IemRUr6NF7lP1qa64v?p=preview

Давайте сделаем так, чтобы цифры двигались не плавно, а появлялись одна за другой раздельно. Для этого скроем все что находится за красным «окошком» и будем сдвигать список влево по шагам.

Всего будет 9 шагов, один шаг для каждой цифры:

```
#stripe.animate  {
  transform: translate(-90%);
  transition: transform 9s steps(9, start);
}
```

В действии:

https://plnkr.co/edit/soAoU4xTFJxTzxe7hhpA?p=preview

Первый аргумент временной функции `steps(9, start)` – количество шагов. Трансформация будет разделена на 9 частей (10% каждая). Временной интервал также будет разделён на 9 частей, таким образом свойство `transition: 9s` обеспечивает нам 9 секунду анимации, что даёт по одной секунде на цифру.

Вторым аргументом является одно из ключевых слов: `start` или `end`.

`start` – означает, что в начале анимации нам необходимо перейти на первый шаг немедленно.

Мы можем наблюдать это во время анимации: когда пользователь нажимает на цифру, значение меняется на `1` (первый шаг) сразу и в следующий раз меняется уже в начале следующей секунды.

Анимация будет происходить так:

- `0s` – `-10%` (первое изменение в начале первой секунды, сразу после нажатия)
- `1s` – `-20%`
- ...
- `8s` – `-80%`

- (на протяжении последней секунды отображается последнее значение).

Альтернативное значение `end` означало бы, что изменения нужно применять не в начале, а в конце каждой секунды.

Анимация будет происходить так:

- `0s` – `0`
- `1s` – `-10%` (первое изменение произойдёт в конце первой секунды)
- `2s` – `-20%`
- ...
- `9s` – `-90%`

Пример `step(9, end)` в действии (обратите внимание на паузу между первым изменением цифр):

https://plnkr.co/edit/bvZ6as2XEHYyml3syaDG?p=preview

Также есть сокращённые значения:

- `step-start` – то же самое, что `steps(1, start)`. Оно означает, что анимация начнётся сразу и произойдёт в один шаг. Таким образом она начнётся и завершится сразу, как будто и нет никакой анимации.
- `step-end` – то же самое, что `steps(1, end)`: выполнит анимацию за один шаг в конце `transition-duration`.

Такие значения используются редко, потому что это не совсем анимация, а точнее будет сказать одношаговые изменения.

Событие transitionend

Когда завершается анимация, срабатывает событие `transitionend`.

Оно широко используется для выполнения действий после завершения анимации, а также для создания последовательности анимаций.

Например, корабль в приведённом ниже примере начинает плавать туда и обратно по клику, каждый раз все дальше и дальше вправо:

Анимация начинается с помощью функции `go`, которая вызывается каждый раз снова, когда переход заканчивается и меняется направление:

```
boat.onclick = function() {
  //...
  let times = 1;

  function go() {
    if (times % 2) {
      // плыть вправо
      boat.classList.remove('back');
      boat.style.marginLeft = 100 * times + 200 + 'px';
    } else {
      // плыть влево
      boat.classList.add('back');
      boat.style.marginLeft = 100 * times - 200 + 'px';
    }

  }

  go();

  boat.addEventListener('transitionend', function() {
    times++;
    go();
  });
};
```

Объект события `transitionend` содержит ряд полезных свойства:

`event.propertyName`

Имя свойства, анимация которого завершилась. Может быть полезным, если мы анимируем несколько свойств.

`event.elapsedTime`

Время (в секундах), которое заняла анимация, без учёта `transition-delay`.

Ключевые кадры

Мы можем объединить несколько простых анимаций вместе, используя CSS-правило `@keyframes`.

Оно определяет «имя» анимации и правила: что, когда и где анимировать. После этого можно использовать свойство `animation`, чтобы назначить анимацию на элемент и определить её дополнительные параметры.

Ниже приведён пример с пояснениями:

```
<div class="progress"></div>

<style>
  @keyframes go-left-right {        /* объявляем имя анимации: "go-left-right" */
    from { left: 0px; }             /* от: left: 0px */
    to { left: calc(100% - 50px); } /* до: left: 100%-50px */
  }

  .progress {
    animation: go-left-right 3s infinite alternate;
    /* применить анимацию "go-left-right" на элементе
       продолжительностью 3 секунды
       количество раз: бесконечно (infinite)
       менять направление анимации каждый раз (alternate)
    */

    position: relative;
    border: 2px solid green;
    width: 50px;
    height: 20px;
    background: lime;
  }
</style>
```

Существует множество статей про `@keyframes`, а также детальная спецификация.

Скорее всего, вам нечасто понадобится `@keyframes`, разве что на вашем сайте все постоянно в движении.

Итого

CSS-анимации позволяют плавно, или не очень, менять одно или несколько свойств.

Они хорошо решают большинство задач по анимации. Также мы можем реализовать анимации через JavaScript, более подробно об этом – в следующей главе.

Ограничения CSS-анимаций в сравнении с JavaScript-анимациями:

Достоинства

- Простые анимации делаются просто.
- Быстрые и не создают нагрузку на CPU.

Недостатки

- JavaScript-анимации более гибкие. В них может присутствовать любая анимационная логика, как например «взорвать» элемент.
- Можно изменять не только свойства. Мы можем создавать новые элементы с помощью JavaScript для анимации.

Большинство анимаций может быть реализовано с использованием CSS, как описано в этой главе. А событие `transitionend` позволяет запускать JavaScript после анимации, поэтому CSS-анимации прекрасно интегрируются с кодом.

Но в следующей главе мы рассмотрим некоторые JavaScript-анимации, которые позволяют решать более сложные задачи.

Задачи

Анимировать самолёт (CSS)

важность: 5

Реализуйте анимацию, как в примере ниже (клик на самолёт):

- При нажатии картинка изменяет размеры с `40x24px` до `400x240px` (увеличивается в 10 раз).
- Время анимации 3 секунды.
- По окончании анимации вывести сообщение: «Анимация закончилась!».
- Если во время анимации будут дополнительные клики по картинке – они не должны ничего «сломать».

Открыть песочницу для задачи.

К решению

Анимировать самолёт с перелётом (CSS)

важность: 5

Модифицируйте решение предыдущей задачи Анимировать самолёт (CSS) , чтобы в процессе анимации изображение выросло больше своего стандартного размера 400x240px («выпрыгнуло»), а затем вернулось к нему.

Должно получиться, как в примере ниже (клик на самолёт):

В качестве исходного кода возьмите решение прошлой задачи.

К решению

Анимированный круг

важность: 5

Напишите функцию `showCircle(cx, cy, radius)`, которая будет рисовать постепенно растущий круг.

- `cx,cy` – координаты центра круга относительно окна браузера,
- `radius` – радиус круга.

Нажмите на кнопку ниже, чтобы увидеть как это должно выглядеть:

showCircle(150, 150, 100)

В исходном коде уже указаны правильные CSS-стили круга, таким образам задача заключается в том, чтобы сделать правильную анимацию.

Открыть песочницу для задачи.

К решению

JavaScript-анимации

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 мы будем 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.

Псевдокод 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() {
  // сколько времени прошло с начала анимации?
  let timePassed = Date.now() - start;

  if (timePassed >= 2000) {
    clearInterval(timer); // закончить анимацию через 2 секунды
    return;
  }

  // отрисовать анимацию на момент timePassed, прошедший с начала анимации
  draw(timePassed);
```

```
}, 20);

// в то время как timePassed идёт от 0 до 2000
// left изменяет значение от 0px до 400px
function draw(timePassed) {
  train.style.left = timePassed / 5 + 'px';
}
```

Для просмотра примера, кликните на него:

https://plnkr.co/edit/rvmnzD6oyiGDWESJdvsZ?p=preview

Использование requestAnimationFrame

Теперь давайте представим, что у нас есть несколько анимаций, работающих одновременно.

Если мы запустим 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 у нас 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 перерисовки лучше сгруппировать вместе, тогда они будут легче для браузера, а значит – не грузить процессор и более плавно выглядеть.

Существует ещё одна вещь, про которую надо помнить: когда CPU перегружен или есть другие причины делать перерисовку реже (например, когда вкладка браузера скрыта), нам не следует делать её каждые `20ms`.

Но как нам узнать об этом в JavaScript? 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 скоро, 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 мы можем 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:

`duration`

Продолжительность анимации. Например, `1000`.

`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;
}
```

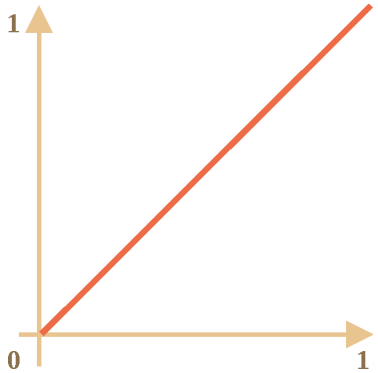


График функции:

Это как если бы в `transition-timing-function` передать значение `linear`. Ниже будут представлены более интересные примеры.

`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 на самом деле и рисует анимацию.

Вот как она могла бы двигать элемент:

```
function draw(progress) {
  train.style.left = progress + 'px';
}
```

…Или делать что-нибудь ещё. Мы можем анимировать что угодно, как захотим.

Теперь давайте используем нашу функцию, чтобы анимировать свойство `width` от `0` до `100%`.

Нажмите на элемент для того, чтобы посмотреть пример:

https://plnkr.co/edit/sj8mFP1F05jGLEsXrcz6?p=preview

Код:

```
animate({
  duration: 1000,
  timing(timeFraction) {
    return timeFraction;
  },
  draw(progress) {
    elem.style.width = progress * 100 + '%';
  }
});
```

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

Мы уже рассмотрели самый простой пример линейной функции расчёта времени выше.

Давайте посмотрим другие. Мы попробуем выполнить анимации с разными функциями расчёта времени, чтобы посмотреть как они работают.

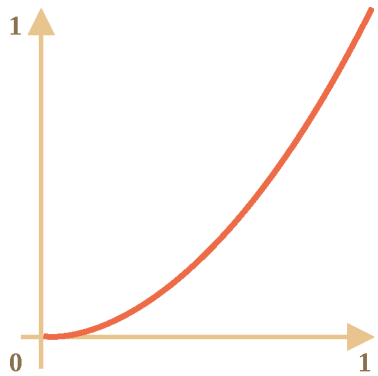
Степень n

Если мы хотим ускорить анимацию, мы можем возвести `progress` в степень `n`.

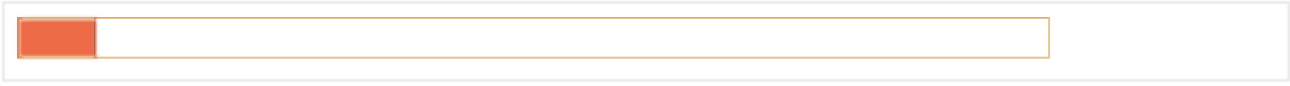
Например, параболическая кривая:

```
function quad(timeFraction) {
  return Math.pow(timeFraction, 2)
}
```

График:

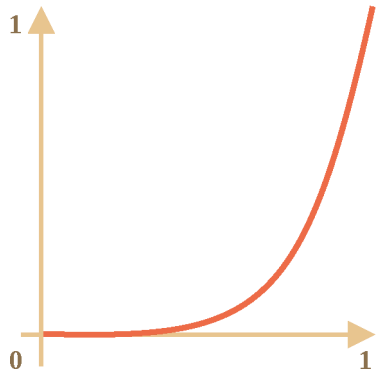


Посмотрим в действии (нажмите для активации):

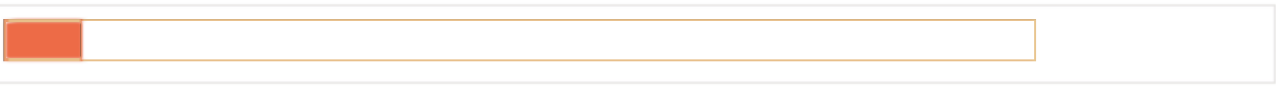


…Или кубическая кривая, или любой другой множитель `n`. Повышение степени увеличивает скорость анимации.

Вот график для функции `progress` в степени `5`:



В действии:

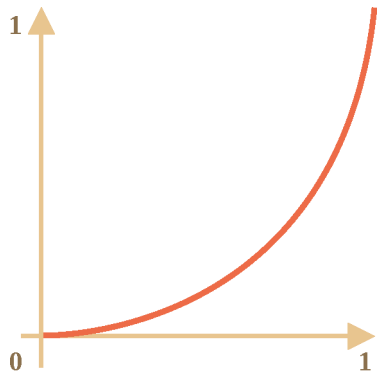


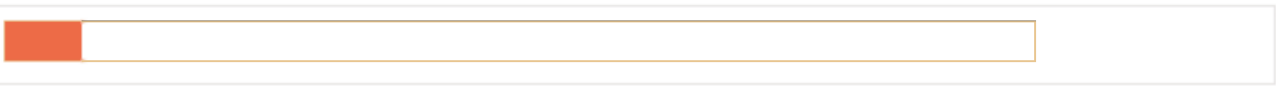
Дуга

Функция:

```
function circ(timeFraction) {
  return 1 - Math.sin(Math.acos(timeFraction));
}
```

График:





Обратно: выстрел из лука

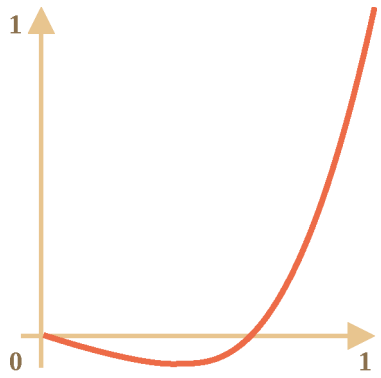
Эта функция совершает «выстрел из лука». В начале «натягивается тетива», а затем «выстрел».

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`:



Для анимации мы используем `x` с определённым значением. Пример для `x` со значением `1.5`:

Отскоки

Представьте, что мы 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)
    }
  }
}
```

В действии:

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 * tim
}
```

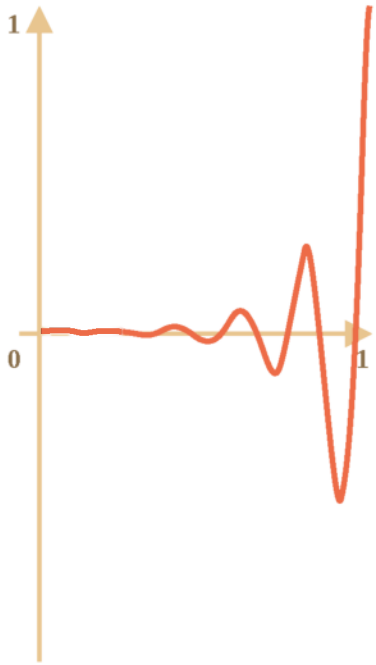


График для `x=1.5`:

В действии со значением `x=1.5`:

Реверсивные функции: ease*

Итак у нас получилась коллекция функций расчёта времени. Их прямое использование называется «easeIn».

Иногда нужно показать анимацию в обратном режиме. Преобразование функции, которое даёт такой эффект, называется «easeOut».

easeOut

В режиме «easeOut» `timing` функции оборачиваются функцией `timingEaseOut`:

```
timingEaseOut(timeFraction) = 1 - timing(1 - timeFraction)
```

Другими словами, мы имеем функцию «преобразования» – `makeEaseOut`, которая берет «обычную» функцию расчёта времени и возвращает обёртку над ней:

```
// принимает функцию расчёта времени и возрващает преобразованный вариант
function makeEaseOut(timing) {
  return function(timeFraction) {
    return 1 - timing(1 - timeFraction);
  }
}
```

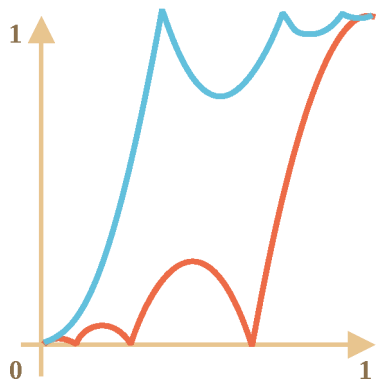
Например, мы можем взять функцию `bounce` описанную выше:

```
let bounceEaseOut = makeEaseOut(bounce);
```

Таким образом, отскоки будут не в начале функции, а в конце. Смотрится гораздо лучше:

https://plnkr.co/edit/mXVKNcxYOdWLtqPXICbx?p=preview

Ниже мы можем 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 внизу, а затем резко подпрыгивает вверх.
- Обратный `easeOut` – объект вначале прыгает вверх, и затем скачет там.

easeInOut

Мы можем 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:

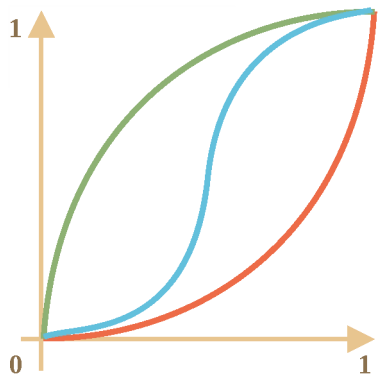
```
function makeEaseInOut(timing) {
  return function(timeFraction) {
    if (timeFraction < .5)
      return timing(2 * timeFraction) / 2;
    else
      return (2 - timing(2 * (1 - timeFraction))) / 2;
  }
}

bounceEaseInOut = makeEaseInOut(bounce);
```

В действии, `bounceEaseInOut`:

https://plnkr.co/edit/ftQcMBFeEjqztHE3RnZI?p=preview

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.

Более интересная функция «draw»

Вместо передвижения элемента мы можем делать что-нибудь ещё. Всё, что нам нужно – это правильно написать функцию `draw`.

Вот пример «скачущей» анимации набирающегося текста:

https://plnkr.co/edit/Yd3kGZRyna0eRWYYDvHE?p=preview

Итого

JavaScript может помочь в тех случаях, когда CSS не справляется или нужен жёсткий контроль над анимацией. JavaScript-анимации должны быть сделаны с помощью `requestAnimationFrame`. Это встроенный метод браузера, который вызывает переданную в него функцию в тот момент, когда браузер готовится совершить перерисовку (обычно это происходит быстро, но конкретные задержки зависят от браузера).

Когда вкладка скрыта, на ней совсем не происходит перерисовок, и функция не будет вызвана: анимация будет приостановлена и не потратит ресурсы. Это хорошо.

Вспомогательная функция `animate` для создания анимации:

```
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);
    }

  });
}
```

Опции:

- `duration` – общая продолжительность анимации в миллисекундах.
- `timing` – функция вычисления прогресса анимации. Получается момент времени от 0 до 1, возвращает прогресс анимации, обычно тоже от 0 до 1.
- `draw` – функция отрисовки анимации.

Конечно, мы могли бы улучшить вспомогательную функцию и добавить в неё больше наворотов. Но JavaScript-анимации не каждый день используются, а только когда хотят сделать что-то интересное и необычное. Не стоит усложнять функцию до тех пор пока это вам не понадобились.

JavaScript-анимации могут использовать любые функции расчёта времени. Мы рассмотрели множество примеров и их вариаций, чтобы сделать их ещё более универсальными. В отличие от CSS, мы здесь не ограничены только кривой Безье.

То же самое и с `draw`: мы можем анимировать всё что угодно, не только CSS-свойства.

Задачи

Анимируйте прыгающий мячик

важность: 5

Создайте прыгающий мячик. Кликните, чтобы посмотреть, как это должно выглядеть:

Открыть песочницу для задачи.

К решению

Анимируйте мячик, прыгающий вправо

важность: 5

Сделайте отскок мяча вправо. Как в примере:

Напишите код для анимации. Расстояние слева `100px`.

Возьмите решение предыдущей задачи Анимируйте прыгающий мячик за основу.

К решению

Веб-компоненты

Веб-компоненты — совокупность стандартов, которая позволяет создавать новые, пользовательские HTML-элементы со своими свойствами, методами, инкапсулированными DOM и стилями.

С орбитальной высоты

Этот раздел описывает набор современных стандартов для «веб-компонентов».

На текущий момент, эти стандарты находятся в процессе разработки. Некоторые фичи имеют хорошую поддержку и интеграцию в современный стандарт HTML/DOM, в то время как другие пока ещё в черновиках. Вы можете попробовать примеры в любом современном браузере (Google Chrome, скорее

всего, имеет наиболее полную поддержку, так как ребята из Google стоят за большинством спецификаций по этой теме).

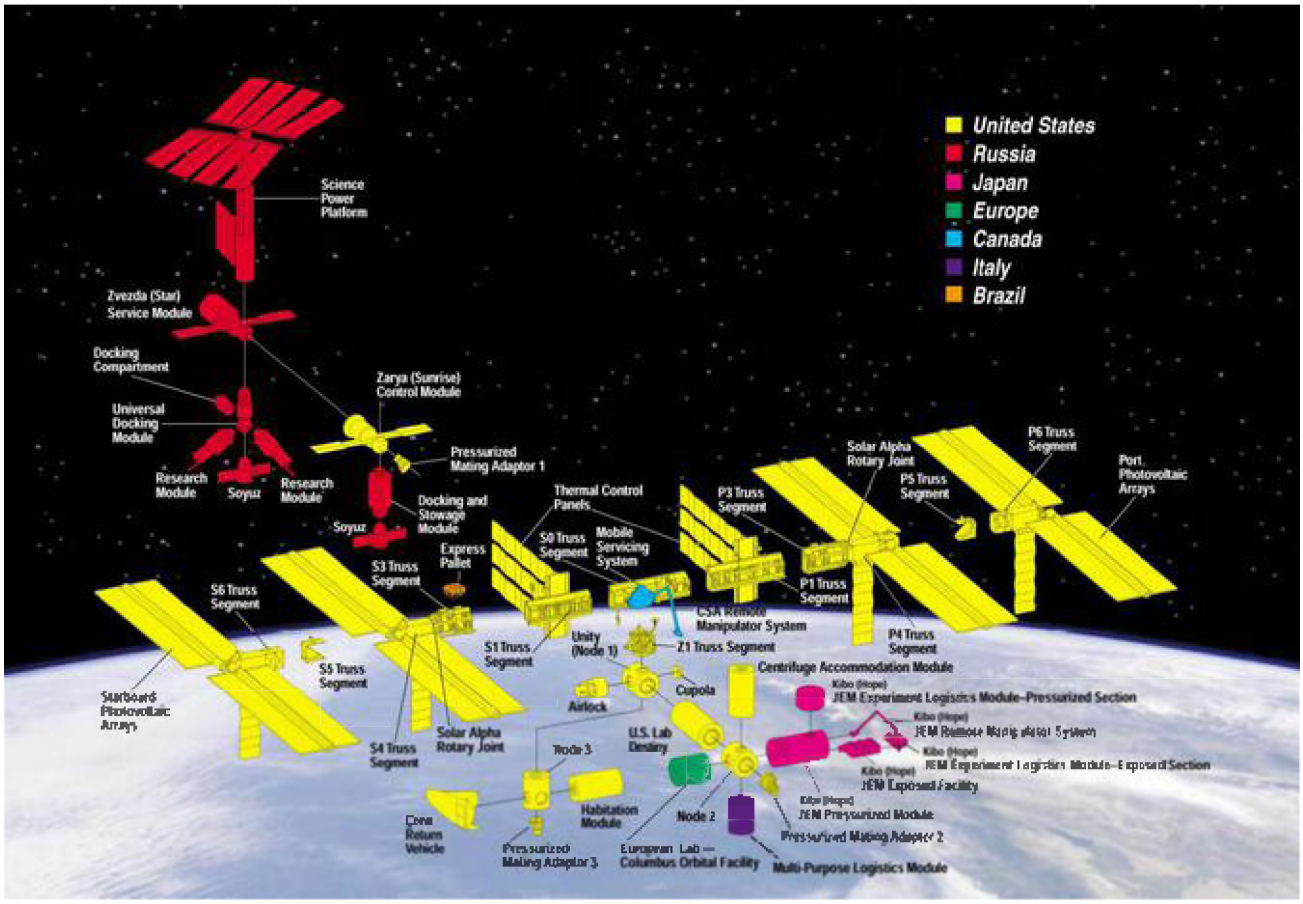
Что общего между…

В идее самостоятельного компонента нет ничего нового. Такой подход используется во многих фреймворках.

Прежде чем мы погрузимся в детали реализации, взгляните на это великое достижение человечества:

Это международная космическая станция (МКС).

А это то, как она устроена (приблизительно):



Международная космическая станция:

- Состоит из множества компонентов.
- Каждый компонент в свою очередь, состоит из множества более мелких деталей.
- Компоненты имеют очень сложное устройство, и гораздо сложнее большинства сайтов.
- Компоненты разработаны на международной основе, командами из разных стран и говорящих на разных языках.

…И эта штука летает, поддерживая жизни людей в космосе!

Как создаются столь сложные устройства?

Какие принципы мы могли бы позаимствовать, чтобы сделать нашу разработку такой же надёжной и масштабируемой? Или, по крайней мере, приблизиться к такому уровню.

Компонентная архитектура

Хорошо известное правило разработки сложного программного обеспечения гласит: не создавай сложное программное обеспечение.

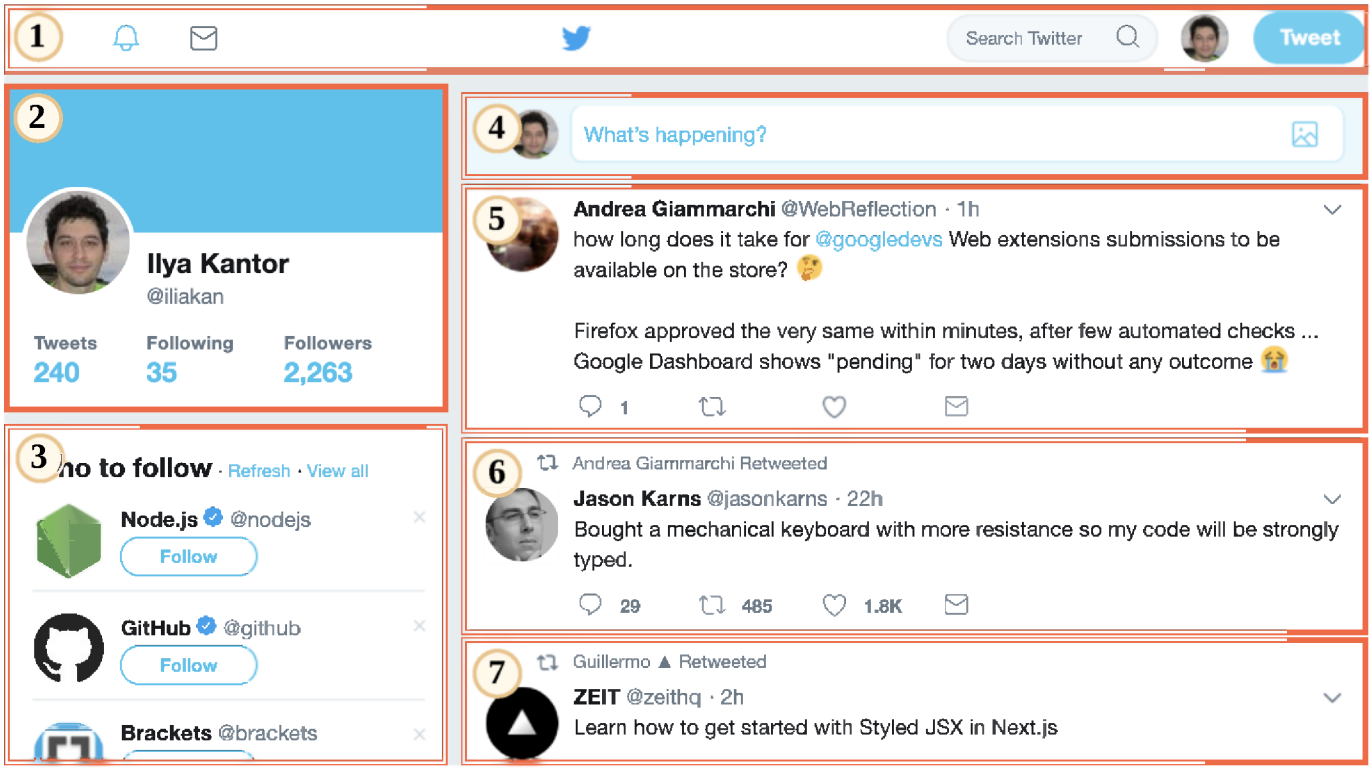
Если что то становится сложным – раздели это на более простые части и соедини наиболее очевидным способом.

Хороший архитектор – это тот, кто может сделать сложное простым.

Мы можем разделить пользовательский интерфейс на визуальные компоненты: каждый из них занимает своё место на странице, выполняет определённую задачу, и отделен от остальных.

Рассмотрим какой-нибудь сайт, например Twitter.

Он естественным образом разделён на компоненты:



1. Верхняя навигация.
2. Данные пользователя.
3. Предложения подписаться.
4. Форма отправки сообщения.
5. (а так же 6 и 7) – сообщения.

Компоненты могут содержать подкомпоненты, например сообщения могут быть частями родительского компонента «список сообщений». Кликабельное фото пользователя может быть самостоятельным компонентом и т.д.

Как мы определяем, что является компонентом? Это приходит из соображений здравого смысла, а также с интуицией и опытом. Обычно это объект, отделимый визуально, который мы можем описать с точки зрения того, что он делает и как он взаимодействует со страницей. В примере выше, страница содержит блоки, каждый из которых играет свою роль, и логично выделить их в компоненты.

Компонент имеет:

- свой собственный JavaScript-класс.
- DOM-структура управляется исключительно своим классом, и внешний код не имеет к ней доступа (принцип «инкапсуляции»).
- CSS-стили, применённые к компоненту.
- API: события, методы класса и т.п., для взаимодействия с другими компонентами.

Ещё раз заметим, в компонентном подходе как таковом нет ничего особенного.

Существует множество фреймворков и методов разработки для их создания, каждый из которых со своими плюсами и минусами. Обычно особые CSS классы и соглашения используются для эмуляции компонентов – области видимости CSS и инкапсуляция DOM.

«Веб-компоненты» предоставляют встроенные возможности браузера для этого, поэтому нам больше не нужно эмулировать их.

- Пользовательские элементы – для определения пользовательских HTML-элементов.
- Теневой DOM – для создания внутреннего DOM компонента, скрытого от остальных.
- Области видимости CSS – для определения стилей, которые применяются только внутри теневого DOM компонента.
- Перенаправление событий и другие мелочи для создания более удобных в разработке пользовательских компонентов.

В следующей главе мы погрузимся в «пользовательские элементы» – фундаментальную для веб-компонентов технологию, имеющую хорошую поддержку в браузерах.

Пользовательские элементы (Custom Elements)

Мы можем создавать пользовательские HTML-элементы, описываемые нашим классом, со своими методами и свойствами, событиями и так далее.

Как только пользовательский элемент определён, мы можем использовать его наравне со встроенными HTML-элементами.

Это замечательно, ведь словарь HTML-тегов богат, но не бесконечен. Не существует `<easy-tabs>`, `<sliding-carousel>`, `<beautiful-upload>`... Просто подумайте о любом другом теге, который мог бы нам понадобиться.

Мы можем определить их с помощью специального класса, а затем использовать, как если бы они всегда были частью HTML.

Существует два вида пользовательских элементов:

1. **Автономные пользовательские элементы** – «полностью новые» элементы, расширяющие абстрактный класс `HTMLElement`.
2. **Пользовательские встроенные элементы** – элементы, расширяющие встроенные, например кнопку `HTMLButtonElement` и т.п.

Сначала мы разберёмся с автономными элементами, а затем перейдём к пользовательским встроенным.

Чтобы создать пользовательский элемент, нам нужно сообщить браузеру ряд деталей о нём: как его показать, что делать, когда элемент добавляется или удаляется со страницы и т.д.

Это делается путём создания класса со специальными методами. Это просто, так как существует всего несколько методов, и все они являются необязательными.

Вот набросок с полным списком:

```
class MyElement extends HTMLElement {
  constructor() {
    super();
    // элемент создан
  }

  connectedCallback() {
    // браузер вызывает этот метод при добавлении элемента в документ
    // (может вызываться много раз, если элемент многократно добавляется/удаляется)
  }

  disconnectedCallback() {
    // браузер вызывает этот метод при удалении элемента из документа
    // (может вызываться много раз, если элемент многократно добавляется/удаляется)
  }

  static get observedAttributes() {
    return [/* массив имён атрибутов для отслеживания их изменений */];
  }

  attributeChangedCallback(name, oldValue, newValue) {
    // вызывается при изменении одного из перечисленных выше атрибутов
  }

  adoptedCallback() {
    // вызывается, когда элемент перемещается в новый документ
    // (происходит в document.adoptNode, используется очень редко)
  }

  // у элемента могут быть ещё другие методы и свойства
}
```

После этого нам нужно зарегистрировать элемент:

```
// сообщим браузеру, что <my-element> обслуживается нашим новым классом
customElements.define("my-element", MyElement);
```

Теперь для любых HTML-элементов с тегом `<my-element>` создаётся экземпляр `MyElement` и вызываются вышеупомянутые методы. Также мы можем использовать `document.createElement('my-element')` в JavaScript.

Имя пользовательского элемента должно содержать дефис `-`

Имя пользовательского элемента должно содержать дефис `-`, например, `my-element` и `super-button` – валидные имена, а `myelement` – нет.

Это чтобы гарантировать отсутствие конфликтов имён между встроенными и пользовательскими элементами HTML.

Пример: «time-formatted»

Например, элемент `<time>` уже существует в HTML для даты/времени. Но сам по себе он не выполняет никакого форматирования. Давайте создадим элемент `<time-formatted>`, который отображает время в удобном формате с учётом языка:

```
<script>
class TimeFormatted extends HTMLElement { // (1)

  connectedCallback() {
    let date = new Date(this.getAttribute('datetime') || Date.now());

    this.innerHTML = new Intl.DateTimeFormat("default", {
      year: this.getAttribute('year') || undefined,
      month: this.getAttribute('month') || undefined,
      day: this.getAttribute('day') || undefined,
      hour: this.getAttribute('hour') || undefined,
      minute: this.getAttribute('minute') || undefined,
      second: this.getAttribute('second') || undefined,
      timeZoneName: this.getAttribute('time-zone-name') || undefined,
    }).format(date);
  }

}
```

```
customElements.define("time-formatted", TimeFormatted); // (2)
</script>

<!-- (3) -->
<time-formatted datetime="2019-12-01"
  year="numeric" month="long" day="numeric"
  hour="numeric" minute="numeric" second="numeric"
  time-zone-name="short"
></time-formatted>
```

December 1, 2019, 3:00:00 AM GMT+3

1. Класс имеет только один метод `connectedCallback()` – браузер вызывает его, когда элемент `<time-formatted>` добавляется на страницу (или когда HTML-парсер обнаруживает его), и он использует встроенный форматировщик данных Intl.DateTimeFormat , хорошо поддерживаемый в браузерах, чтобы показать красиво отформатированное время.
2. Нам нужно зарегистрировать наш новый элемент, используя `customElements.define(tag, class)`.
3. И тогда мы сможем использовать его везде.

Обновление пользовательских элементов

Если браузер сталкивается с элементами `<time-formatted>` до `customElements.define`, то это не ошибка. Но элемент пока неизвестен, как и любой нестандартный тег.

Такие «неопределённые» элементы могут быть стилизованы с помощью CSS селектора `:not(:defined)`.

Когда вызывается `customElements.define`, они «обновляются»: для каждого создаётся новый экземпляр `TimeFormatted` и вызывается `connectedCallback`. Они становятся `:defined`.

Чтобы получить информацию о пользовательских элементах, есть следующие методы:

- `customElements.get(name)` – возвращает класс пользовательского элемента с указанным именем `name`,
- `customElements.whenDefined(name)` – возвращает промис, который переходит в состояние «успешно выполнен» (без значения), когда определён пользовательский элемент с указанным именем `name`.

Рендеринг происходит в `connectedCallback`, не в `constructor`

В приведённом выше примере содержимое элемента рендерится (создаётся) в `connectedCallback`.

Почему не в `constructor`?

Причина проста: когда вызывается `constructor`, делать это слишком рано. Экземпляр элемента создан, но на этом этапе браузер ещё не обработал/назначил атрибуты: вызовы `getAttribute` вернули бы `null`. Так что мы не можем рендерить здесь.

Кроме того, если подумать, это лучше с точки зрения производительности – отложить работу до тех пор, пока она действительно не понадобится.

`connectedCallback` срабатывает, когда элемент добавляется в документ. Не просто добавляется к другому элементу как дочерний, но фактически становится частью страницы. Таким образом, мы можем построить отдельный DOM, создать элементы и подготовить их для последующего использования. Они будут рендериться только тогда, когда попадут на страницу.

Наблюдение за атрибутами

В текущей реализации `<time-formatted>` после того, как элемент отрендерился, дальнейшие изменения атрибутов не дают никакого эффекта. Это странно для HTML-элемента. Обычно, когда мы изменяем атрибут, например `a.href`, мы ожидаем, что изменение будет видно сразу. Так что давайте исправим это.

Мы можем наблюдать за атрибутами, поместив их список в статический геттер `observedAttributes()`. При изменении таких атрибутов вызывается `attributeChangedCallback`. Он срабатывает не для любого атрибута по соображениям производительности.

Вот новый `<time-formatted>`, который автоматически обновляется при изменении атрибутов:

```
<script>
class TimeFormatted extends HTMLElement {

  render() { // (1)
    let date = new Date(this.getAttribute('datetime') || Date.now());

    this.innerHTML = new Intl.DateTimeFormat("default", {
      year: this.getAttribute('year') || undefined,
      month: this.getAttribute('month') || undefined,
```

```
      day: this.getAttribute('day') || undefined,
      hour: this.getAttribute('hour') || undefined,
      minute: this.getAttribute('minute') || undefined,
      second: this.getAttribute('second') || undefined,
      timeZoneName: this.getAttribute('time-zone-name') || undefined,
    }).format(date);
  }

  connectedCallback() { // (2)
    if (!this.rendered) {
      this.render();
      this.rendered = true;
    }
  }

  static get observedAttributes() { // (3)
    return ['datetime', 'year', 'month', 'day', 'hour', 'minute', 'second', 'time-z
  }

  attributeChangedCallback(name, oldValue, newValue) { // (4)
    this.render();
  }

}

customElements.define("time-formatted", TimeFormatted);
</script>

<time-formatted id="elem" hour="numeric" minute="numeric" second="numeric"></time-f

<script>
setInterval(() => elem.setAttribute('datetime', new Date()), 1000); // (5)
</script>
```

8:34:38 PM

1. Логика рендеринга перенесена во вспомогательный метод `render()`.
2. Мы вызываем его один раз, когда элемент вставляется на страницу.
3. При изменении атрибута, указанного в `observedAttributes()`, вызывается `attributeChangedCallback`.
4. …и происходит перерендеринг элемента.
5. В конце мы легко создаём живой таймер.

Порядок рендеринга

Когда HTML-парсер строит DOM, элементы обрабатываются друг за другом, родители до детей. Например, если у нас есть `<outer><inner></inner>`

`</outer>`, то элемент `<outer>` создаётся и включается в DOM первым, а затем `<inner>`.

Это приводит к важным последствиям для пользовательских элементов.

Например, если пользовательский элемент пытается получить доступ к `innerHTML` в `connectedCallback`, он ничего не получает:

```
<script>
customElements.define('user-info', class extends HTMLElement {

  connectedCallback() {
    alert(this.innerHTML); // пусто (*)
  }

});
</script>

<user-info>Джон</user-info>
```

Если вы запустите это, `alert` будет пуст.

Это происходит именно потому, что на этой стадии ещё не существуют дочерние элементы, DOM не завершён. HTML-парсер подключил пользовательский элемент `<user-info>` и теперь собирается перейти к его дочерним элементам, но пока не сделал этого.

Если мы хотим передать информацию в пользовательский элемент, мы можем использовать атрибуты. Они доступны сразу.

Или, если нам действительно нужны дочерние элементы, мы можем отложить доступ к ним, используя `setTimeout` с нулевой задержкой.

Это работает:

```
<script>
customElements.define('user-info', class extends HTMLElement {

  connectedCallback() {
    setTimeout(() => alert(this.innerHTML)); // Джон (*)
  }

});
</script>

<user-info>Джон</user-info>
```

Теперь `alert` в строке `(*)` показывает «Джон», поскольку мы запускаем его асинхронно, после завершения парсинга HTML. Мы можем обработать дочерние элементы при необходимости и завершить инициализацию.

С другой стороны, это решение также не идеально. Если вложенные пользовательские элементы тоже используют `setTimeout` для инициализации, то они встают в очередь: первым запускается внешний `setTimeout`, а затем внутренний.

Так что внешний элемент завершает инициализацию раньше внутреннего.

Продемонстрируем это на примере:

```
<script>
customElements.define('user-info', class extends HTMLElement {
  connectedCallback() {
    alert(`${this.id} connected.`);
    setTimeout(() => alert(`${this.id} initialized.`));
  }
});
</script>

<user-info id="outer">
  <user-info id="inner"></user-info>
</user-info>
```

Порядок вывода:

1. outer connected.
2. inner connected.
3. outer initialized.
4. inner initialized.

Мы ясно видим, что внешний элемент `outer` завершает инициализацию `(3)` до внутреннего `inner` `(4)`.

Нет встроенного колбэка, который срабатывает после того, как вложенные элементы готовы. Если нужно, мы можем реализовать подобное самостоятельно. Например, внутренние элементы могут отправлять события наподобие `initialized`, а внешние могут слушать и реагировать на них.

Модифицированные встроенные элементы

Новые элементы, которые мы создаём, такие как `<time-formatted>`, не имеют связанной с ними семантики. Они не известны поисковым системам, а

устройства для людей с ограниченными возможностями не могут справиться с ними.

Но такие вещи могут быть важны. Например, поисковой системе было бы интересно узнать, что мы показываем именно время. А если мы делаем специальный вид кнопки, почему не использовать существующую функциональность `<button>`?

Мы можем расширять и модифицировать встроенные HTML-элементы, наследуя их классы.

Например, кнопки `<button>` являются экземплярами класса `HTMLButtonElement`, давайте построим элемент на его основе.

1. Унаследуем `HTMLButtonElement` нашим классом:

   ```
   class HelloButton extends HTMLButtonElement { /* методы пользовательского элемент
   ```

2. Предоставим третий аргумент в `customElements.define`, указывающий тег:

   ```
   customElements.define('hello-button', HelloButton, {extends: 'button'});
   ```

   Бывает, что разные теги имеют одинаковый DOM-класс, поэтому указание тега необходимо.

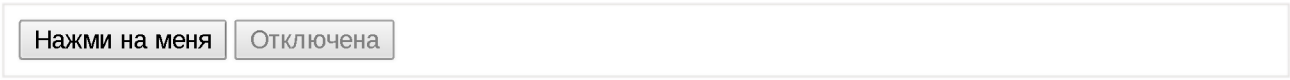
3. В конце, чтобы использовать наш пользовательский элемент, вставим обычный тег `<button>`, но добавим к нему `is="hello-button"`:

   ```
   <button is="hello-button">...</button>
   ```

Вот полный пример:

```
<script>
// Кнопка, говорящая "привет" по клику
class HelloButton extends HTMLButtonElement {
  constructor() {
    super();
    this.addEventListener('click', () => alert("Привет!"));
  }
}

customElements.define('hello-button', HelloButton, {extends: 'button'});
</script>
```

```
<button is="hello-button">Нажми на меня</button>

<button is="hello-button" disabled>Отключена</button>
```



Наша новая кнопка расширяет встроенную. Так что она сохраняет те же стили и стандартные возможности, наподобие атрибута `disabled`.

Ссылки

- HTML Living Standard: https://html.spec.whatwg.org/#custom-elements .
- Совместимость: https://caniuse.com/#feat=custom-elements .

Итого

Есть два типа пользовательских элементов:

1. «Автономные» – новые теги, расширяющие `HTMLElement`.

   Схема определения:

   ```
   class MyElement extends HTMLElement {
     constructor() { super(); /* ... */ }
     connectedCallback() { /* ... */ }
     disconnectedCallback() { /* ... */  }
     static get observedAttributes() { return [/* ... */]; }
     attributeChangedCallback(name, oldValue, newValue) { /* ... */ }
     adoptedCallback() { /* ... */ }
    }
   customElements.define('my-element', MyElement);
   /* <my-element> */
   ```

2. «Модифицированные встроенные элементы» – расширения существующих элементов.

   Требуют ещё один аргумент в `.define` и атрибут `is="..."` в HTML:

   ```
   class MyButton extends HTMLButtonElement { /*...*/ }
   customElements.define('my-button', MyElement, {extends: 'button'});
   /* <button is="my-button"> */
   ```

Пользовательские элементы широко поддерживаются браузерами. Edge немного отстаёт, но есть полифил

https://github.com/webcomponents/webcomponentsjs .

Задачи

Элемент "живой таймер"

У нас уже есть элемент `<time-formatted>`, показывающий красиво отформатированное время.

Создайте элемент `<live-timer>`, показывающий текущее время:

1. Внутри он должен использовать `<time-formatted>`, не дублировать его функциональность.
2. Должен тикать (обновляться) каждую секунду.
3. На каждом тике должно генерироваться пользовательское событие с именем `tick`, содержащее текущую дату в `event.detail` (смотрите главу Генерация пользовательских событий).

Использование:

```
<live-timer id="elem"></live-timer>

<script>
  elem.addEventListener('tick', event => console.log(event.detail));
</script>
```

Демо:

8:34:38 PM

Открыть песочницу для задачи.

К решению

Shadow DOM

Теневой DOM («Shadow DOM») используется для инкапсуляции. Благодаря ему в компоненте есть собственное «теневое» DOM-дерево, к которому нельзя просто так обратиться из главного документа, у него могут быть изолированные CSS-правила и т.д.

Встроенный теневой DOM

Задумывались ли вы о том, как устроены и стилизованы сложные браузерные элементы управления?

Например, `<input type="range">`:

Браузер рисует их своими силами и по своему усмотрению. Их DOM-структура обычно нам не видна, но в инструментах разработчика можно её посмотреть. К примеру, в Chrome для этого нужно активировать пункт «Show user agent shadow DOM».

После этого `<input type="range">` выглядит так:

То, что находится под `#shadow-root` – и называется «shadow DOM» (теневой DOM).

Мы не можем получить доступ к теневому DOM встроенных элементов с помощью обычных JavaScript-вызовов или с помощью селекторов. Это не просто обычные потомки, это мощное средство инкапсуляции.

В примере выше можно увидеть полезный атрибут `pseudo`. Он нестандартный и существует по историческим причинам. С его помощью можно стилизовать подэлементы через CSS, например, так:

```
<style>
/* делаем цвет шкалы ползунка красным */
input::-webkit-slider-runnable-track {
  background: red;
}
</style>

<input type="range">
```

Ещё раз заметим, что `pseudo` – нестандартный атрибут. Если говорить хронологически, то сначала браузеры начали экспериментировать с

инкапсуляцией внутренних DOM-структур для элементов, а уже потом, через некоторое время, появился стандарт Shadow DOM, который позволяет делать то же самое нам, разработчикам.

Далее мы воспользуемся современным стандартом Shadow DOM, описанным в спецификации DOM spec и других спецификациях.

Теневое дерево

Каждый DOM-элемент может иметь 2 типа поддеревьев DOM:

1. Light tree – обычное, «светлое», DOM-поддерево, состоящее из HTML-потомков. Все поддеревья, о которых мы говорили в предыдущих главах, были «light».
2. Shadow tree – скрытое, «теневое», DOM-поддерево, не отражённое в HTML, скрытое от посторонних глаз.

Если у элемента имеются оба поддерева, браузер отрисовывает только теневое дерево. Также мы всё же можем задать «композицию» теневого и обычного деревьев. Позже в главе Слоты теневого DOM, композиция мы рассмотрим детали.

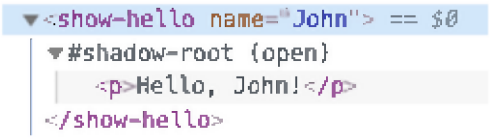
Теневое дерево можно использовать в пользовательских элементах (Custom Elements), чтобы спрятать внутренности компонента и применить к ним локальные стили.

Например, этот `<show-hello>` элемент прячет свой внутренний DOM в теневом дереве:

```
<script>
customElements.define('show-hello', class extends HTMLElement {
  connectedCallback() {
    const shadow = this.attachShadow({mode: 'open'});
    shadow.innerHTML = `<p>
      Hello, ${this.getAttribute('name')}
    </p>`;
  }
});
</script>

<show-hello name="John"></show-hello>
```

Hello, John

А вот как получившийся DOM выглядит в инструментах разработчика в Chrome, весь контент внутри «#shadow-root»:



Итак, вызов `elem.attachShadow({mode: …})` создаёт теневое дерево.

Есть два ограничения:

1. Для каждого элемента мы можем создать только один shadow root.
2. В качестве `elem` может быть использован пользовательский элемент (Custom Element), либо один из следующих элементов: «article», «aside», «blockquote», «body», «div», «footer», «h1…h6», «header», «main» «nav», «p», «section» или «span». Остальные, например, `<img>`, не могут содержать теневое дерево.

Свойство `mode` задаёт уровень инкапсуляции. У него может быть только два значения:

- `"open"` – корень теневого дерева («shadow root») доступен как `elem.shadowRoot`.

  Любой код может получить теневое дерево `elem`.

- `"closed"` – `elem.shadowRoot` всегда возвращает `null`.

  До теневого DOM в таком случае мы сможем добраться только по ссылке, которую возвращает `attachShadow` (и, скорее всего, она будет спрятана внутри класса). Встроенные браузерные теневые деревья, такие как у `<input type="range">`, закрыты. До них не добраться.

С возвращаемым методом `attachShadow` объектом корнем теневого дерева, можно работать как с обычным DOM-элементом: менять его `innerHTML` или использовать методы DOM, такие как `append`, чтобы заполнить его.

Элемент с корнем теневого дерева называется – «хозяин» (host) теневого дерева, и он доступен в качестве свойства `host` у shadow root:

```
// при условии, что {mode: "open"}, иначе elem.shadowRoot равен null
alert(elem.shadowRoot.host === elem); // true
```

Инкапсуляция

Теневой DOM отделён от главного документа:

1. Элементы теневого DOM не видны из обычного DOM через `querySelector`. В частности, элементы теневого DOM могут иметь такие

же идентификаторы, как у элементов в обычном DOM (light DOM). Они должны быть уникальными только внутри теневого дерева.
2. У теневого DOM свои стили. Стили из внешнего DOM не применятся.

Например:

```
<style>
  /* стили документа не применятся в теневом дереве внутри #elem (1) */
  p { color: red; }
</style>

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

<script>
  elem.attachShadow({mode: 'open'});
    // у теневого дерева свои стили (2)
  elem.shadowRoot.innerHTML = `
    <style> p { font-weight: bold; } </style>
    <p>Hello, John!</p>
  `;

  // <p> виден только запросам внутри теневого дерева (3)
  alert(document.querySelectorAll('p').length); // 0
  alert(elem.shadowRoot.querySelectorAll('p').length); // 1
</script>
```

1. Стили главного документа не влияют на теневое дерево.
2. …Но свои внутренние стили работают.
3. Чтобы добраться до элементов в теневом дереве, нам нужно искать их изнутри самого дерева.

Ссылки

- DOM: https://dom.spec.whatwg.org/#shadow-trees
- Совместимость: https://caniuse.com/#feat=shadowdomv1
- Теневой DOM упоминается во многих других спецификациях, например DOM Parsing указывает, что у shadow root есть `innerHTML`.

Итого

Теневой DOM – это способ создать свой, изолированный, DOM для компонента.

1. `shadowRoot = elem.attachShadow({mode: open|closed})` – создаёт теневой DOM для `elem`. Если `mode="open"`, он доступен через свойство

`elem.shadowRoot`.
2. Мы можем создать подэлементы внутри `shadowRoot` с помощью `innerHTML` или других методов DOM.

Элементы теневого DOM:

- Обладают собственной областью видимости идентификаторов
- Невидимы JavaScript селекторам из главного документа, таким как `querySelector`,
- Стилизуются своими стилями из теневого дерева, не из главного документа.

Теневой DOM, если имеется, отрисовывается браузером вместо обычных потомков (light DOM). В главе Слоты теневого DOM, композиция мы разберём, делать их композицию.

Элемент "template"

Встроенный элемент `<template>` предназначен для хранения шаблона HTML. Браузер полностью игнорирует его содержимое, проверяя лишь синтаксис, но мы можем использовать этот элемент в JavaScript, чтобы создать другие элементы.

В теории, для хранения разметки мы могли бы создать невидимый элемент в любом месте HTML. Что такого особенного в `<template>`?

Во-первых, его содержимым может быть любой корректный HTML-код, даже такой, который обычно нуждается в специальном родителе.

К примеру, мы можем поместить сюда строку таблицы `<tr>`:

```
<template>
  <tr>
    <td>Содержимое</td>
  </tr>
</template>
```

Обычно, если элемент `<tr>` мы поместим, скажем, в `<div>`, браузер обнаружит неправильную структуру DOM и «исправит» её, добавив снаружи `<table>`. Это может оказаться не тем, что мы хотели. `<template>` же оставит разметку ровно такой, какой мы её туда поместили.

Также внутри `<template>` можно поместить стили и скрипты:

```
<template>
  <style>
```

```
    p { font-weight: bold; }
  </style>
  <script>
    alert("Привет");
  </script>
</template>
```

Браузер рассматривает содержимое `<template>` как находящееся «вне документа»: стили, определённые в нём, не применяются, скрипты не выполнятся, `<video autoplay>` не запустится и т.д.

Содержимое оживёт (скрипт выполнится), когда мы поместим его в нужное нам место.

Использование template

Содержимое шаблона доступно по его свойству `content` в качестве DocumentFragment – особый тип DOM-узла.

Можно обращаться с ним так же, как и с любыми другими DOM-узлами, за исключением одной особенности: когда мы его куда-то вставляем, то в это место вставляется не он сам, а его дети.

Пример:

```
<template id="tmpl">
  <script>
    alert("Привет");
  </script>
  <div class="message">Привет, Мир!</div>
</template>

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

  // Клонируем содержимое шаблона для того, чтобы переиспользовать его несколько ра
  elem.append(tmpl.content.cloneNode(true));

  document.body.append(elem);
  // Сейчас скрипт из <template> выполнится
</script>
```

Давайте перепишем пример Shadow DOM из прошлой главы учебника с помощью `<template>`:

```
<template id="tmpl">
  <style> p { font-weight: bold; } </style>
```

```
  <p id="message"></p>
</template>

<div id="elem">Нажми на меня</div>

<script>
  elem.onclick = function() {
    elem.attachShadow({mode: 'open'});

    elem.shadowRoot.append(tmpl.content.cloneNode(true)); // (*)

    elem.shadowRoot.getElementById('message').innerHTML = "Привет из теней!";
  };
</script>
```

Нажми на меня

Когда мы клонируем и вставляем `tmpl.content` в строке `(*)`, то, так как это `DocumentFragment`, вместо него вставляются его потомки (`<style>`, `<p>`).

Именно они и формируют теневой DOM:

```
<div id="elem">
  #shadow-root
    <style> p { font-weight: bold; } </style>
    <p id="message"></p>
</div>
```

Итого

Подводим итоги:

- Содержимым `<template>` может быть любой синтаксически корректный HTML.
- Содержимое `<template>` считается находящимся «вне документа», поэтому оно ни на что не влияет.
- Мы можем получить доступ к `template.content` из JavaScript, клонировать его и переиспользовать в новом компоненте.

Элемент `<template>` уникальный по следующим причинам:

- Браузер проверяет правильность HTML-синтаксиса в нём (в отличие от строк в скриптах).
- …При этом позволяет использовать любые HTML-теги, даже те, которые без соответствующей обёртки не используются (например `<tr>`).

- Его содержимое оживает (скрипты выполняются, `<video autoplay>` проигрывается и т. д.), когда помещается в документ.

Элемент `<template>` не поддерживает итерацию, связывания данных или подстановки переменных. Однако эти возможности можно реализовать поверх него.

Слоты теневого DOM, композиция

Многим типам компонентов, таким как вкладки, меню, галереи изображений и другие, нужно какое-то содержимое для отображения.

Так же, как встроенный в браузер `<select>` ожидает получить контент пунктов `<option>`, компонент `<custom-tabs>` может ожидать, что будет передано фактическое содержимое вкладок, а `<custom-menu>` – пунктов меню.

Код, использующий меню `<custom-menu>`, может выглядеть так:

```
<custom-menu>
  <title>Сладости</title>
  <item>Леденцы</item>
  <item>Фруктовые тосты</item>
  <item>Кексы</item>
</custom-menu>
```

…Затем компонент должен правильно его отобразить – как обычное меню с заданным названием и пунктами, обрабатывать события меню и т.д.

Как это реализовать?

Можно попробовать проанализировать содержимое элемента и динамически скопировать и переставить DOM-узлы. Это возможно, но если мы будем перемещать элементы в теневой DOM, CSS-стили документа не будут применяться, и мы потеряем визуальное оформление. Кроме того, нужно будет писать дополнительный код.

К счастью, нам этого делать не нужно. Теневой DOM поддерживает элементы `<slot>`, которые автоматически наполняются контентом из обычного, «светлого» DOM-дерева.

Именованные слоты

Давайте рассмотрим работу слотов на простом примере.

Теневой DOM `<user-card>` имеет два слота, заполняемых из обычного DOM:

```
<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `
      <div>Имя:
        <slot name="username"></slot>
      </div>
      <div>Дата рождения:
        <slot name="birthday"></slot>
      </div>
    `;
  }
});
</script>

<user-card>
  <span slot="username">Иван Иванов</span>
  <span slot="birthday">01.01.2001</span>
</user-card>
```

Имя: Иван Иванов
Дата рождения: 01.01.2001

В теневом DOM `<slot name="X">` определяет «точку вставки» – место, где отображаются элементы с `slot="X"`.

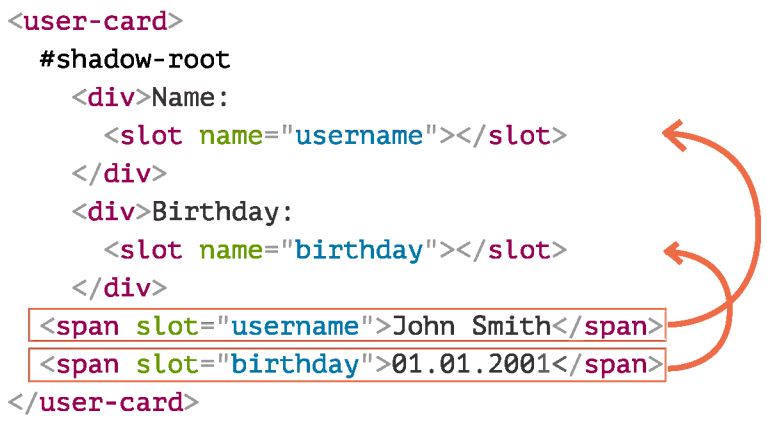
Затем браузер выполняет «композицию»: берёт элементы из обычного DOM-дерева и отображает их в соответствующих слотах теневого DOM-дерева. В результате мы получаем именно то, что хотели – компонент, который можно наполнить данными.

После выполнения скрипта структура DOM выглядит следующим образом (без учёта композиции):

```
<user-card>
  #shadow-root
    <div>Имя:
      <slot name="username"></slot>
    </div>
    <div>Дата рождения:
      <slot name="birthday"></slot>
    </div>
  <span slot="username">Иван Иванов</span>
  <span slot="birthday">01.01.2001</span>
</user-card>
```

Мы создали теневой DOM, он изображён под #shadow-root. Теперь у элемента есть два DOM-дерева: обычное («светлое») и теневое.

Чтобы отобразить содержимое, для каждого `<slot name="...">` в теневом DOM браузер ищет `slot="..."` с таким же именем в обычном DOM. Эти элементы отображаются внутри слотов:



В результате выстраивается так называемое «развёрнутое» (flattened) DOM-дерево:

```
<user-card>
  #shadow-root
    <div>Имя:
      <slot name="username">
        <!-- элемент слота вставляется в слот -->
        <span slot="username">Иван Иванов</span>
      </slot>
    </div>
    <div>Дата рождения:
      <slot name="birthday">
        <span slot="birthday">01.01.2001</span>
      </slot>
    </div>
</user-card>
```

…Но развёрнутое DOM-дерево существует только для целей отображения и обработки событий. Это то, что мы видим на экране. Оно, в некотором плане, «виртуальное». Фактически в документе расположение узлов не меняется.

Это можно легко проверить, запустив `querySelector`: все узлы находятся на своих местах.

```
// узлы светлого DOM находятся в том же месте, в `<user-card>`
alert( document.querySelector('user-card span').length ); // 2
```

Так что развёрнутый DOM составляется из теневого вставкой в слоты. Браузер использует его для рендеринга и при всплытии событий (об этом позже). Но JavaScript видит документ «как есть» – до построения развёрнутого DOM-дерева.

⚠️ **Атрибут slot="..." могут иметь только дети первого уровня**

Атрибут `slot="..."` работает только на непосредственных детях элемента-хозяина теневого дерева (в нашем примере это элемент `<user-card>`). Для вложенных элементов он игнорируется.

Например, здесь второй `<span>` игнорируется (так как он не является потомком верхнего уровня элемента `<user-card>`):

```
<user-card>
  <span slot="username">Иван Иванов</span>
  <div>
    <!-- некорректный слот, должен быть на верхнем уровне user-card: -->
    <span slot="birthday">01.01.2001</span>
  </div>
</user-card>
```

Если в светлом DOM есть несколько элементов с одинаковым именем слота, они добавляются в слот один за другим.

Например, этот код:

```
<user-card>
  <span slot="username">Иван</span>
  <span slot="username">Иванов</span>
</user-card>
```

Даст такой развёрнутый DOM с двумя элементами в `<slot name="username">`:

```
<user-card>
  #shadow-root
    <div>Имя:
      <slot name="username">
        <span slot="username">Иван</span>
```

```
        <span slot="username">Иванов</span>
      </slot>
    </div>
    <div>Дата рождения:
      <slot name="birthday"></slot>
    </div>
</user-card>
```

Содержимое слота «по умолчанию»

Если мы добавляем данные в `<slot>`, это становится содержимым «по умолчанию». Браузер отображает его, если в светлом DOM-дереве отсутствуют данные для заполнения слота.

Например, в этой части теневого дерева текст `Аноним` отображается, если в светлом дереве нет значения `slot="username"`.

```
<div>Имя:
  <slot name="username">Аноним</slot>
</div>
```

Слот по умолчанию (первый без имени)

Первый `<slot>` в теневом дереве без атрибута `name` является слотом по умолчанию. Он будет отображать данные со всех узлов светлого дерева, не добавленные в другие слоты

Например, давайте добавим слот по умолчанию в наш элемент `<user-card>`; он будет собирать всю информацию о пользователе, не занесённую в другие слоты:

```
<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `
    <div>Имя:
      <slot name="username"></slot>
    </div>
    <div>Дата рождения:
      <slot name="birthday"></slot>
    </div>
    <fieldset>
      <legend>Другая информация</legend>
      <slot></slot>
    </fieldset>
    `;
```

```
  }
});
</script>

<user-card>
  <div>Я люблю плавать.</div>
  <span slot="username">Иван Иванов</span>
  <span slot="birthday">01.01.2001</span>
  <div>...И играть в волейбол!</div>
</user-card>
```

Имя: Иван Иванов
Дата рождения: 01.01.2001
Другая информация
Я люблю плавать.
...И играть в волейбол!

Всё содержимое обычного дерева, не добавленное в слоты, попало в `<fieldset>` «Другая информация».

Элементы добавляются в слот по очереди, один за другим, поэтому оба элемента данных, которые не были добавлены в слоты, попадают в слот по умолчанию.

Развёрнутое DOM-дерево выглядит так:

```
<user-card>
  #shadow-root
    <div>Имя:
      <slot name="username">
        <span slot="username">Иван Иванов</span>
      </slot>
    </div>
    <div>Дата рождения:
      <slot name="birthday">
        <span slot="birthday">01.01.2001</span>
      </slot>
    </div>
    <fieldset>
      <legend>Обо мне</legend>
      <slot>
        <div>Привет!</div>
        <div>Я Иван!</div>
      </slot>
    </fieldset>
</user-card>
```

Пример меню

Давайте вернёмся к меню `<custom-menu>`, упомянутому в начале главы.

Мы можем использовать слоты для распределения элементов.

Вот разметка для меню `<custom-menu>`:

```
<custom-menu>
  <span slot="title">Сладости</span>
  <li slot="item">Леденцы</li>
  <li slot="item">Фруктовые тосты</li>
  <li slot="item">Кексы</li>
</custom-menu>
```

Шаблон теневого DOM-дерева с правильными слотами:

```
<template id="tmpl">
  <style> /* стили меню */ </style>
  <div class="menu">
    <slot name="title"></slot>
    <ul><slot name="item"></slot></ul>
  </div>
</template>
```

1. `<span slot="title">` попадает в `<slot name="title">`.
2. В шаблоне много элементов `<li slot="item">`, но только один слот `<slot name="item">`. Поэтому все такие `<li slot="item">` добавляются в `<slot name="item">` один за другим, формируя список.

Развёрнутое DOM-дерево становится таким:

```
<custom-menu>
  #shadow-root
    <style> /* стили меню */ </style>
    <div class="menu">
      <slot name="title">
        <span slot="title">Сладости</span>
      </slot>
      <ul>
        <slot name="item">
          <li slot="item">Леденцы</li>
          <li slot="item">Фруктовые тосты</li>
          <li slot="item">Кексы</li>
        </slot>
      </ul>
    </div>
</custom-menu>
```

Можно заметить, что в валидном DOM-дереве тег `<li>` должен быть прямым потомком тега `<ul>`. Но это развёрнутый DOM, который описывает то, как компонент отображается, в нём такая ситуация нормальна.

Осталось только добавить обработчик `click` для открытия и закрытия списка, и меню `<custom-menu>` готово:

```
customElements.define('custom-menu', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});

    // tmpl -- шаблон для теневого DOM-дерева (выше)
    this.shadowRoot.append( tmpl.content.cloneNode(true) );

    // мы не можем выбирать узлы светлого DOM, поэтому обработаем клики на слоте
    this.shadowRoot.querySelector('slot[name="title"]').onclick = () => {
      // открыть/закрыть меню
      this.shadowRoot.querySelector('.menu').classList.toggle('closed');
    };
  }
});
```

Вот полное демо:

Сладости
- Леденцы
- Фруктовые тосты
- Кексы

Конечно, мы можем расширить функционал меню, добавив события, методы и т.д.

Обновление слотов

Что если внешний код хочет динамически добавить или удалить пункты меню?

Браузер наблюдает за слотами и обновляет отображение при добавлении и удалении элементов в слотах.

Также, поскольку узлы светлого DOM-дерева не копируются, а только отображаются в слотах, изменения внутри них сразу же становятся видны.

Таким образом, нам ничего не нужно делать для обновления отображения. Но если код компонента хочет узнать об изменениях в слотах, можно использовать событие `slotchange`.

Например, здесь пункт меню вставляется динамически через 1 секунду, и заголовок меняется через 2 секунды:

```
<custom-menu id="menu">
  <span slot="title">Сладости</span>
</custom-menu>

<script>
customElements.define('custom-menu', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `<div class="menu">
      <slot name="title"></slot>
      <ul><slot name="item"></slot></ul>
    </div>`;

    // shadowRoot не может иметь обработчиков событий, поэтому используется первый
    this.shadowRoot.firstElementChild.addEventListener('slotchange',
      e => alert("slotchange: " + e.target.name)
    );
  }
});

setTimeout(() => {
  menu.insertAdjacentHTML('beforeEnd', '<li slot="item">Леденцы</li>')
}, 1000);

setTimeout(() => {
  menu.querySelector('[slot="title"]').innerHTML = "Новое меню";
}, 2000);
</script>
```

Отображение меню обновляется каждый раз без нашего вмешательства.

Здесь есть два события `slotchange`:

1. При инициализации:

   `slotchange: title` запускается сразу же, как только `slot="title"` из обычного дерева попадает в соответствующий слот.

2. Через 1 секунду:

   `slotchange: item` запускается, когда добавляется новый элемент `<li slot="item">`.

Обратите внимание, что событие `slotchange` не запускается через 2 секунды, когда меняется контент `slot="title"`. Это происходит потому, что сам слот не меняется. Мы изменяем содержимое элемента, который находится в слоте, а это совсем другое.

Если мы хотим отслеживать внутренние изменения обычного DOM-дерева из JavaScript, можно также использовать более обобщённый механизм: MutationObserver.

API слотов

И, наконец, давайте поговорим о методах JavaScript, связанных со слотами.

Как мы видели раньше, JavaScript смотрит на «реальный», а не на развёрнутый DOM. Но если у теневого дерева стоит `{mode: 'open'}`, то мы можем выяснить, какие элементы находятся в слоте, и, наоборот, определить слот по элементу, который в нём находится:

- `node.assignedSlot` – возвращает элемент `<slot>`, в котором находится `node`.
- `slot.assignedNodes({flatten: true/false})` – DOM-узлы, которые находятся в слоте. Опция `flatten` имеет значение по умолчанию `false`. Если явно изменить значение на `true`, она просматривает развёрнутый DOM глубже и возращает вложенные слоты, если есть вложенные компоненты, и резервный контент, если в слоте нет узлов.
- `slot.assignedElements({flatten: true/false})` – DOM-элементы, которые находятся в слоте (то же самое, что выше, но только узлы-элементы).

Эти методы можно использовать не только для отображения содержимого, которое находится в слотах, но и для его отслеживания в JavaScript.

Например, если компонент `<custom-menu>` хочет знать, что он показывает, он может отследить событие `slotchange` и получить пункты меню из `slot.assignedElements`:

```
<custom-menu id="menu">
  <span slot="title">Сладости</span>
  <li slot="item">Леденцы</li>
  <li slot="item">Фруктовые тосты</li>
</custom-menu>

<script>
customElements.define('custom-menu', class extends HTMLElement {
  items = []

  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `<div class="menu">
      <slot name="title"></slot>
      <ul><slot name="item"></slot></ul>
    </div>`;
```

```
    // слотовый элемент добавляется/удаляется/заменяется
    this.shadowRoot.firstElementChild.addEventListener('slotchange', e => {
      let slot = e.target;
      if (slot.name == 'item') {
        this.items = slot.assignedElements().map(elem => elem.textContent);
        alert("Items: " + this.items);
      }
    });
  }
});

// пункты меню обновятся через 1 секунду
setTimeout(() => {
  menu.insertAdjacentHTML('beforeEnd', '<li slot="item">Кексы</li>')
}, 1000);
</script>
```

Итого

Обычно, если у элемента есть теневое дерево, то содержимое обычного, светлого DOM не показывается. Слоты позволяют показать элементы светлого DOM на заданных местах в теневом DOM.

Существует два вида слотов:

- Именованные слоты: `<slot name="X">...</slot>` – получают элементы светлого DOM с `slot="X"`.
- Слот по умолчанию: первый `<slot>` без имени (последующие неименованные слоты игнорируются) – показывает элементы элементов светлого дерева, которые не находятся в других слотах.
- Если одному слоту назначено несколько элементов, они добавляются один за другим.
- Содержимое элемента `<slot>` используется как резервное. Оно отображается, если в слоте нет элементов из светлого дерева.

Процесс отображения элементов внутри слота называется «композицией». В результате композиции строится «развёрнутый DOM».

При композиции не происходит перемещения узлов – с точки зрения JavaScript, DOM остаётся прежним.

JavaScript может получить доступ к слотам с помощью следующих методов:

- `slot.assignedNodes/Elements()` – возвращает узлы/элементы, которые находятся внутри `slot`.
- `node.assignedSlot` – обратный метод, возвращает слот по узлу.

Если мы хотим знать, что показываем, мы можем отследить контент слота следующими способами:

- событие `slotchange` – запускается, когда слот наполняется контентом в первый раз, и при каждой операции добавления/удаления/замещения элемента в слоте, за исключением его потомков. Сам слот будет `event.target`.
- MutationObserver для более глубокого просмотра содержимого элемента в слоте и отслеживания изменений в нём.

Теперь, когда мы научились показывать элементы светлого DOM в теневом DOM, давайте посмотрим, как их правильно стилизовать. Основное правило звучит так: теневые элементы стилизуются внутри, а обычные элементы – снаружи; однако есть заметные исключения.

Мы рассмотрим их подробно в следующей главе.

Настройка стилей теневого DOM

Теневой DOM может содержать теги `<style>` и `<link rel="stylesheet" href="…">`. В последнем случае таблицы стилей кешируются по протоколу HTTP, так что они не будут загружаться повторно при использовании одного шаблона для многих компонентов.

Как правило, локальные стили работают только внутри теневого DOM, а стили документа – вне его. Но есть несколько исключений.

:host

Селектор `:host` позволяет выбрать элемент-хозяин (элемент, содержащий теневое дерево).

Например, мы создаём элемент `<custom-dialog>` который нужно расположить по-центру. Для этого нам необходимо стилизовать сам элемент `<custom-dialog>`.

Это именно то, что делает `:host`:

```
<template id="tmpl">
  <style>
    /* стиль будет применён изнутри к элементу <custom-dialog> */
    :host {
      position: fixed;
      left: 50%;
      top: 50%;
      transform: translate(-50%, -50%);
      display: inline-block;
```

```
      border: 1px solid red;
      padding: 10px;
    }
  </style>
  <slot></slot>
</template>

<script>
customElements.define('custom-dialog', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'}).append(tmpl.content.cloneNode(true));
  }
});
</script>

<custom-dialog>
  Hello!
</custom-dialog>
```

Каскадирование

Элемент-хозяин (элемент `<custom-dialog>`) находится в светлом DOM, поэтому к нему применяются CSS-стили документа.

Если есть некоторое свойство, стилизованное как в `:host` локально, так и в документе, то стиль документа будет приоритетным.

Например, если в документе из примера поставить:

```
<style>
custom-dialog {
  padding: 0;
}
</style>
```

...то `<custom-dialog>` будет без `padding`.

Это очень удобно, поскольку мы можем задать стили «по умолчанию» в компонента в его правиле `:host`, а затем, при желании, легко переопределить их в документе.

Исключение составляет тот случай, когда локальное свойство помечено как `!important`, для таких свойств приоритет имеют локальные стили.

:host(selector)

То же, что и `:host`, но применяется только в случае, если элемент-хозяин подходит под селектор `selector`.

Например, мы бы хотели выровнять по центру `<custom-dialog>`, только если он содержит атрибут `centered`:

```
<template id="tmpl">
  <style>
    :host([centered]) {
      position: fixed;
      left: 50%;
      top: 50%;
      transform: translate(-50%, -50%);
      border-color: blue;
    }

    :host {
      display: inline-block;
      border: 1px solid red;
      padding: 10px;
    }
  </style>
  <slot></slot>
</template>

<script>
customElements.define('custom-dialog', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'}).append(tmpl.content.cloneNode(true));
  }
});
</script>


<custom-dialog centered>
  Centered!
</custom-dialog>

<custom-dialog>
  Not centered.
</custom-dialog>
```

Теперь дополнительные стили для выравнивания по центру применяются только к первому элементу: `<custom-dialog centered>`.

:host-context(selector)

То же самое, что и `:host`, но применяется только в том случае, если элемент-хозяин или любой из его предков во внешнем документе подходит под селектор `selector`.

Например: правила в `:host-context(.dark-theme)` применятся, только если на элементе `<custom-dialog>` или где-то выше есть класс `dark-theme`:

```
<body class="dark-theme">
  <!--
    :host-context(.dark-theme) применится к custom-dialog внутри .dark-theme
  -->
  <custom-dialog>...</custom-dialog>
</body>
```

Подводя итог, мы можем использовать семейство селекторов `:host` для стилизации основного элемента компонента в зависимости от контекста. Эти стили (если только не стоит !important) могут быть переопределены документом.

Применение стилей к содержимому слотов

Теперь давайте рассмотрим ситуацию со слотами.

Элементы слотов происходят из светлого DOM, поэтому они используют стили документа. Локальные стили не влияют на содержимое слотов.

В примере ниже текст в `<span>` жирный в соответствии со стилями документа, но не берёт `background` из локальных стилей:

```
<style>
  span { font-weight: bold }
</style>

<user-card>
  <div slot="username"><span>John Smith</span></div>
</user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `
      <style>
      span { background: red; }
```

```
      </style>
      Имя: <slot name="username"></slot>
    `;
  }
});
</script>
```

Имя:
John Smith

В результате текст жирный, но не красный.

Если мы хотим стилизовать слотовые элементы в нашем компоненте, то есть два варианта.

Первое – можно стилизовать сам `<slot>` и полагаться на наследование CSS:

```
<user-card>
  <div slot="username"><span>John Smith</span></div>
</user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `
      <style>
      slot[name="username"] { font-weight: bold; }
      </style>
      Имя: <slot name="username"></slot>
    `;
  }
});
</script>
```

Имя:
John Smith

Здесь `<p>John Smith</p>` выделяется жирным шрифтом, потому что наследование CSS действует между `<slot>` и его содержимым. Но в CSS как таковом не все свойства наследуются.

Другой вариант – использовать псевдокласс `::slotted(селектор)`. Соответствует элементам, если выполняются два условия:

1. Это слотовый элемент, пришедший из светлого DOM. Имя слота не имеет значения. Просто любой элемент, вставленный в `<slot>`, но только сам

элемент, а не его потомки.

2. Элемент соответствует `селектору`.

В нашем примере `::slotted(div)` выбирает в точности `<div slot="username">`, но не его дочерние элементы:

```
<user-card>
  <div slot="username">
    <div>John Smith</div>
  </div>
</user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `
      <style>
      ::slotted(div) { border: 1px solid red; }
      </style>
      Name: <slot name="username"></slot>
    `;
  }
});
</script>
```

Name:
John Smith

Обратите внимание, что селектор `::slotted` не может спускаться дальше в слот. Эти селекторы недействительны:

```
::slotted(div span) {
  /* наш слот <div> не соответствует этому */
}

::slotted(div) p {
  /* не может войти в светлый DOM */
}
```

Кроме того, `::slotted` можно использовать только в CSS. Мы не можем использовать его в `querySelector`.

CSS-хуки с пользовательскими свойствами

Как можно стилизовать внутренние элементы компонента из основного документа?

Селекторы типа `:host` применяют правила к элементу `<custom-dialog>` или `<user-card>`, но как стилизовать элементы теневого DOM внутри них? Например, в `<user-card>` мы хотели бы разрешить внешнему документу изменять внешний вид пользовательских полей.

Аналогично тому, как мы предусматриваем у компонента методы, чтобы взаимодействовать с ним, мы можем использовать переменные CSS (пользовательские свойства CSS) для его стилизации.

Пользовательские свойства CSS существуют одновременно на всех уровнях, как светлом, так и в тёмном DOM.

Например, в теневом DOM мы можем использовать CSS-переменную `--user-card-field-color` для стилизации полей, а документ будет её устанавливать:

```
<style>
  .field {
    color: var(--user-card-field-color, black);
    /* если переменная --user-card-field-color не определена, будет использован цве
  }
</style>
<div class="field">Имя: <slot name="username"></slot></div>
<div class="field">Дата рождения: <slot name="birthday"></slot></div>
</style>
```

Затем мы можем объявить это свойство во внешнем документе для `<user-card>`:

```
user-card {
  --user-card-field-color: green;
}
```

Пользовательские CSS свойства проникают через теневой DOM, они видны повсюду, поэтому внутреннее правило `.field` будет использовать его.

Вот полный пример::

```
<style>
  user-card {
    --user-card-field-color: green;
  }
</style>
```

```
<template id="tmpl">
  <style>
    .field {
      color: var(--user-card-field-color, black);
    }
  </style>
  <div class="field">Имя: <slot name="username"></slot></div>
  <div class="field">Дата рождения: <slot name="birthday"></slot></div>
</template>

<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.append(document.getElementById('tmpl').content.cloneNode(true))
  }
});
</script>

<user-card>
  <span slot="username">John Smith</span>
  <span slot="birthday">01.01.2001</span>
</user-card>
```

Имя: John Smith
Дата рождения: 01.01.2001

Итого

Теневой DOM может включать в себя стили, такие как `<style>` или `<link rel="stylesheet">`.

Локальные стили могут влиять на:

- теневое дерево,
- элемент-хозяин, при помощи псевдоклассов семейства `:host`,
- слотовые элементы (из светлого DOM), `::slotted(селектор)` позволяет стилизовать сами слотовые элементы, но не их дочерние элементы.

Стили документов могут влиять на:

- элемент-хозяин (так как он находится во внешнем документе)
- слотовые элементы и их содержимое (так как они также физически присутствуют во внешнем документе)

Когда свойства CSS конфликтуют, обычно стили документа имеют приоритет, если только свойство не помечено как `!important`. Тогда предпочтение

отдаётся локальным стилям.

Пользовательские свойства CSS проникают через теневой DOM. Они используются как «хуки» для придания элементам стиля:

1. Компонент использует пользовательское CSS-свойство для стилизации ключевых элементов, например `var(--component-name-title, <значение по умолчанию>)`.
2. Автор компонента публикует эти свойства для разработчиков, они так же важны, как и другие общедоступные методы компонента.
3. Когда разработчик хочет стилизовать заголовок, он назначает CSS-свойство `--component-name-title` для элемента-хозяина или выше.
4. Profit!

Теневой DOM и события

Смысл создания теневого DOM-дерева – это инкапсуляция внутренних деталей компонента.

Допустим, клик произошёл внутри теневого DOM на компоненте `<user-card>`. Но скрипты основного документа ничего не знают о внутреннем устройстве теневой DOM-структуры, в особенности, если компонент создан сторонней библиотекой.

Поэтому, чтобы не нарушать инкапсуляцию, браузер *меняет у этого события целевой элемент*.

События, которые произошли в теневом DOM, но пойманы снаружи этого DOM, имеют элемент-хозяин в качестве целевого элемента `event.target`.

Рассмотрим простой пример:

```
<user-card></user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `<p>
      <button>Нажми меня</button>
    </p>`;
    this.shadowRoot.firstElementChild.onclick =
      e => alert("Внутренний целевой элемент: " + e.target.tagName);
  }
});

document.onclick =
```

```
  e => alert("Внешний целевой элемент: " + e.target.tagName);
</script>
```

Нажми меня

Если нажать на кнопку, то выведется следующее:

1. Внутренний целевой элемент: `BUTTON` – внутренний обработчик событий получает правильный целевой элемент – элемент, находящийся внутри теневого DOM.
2. Внешний целевой элемент: `USER-CARD` – обработчик событий на уровне документа получает элемент-хозяин в качестве целевого.

Хорошо, что браузер подменяет целевые элементы событий. Потому что внешний документ ничего не знает о внутреннем устройстве компонента. С его (внешнего документа) точки зрения, событие происходит на `<user-card>`.

Подмена целевого элемента не происходит, если событие берёт начало на элементе из слота, который фактически находится в обычном, светлом DOM.

Например, если пользователь кликнет на `<span slot="username">` в примере ниже – целевой элемент события будет именно этот `span` для обоих обработчиков – теневого и обычного (светлого):

```
<user-card id="userCard">
  <span slot="username">John Smith</span>
</user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `<div>
      <b>Имя:</b> <slot name="username"></slot>
    </div>`;

    this.shadowRoot.firstElementChild.onclick =
      e => alert("Внутренний целевой элемент: " + e.target.tagName);
  }
});

userCard.onclick = e => alert(`Внешний целевой элемент: ${e.target.tagName}`);
</script>
```

Имя: John Smith

Если клик произойдёт на `"John Smith"`, то для обоих обработчиков – внутреннего и внешнего – целевым элементом будет `<span slot="username">`. Это элемент обычного (светлого) DOM, так что подмены не происходит.

С другой стороны, если клик произойдёт на элементе, который находится в теневом DOM, например, на `<b>Имя</b>`, то как только всплытие выйдет за пределы теневой DOM-структуры, его `event.target` станет `<user-card>`.

Всплытие и метод event.composedPath()

Для обеспечения всплытия событий используется развёрнутый DOM.

Таким образом, если у нас есть элемент в слоте, и событие происходит где-то внутри него, то оно всплывает до `<slot>` и выше.

Полный путь к изначальному целевому элементу, со всеми теневыми элементами, можно получить, воспользовавшись методом `event.composedPath()`. Как видно из названия, этот метод возвращает путь после композиции.

В примере выше развёрнутое DOM-дерево будет таким:

```
<user-card id="userCard">
  #shadow-root
    <div>
      <b>Имя:</b>
      <slot name="username">
        <span slot="username">John Smith</span>
      </slot>
    </div>
</user-card>
```

Так что, при клике по `<span slot="username">` вызов метода `event.composedPath()` вернёт массив: [`span`, `slot`, `div`, `shadow-root`, `user-card`, `body`, `html`, `document`, `window`]. Что в точности отражает цепочку родителей от целевого элемента в развёрнутой DOM-структуре после композиции.

⚠️ **Детали теневого DOM-дерева доступны только для деревьев с `{mode:'open'}`**

Если теневое DOM-дерево было создано с `{mode: 'closed'}`, то после композиции путь будет начинаться с элемента-хозяина: `user-card` и дальше вверх по дереву.

Этот метод следует тем же принципам, что и остальные. Внутреннее устройство закрытых DOM-деревьев совершенно скрыто.

Свойство: event.composed

Большинство событий успешно всплывают сквозь границу теневого DOM. Но не все.

Это поведение регулируется с помощью свойства `composed` объекта события. Если оно `true`, то событие пересекает границу. Иначе, оно может быть поймано лишь внутри теневого DOM.

Если посмотреть в спецификацию UI Events, то большинство событий имеют `composed: true`:

- `blur`, `focus`, `focusin`, `focusout`,
- `click`, `dblclick`,
- `mousedown`, `mouseup` `mousemove`, `mouseout`, `mouseover`,
- `wheel`,
- `beforeinput`, `input`, `keydown`, `keyup`.

Все события курсора и сенсорные события также имеют `composed: true`.

Хотя есть и события, имеющие `composed: false`:

- `mouseenter`, `mouseleave` (они вообще не всплывают),
- `load`, `unload`, `abort`, `error`,
- `select`,
- `slotchange`.

Эти события могут быть пойманы только на элементах того же DOM, в котором находится целевой элемент события.

Генерация событий

Когда мы генерируем своё событие, то, чтобы оно всплывало за пределы компонента, нужно установить оба свойства: `bubbles` и `composed` – в

значение `true`.

Например, здесь мы создаём элемент `div#inner` в теневом DOM-дереве элемента `div#outer` и генерируем на нём два события. Только одно с флагом `composed: true` выйдет наружу, в документ:

```
<div id="outer"></div>

<script>
outer.attachShadow({mode: 'open'});

let inner = document.createElement('div');
outer.shadowRoot.append(inner);

/*
div(id=outer)
  #shadow-dom
    div(id=inner)
*/

document.addEventListener('test', event => alert(event.detail));

inner.dispatchEvent(new CustomEvent('test', {
  bubbles: true,
  composed: true,
  detail: "composed"
}));

inner.dispatchEvent(new CustomEvent('test', {
  bubbles: true,
  composed: false,
  detail: "not composed"
}));
</script>
```

Итого

Только те события пересекают границы теневого DOM, у которых флаг `composed` установлен в значение `true`.

У большинства встроенных событий стоит `composed: true`, это описано в соответствующих спецификациях:

- UI Events https://www.w3.org/TR/uievents .
- Touch Events https://w3c.github.io/touch-events .
- Pointer Events https://www.w3.org/TR/pointerevents .
- …И так далее.

У некоторых встроенных событий всё же стоит `composed: false`:

- `mouseenter`, `mouseleave` (вообще не всплывают),
- `load`, `unload`, `abort`, `error`,
- `select`,
- `slotchange`.

Эти события могут быть пойманы только на элементах, принадлежащих тому же DOM-дереву.

Если мы генерируем своё событие `CustomEvent`, то должны явно поставить флаг `composed: true`.

Обратите внимание, что в случае вложенных компонентов теневые DOM могут быть вложены друг в друга. События с флагом `composed` всплывают через границы всех теневых DOM. Поэтому, если событие предназначено только для ближайшего внешнего компонента-родителя, мы можем инициировать его на элементе-хозяине и установить флаг `composed: false`. Тогда оно будет уже вне теневого DOM компонента, но не выплывает наружу в «ещё более внешний» DOM.

Регулярные выражения

Регулярные выражения – мощный способ поиска и замены для строк.

Введение: шаблоны и флаги

Регулярные выражения – мощное средство поиска и замены в строке.

В JavaScript регулярные выражения реализованы отдельным объектом RegExp и интегрированы в методы строк.

Регулярные выражения

Регулярное выражение (оно же «регэксп», «регулярка» или просто «рег»), состоит из *шаблона* (также говорят «паттерн») и необязательных *флагов*.

Существует два синтаксиса для создания регулярного выражения.

«Длинный» синтаксис:

```
regexp = new RegExp("шаблон", "флаги");
```

…И короткий синтаксис, использующий слеши `"/"`:

```
regexp = /шаблон/; // без флагов
regexp = /шаблон/gmi; // с флагами gmi (будут описаны далее)
```

Слеши `/.../` говорят JavaScript о том, что это регулярное выражение. Они играют здесь ту же роль, что и кавычки для обозначения строк.

Регулярное выражение `regexp` в обоих случаях является объектом встроенного класса `RegExp`.

Основная разница между этими двумя способами создания заключается в том, что слеши `/.../` не допускают никаких вставок переменных (наподобие возможных в строках через `${...}`). Они полностью статичны.

Слеши используются, когда мы на момент написания кода точно знаем, каким будет регулярное выражение – и это большинство ситуаций. А `new RegExp` – когда мы хотим создать регулярное выражение «на лету» из динамически сгенерированной строки, например:

```
let tag = prompt("Какой тег вы хотите найти?", "h2");

let regexp = new RegExp(`<${tag}>`); // то же, что /<h2>/  при ответе "h2" на promp
```

Флаги

Регулярные выражения могут иметь флаги, которые влияют на поиск.

В JavaScript их всего шесть:

`i`

С этим флагом поиск не зависит от регистра: нет разницы между `A` и `a` (см. пример ниже).

`g`

С этим флагом поиск ищет все совпадения, без него – только первое.

`m`

Многострочный режим (рассматривается в главе Многострочный режим якорей ^ $, флаг "m").

`s`

Включает режим «dotall», при котором точка `.` может соответствовать символу перевода строки `\n` (рассматривается в главе Символьные классы).

`u`

Включает полную поддержку юникода. Флаг разрешает корректную обработку суррогатных пар (подробнее об этом в главе Юникод: флаг "u" и класс \p{...}).

`y`

Режим поиска на конкретной позиции в тексте (описан в главе Поиск на заданной позиции, флаг "y")

Цветовые обозначения

Здесь и далее в тексте используется следующая цветовая схема:

- регулярное выражение – `красный`
- строка (там где происходит поиск) – `синий`
- результат – `зелёный`

Поиск: str.match

Как уже говорилось, использование регулярных выражений интегрировано в методы строк.

Метод `str.match(regexp)` для строки `str` возвращает совпадения с регулярным выражением `regexp`.

У него есть три режима работы:

1. Если у регулярного выражения есть флаг `g`, то он возвращает массив всех совпадений:

```
let str = "Любо, братцы, любо!";

alert( str.match(/любо/gi) ); // Любо,любо (массив из 2х подстрок-совпадений)
```

   Обратите внимание: найдены и `Любо` и `любо`, благодаря флагу `i`, который делает регулярное выражение регистро-независимым.

2. Если такого флага нет, то возвращает только первое совпадение в виде массива, в котором по индексу `0` находится совпадение, и есть свойства с дополнительной информацией о нём:

```
let str = "Любо, братцы, любо!";

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

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

// Дополнительная информация:
alert( result.index );  // 0 (позиция совпадения)
alert( result.input );  // Любо, братцы, любо! (исходная строка)
```

В этом массиве могут быть и другие индексы, кроме `0`, если часть регулярного выражения выделена в скобки. Мы разберём это в главе Скобочные группы.

3. И, наконец, если совпадений нет, то, вне зависимости от наличия флага `g`, возвращается `null`.

   Это очень важный нюанс. При отсутствии совпадений возвращается не пустой массив, а именно `null`. Если об этом забыть, можно легко допустить ошибку, например:

```
let matches = "JavaScript".match(/HTML/); // = null

if (!matches.length) { // Ошибка: у null нет свойства length
  alert("Ошибка в строке выше");
}
```

   Если хочется, чтобы результатом всегда был массив, можно написать так:

```
let matches = "JavaScript".match(/HTML/) || [];

if (!matches.length) {
  alert("Совпадений нет"); // теперь работает
}
```

Замена: str.replace

Метод `str.replace(regexp, replacement)` заменяет совпадения с `regexp` в строке `str` на `replacement` (все, если есть флаг `g`, иначе только первое).

Например:

```
// без флага g
alert( "We will, we will".replace(/we/i, "I") ); // I will, we will

// с флагом g
alert( "We will, we will".replace(/we/ig, "I") ); // I will, I will
```

В строке замены `replacement` мы можем использовать специальные комбинации символов для вставки фрагментов совпадения:

Спецсимволы	Действие в строке замены
`$&`	вставляет всё найденное совпадение
`$``	вставляет часть строки до совпадения
`$'`	вставляет часть строки после совпадения
`$n`	если `n` это 1-2 значное число, вставляет содержимое n-й скобочной группы регулярного выражения, больше об этом в главе Скобочные группы
`$<name>`	вставляет содержимое скобочной группы с именем `name`, также изучим в главе Скобочные группы
`$$`	вставляет символ `"$"`

Пример с `$&`:

```
alert( "Люблю HTML".replace(/HTML/, "$& и JavaScript") ); // Люблю HTML и JavaScrip
```

Проверка: regexp.test

Метод `regexp.test(str)` проверяет, есть ли хоть одно совпадение, если да, то возвращает `true`, иначе `false`.

```
let str = "Я ЛюБлЮ JavaScript";
let regexp = /люблю/i;

alert( regexp.test(str) ); // true
```

Далее в этом разделе мы будем изучать регулярные выражения, увидим ещё много примеров их использования, а также познакомимся с другими методами.

Полная информация о различных методах дана в главе Методы RegExp и String.

Итого

- Регулярное выражение состоит из шаблона и необязательных флагов: `g`, `i`, `m`, `u`, `s`, `y`.
- Без флагов и специальных символов, которые мы изучим позже, поиск по регулярному выражению аналогичен поиску подстроки.

- Метод `str.match(regexp)` ищет совпадения: все, если есть флаг `g`, иначе только первое.
- Метод `str.replace(regexp, replacement)` заменяет совпадения с `regexp` на `replacement`: все, если у регулярного выражения есть флаг `g`, иначе только первое.
- Метод `regexp.test(str)` возвращает `true`, если есть хоть одно совпадение, иначе `false`.

Символьные классы

Рассмотрим практическую задачу – у нас есть номер телефона вида `"+7(903)-123-45-67"`, и нам нужно превратить его в строку только из чисел: `79035419441`.

Для этого мы можем найти и удалить все, что не является числом. С этим нам помогут символьные классы.

Символьный класс – это специальное обозначение, которое соответствует любому символу из определённого набора.

Для начала давайте рассмотрим класс «цифра». Он обозначается как `\d` и в регулярном выражении соответствует «любой одной цифре».

Например, давайте найдём первую цифру в номере телефона:

```
let str = "+7(903)-123-45-67";

let regexp = /\d/;

alert( str.match(regexp) ); // 7
```

Без флага `g` регулярное выражение ищет только первое совпадение, то есть первую цифру `\d`.

Давайте добавим флаг `g`, чтобы найти все цифры:

```
let str = "+7(903)-123-45-67";

let regexp = /\d/g;

alert( str.match(regexp) ); // массив совпадений: 7,9,0,3,1,2,3,4,5,6,7

// и можно сделать из них уже чисто цифровой номер телефона
alert( str.match(regexp).join('') ); // 79035419441
```

Это был символьный класс для цифр. Есть и другие символьные классы.

Наиболее используемые:

`\d` («d» от английского «digit» означает «цифра»)

Цифра: символ от `0` до `9`.

`\s` («s»: от английского «space» – «пробел»)

Пробельные символы: включает в себя символ пробела, табуляции `\t`, перевода строки `\n` и некоторые другие редкие пробельные символы, обозначаемые как `\v`, `\f` и `\r`.

`\w` («w»: от английского «word» – «слово»)

Символ «слова», а точнее – буква латинского алфавита или цифра или подчёркивание `_`. Нелатинские буквы не являются частью класса `\w`, то есть буква русского алфавита не подходит.

Для примера, `\d\s\w` обозначает «цифру», за которой идёт пробельный символ, а затем символ слова, например `1 a`.

Регулярное выражение может содержать как обычные символы, так и символьные классы.

Например, `CSS\d` соответствует строке `CSS` с цифрой после неё:

```
let str = "Есть ли стандарт CSS4?";
let regexp = /CSS\d/

alert( str.match(regexp) ); // CSS4
```

Также мы можем использовать несколько символьных классов:

```
alert( "I love HTML5!".match(/\s\w\w\w\w\d/) ); // ' HTML5'
```

Соответствие (каждому символьному классу соответствует один символ результата):

Обратные символьные классы

Для каждого символьного класса существует «обратный класс», обозначаемый той же буквой, но в верхнем регистре.

«Обратный» означает, что он соответствует всем другим символам, например:

`\D`

Не цифра: любой символ, кроме `\d`, например буква.

`\S`

Не пробел: любой символ, кроме `\s`, например буква.

`\W`

Любой символ, кроме `\w`, то есть не буквы из латиницы, не знак подчёркивания и не цифра. В частности, русские буквы принадлежат этому классу.

Мы уже видели, как сделать чисто цифровой номер из строки вида `+7(903)-123-45-67`: найти все цифры и соединить их.

```
let str = "+7(903)-123-45-67";

alert( str.match(/\d/g).join('') ); // 79031234567
```

Альтернативный, более короткий путь – найти нецифровые символы `\D` и удалить их из строки:

```
let str = "+7(903)-123-45-67";

alert( str.replace(/\D/g, "") ); // 79031234567
```

Точка – это любой символ

Точка `.` – это специальный символьный класс, который соответствует «любому символу, кроме новой строки».

Для примера:

```
alert( "Ю".match(/./) ); // Ю
```

Или в середине регулярного выражения:

```
let regexp = /CS.4/;

alert( "CSS4".match(regexp) ); // CSS4
alert( "CS-4".match(regexp) ); // CS-4
alert( "CS 4".match(regexp) ); // CS 4 (пробел тоже является символом)
```

Обратите внимание, что точка означает «любой символ», но не «отсутствие символа». Там должен быть какой-либо символ, чтобы соответствовать условию поиска:

```
alert( "CS4".match(/CS.4/) ); // null, нет совпадений потому что нет символа для то
```

Точка как буквально любой символ, с флагом «s»

Обычно точка не соответствует символу новой строки `\n`.

То есть, регулярное выражение `A.B` будет искать символ `A` и затем `B`, с любым символом между ними, кроме перевода строки `\n`:

```
alert( "A\nB".match(/A.B/) ); // null (нет совпадения)
```

Но во многих ситуациях точкой мы хотим обозначить действительно «любой символ», включая перевод строки.

Как раз для этого нужен флаг `s`. Если регулярное выражение имеет его, то точка `.` соответствует буквально любому символу:

```
alert( "A\nB".match(/A.B/s) ); // A\nB (совпадение!)
```

Внимание, пробелы!

Обычно мы уделяем мало внимания пробелам. Для нас строки `1-5` и `1 - 5` практически идентичны.

Но если регулярное выражение не учитывает пробелы, оно может не сработать.

Давайте попробуем найти цифры, разделённые дефисом:

```
alert( "1 - 5".match(/\d-\d/) ); // null, нет совпадения!
```

Исправим это, добавив пробелы в регулярное выражение `\d - \d`:

```
alert( "1 - 5".match(/\d - \d/) ); // 1 - 5, теперь работает
// или можно использовать класс \s:
alert( "1 - 5".match(/\d - \d/) ); // 1 - 5, тоже работает
```

Пробел – это символ. Такой же важный, как любой другой.

Нельзя просто добавить или удалить пробелы из регулярного выражения, и ожидать, что оно будет также работать.

Другими словами, в регулярном выражении все символы имеют значение, даже пробелы.

Итого

Существуют следующие символьные классы:

- `\d` – цифры.
- `\D` – не цифры.
- `\s` – пробельные символы, табы, новые строки.
- `\S` – все, кроме `\s`.
- `\w` – латиница, цифры, подчёркивание `'_'`.
- `\W` – все, кроме `\w`.
- `.` – любой символ, если с флагом регулярного выражения `s`, в противном случае любой символ, кроме перевода строки `\n`.

…Но это не всё!

В кодировке Юникод, которую JavaScript использует для строк, каждому символу соответствует ряд свойств, например – какого языка это буква (если

буква), является ли символ знаком пунктуации, и т.п.

Можно искать, в том числе, и по этим свойствам. Для этого нужен флаг `u`, который мы рассмотрим в следующей главе.

Юникод: флаг "u" и класс \p{...}

В JavaScript для строк используется кодировка Юникод . Обычно символы кодируются с помощью 2 байтов, что позволяет закодировать максимум 65536 символов.

Этого диапазона не хватает для того, чтобы закодировать все символы. Поэтому некоторые редкие символы кодируются с помощью 4 байтов, например `𝒳` (математический X) или `😄` (смайлик), некоторые иероглифы, и т.п.

В таблице ниже приведены юникоды нескольких символов:

Символ	Юникод	Количество байт в юникоде
a	`0x0061`	2
≈	`0x2248`	2
𝒳	`0x1d4b3`	4
𝒴	`0x1d4b4`	4
😄	`0x1f604`	4

Таким образом, символы типа `a` и `≈` занимают по 2 байта, а коды для `𝒳`, `𝒴` и `😄` – длиннее, в них 4 байта.

Когда-то давно, на момент создания языка JavaScript, кодирока Юникод была проще: символов в 4 байта не существовало. И, хотя это время давно прошло, многие строковые функции всё ещё могут работать некорректно.

Например, свойство `length` считает, что здесь два символа:

```
alert('😄'.length); // 2
alert('𝒳'.length); // 2
```

…Но мы видим, что только один, верно? Дело в том, что свойство `length` воспринимает 4-байтовый символ как два символа по 2 байта. Это неверно, потому что эти два символа должны восприниматься как единое целое (так называемая «суррогатная пара», вы также можете прочитать об этом в главе Строки).

Регулярные выражения также по умолчанию воспринимают 4-байтные «длинные символы» как пары 2-байтных. Как и со строками, это может приводить к странным результатам. Мы увидим примеры чуть позже, в главе Наборы и диапазоны [...].

В отличие от строк, у регулярных выражений есть специальный флаг `u`, который исправляет эту проблему. При его наличии регулярное выражение работает с 4-байтными символами правильно. И, кроме того, становится доступным поиск по юникодным свойствам, который мы рассмотрим далее.

Юникодные свойства \p{...}

> **Не поддерживается в Firefox и Edge**
>
> Несмотря на то, что это часть стандарта с 2018 года, юникодные свойства не поддерживаются в Firefox (задача) и Edge (задача).
>
> Существует библиотека XRegExp , которая реализует «расширенные» регулярные выражения с кросс-браузерной поддержкой юникодных свойств.

Каждому символу в кодировке Юникод соответствует множество свойств. Они описывают к какой «категории» относится символ, содержат различную информацию о нём.

Например, свойство `Letter` у символа означает, что это буква какого-то алфавита, причём любого. А свойство `Number` означает, что это цифра – арабская или китайская, и т.п, на каком-то из языков.

В регулярном выражении можно искать символы с заданным свойствам, указав его в `\p{…}`. Для таких регулярных выражений обязательно использовать флаг `u`.

Например, `\p{Letter}` обозначает букву в любом языке. Также можно использовать запись `\p{L}`, так как `L` – это псевдоним `Letter`. Существуют короткие записи почти для всех свойств.

В примере ниже будут найдены английская, грузинская и корейская буквы:

```
let str = "A ბ 가";

alert( str.match(/\p{L}/gu) ); // A,ბ,가
alert( str.match(/\p{L}/g) ); // null (ничего не нашло, так как нет флага "u")
```

Вот основные категории символов и их подкатегории:

- Буквы `L`:
  - в нижнем регистре `Ll`,
  - модификаторы `Lm`,
  - заглавные буквы `Lt`,
  - в верхнем регистре `Lu`,
  - прочие `Lo`.
- Числа `N`:
  - десятичная цифра `Nd`,
  - цифры обозначаемые буквами (римские) `Nl`,
  - прочие `No`:
- Знаки пунктуации `P`:
  - соединители `Pc`,
  - тире `Pd`,
  - открывающие кавычки `Pi`,
  - закрывающие кавычки `Pf`,
  - открывающие скобки `Ps`,
  - закрывающие скобки `Pe`,
  - прочее `Po`.
- Отметки `M` (например, акценты):
  - двоеточия `Mc`,
  - вложения `Me`,
  - апострофы `Mn`.
- Символы `S`:
  - валюты `Sc`, модификаторы `Sk`, математические `Sm`, прочие `So`.
- Разделители `Z`:
  - линия `Zl`,
  - параграф `Zp`,
  - пробел `Zs`.
- Прочие `C`:
  - контрольные `Cc`,
  - форматирование `Cf`,
  - не назначенные `Cn`,
  - для приватного использования `Co`,
  - суррогаты `Cs`.

Так что, например, если нам нужны буквы в нижнем регистре, то можно написать `\p{Ll}`, знаки пунктуации: `\p{P}` и так далее.

Есть и другие категории – производные, например:

- `Alphabetic` (`Alpha`), включающая в себя буквы `L`, плюс «буквенные цифры» `Nl` (например XII – символ для римской записи числа 12), и некоторые другие символы `Other_Alphabetic` (`OAlpha`).
- `Hex_Digit` включает символы для шестнадцатеричных чисел: `0-9`, `a-f`.
- И так далее.

Юникод поддерживает много различных свойств, их полное перечисление потребовало бы очень много места, поэтому вот ссылки:

- По символу посмотреть его свойства: https://unicode.org/cldr/utility/character.jsp .
- По свойству посмотреть символы с ним: https://unicode.org/cldr/utility/list-unicodeset.jsp .
- Короткие псевдонимы для свойств: https://www.unicode.org/Public/UCD/latest/ucd/PropertyValueAliases.txt .
- Полная база Юникод-символов в текстовом формате вместе со всеми свойствами, находится здесь: https://www.unicode.org/Public/UCD/latest/ucd/ .

Пример: шестнадцатиричные числа

Например, давайте поищем шестнадцатеричные числа, записанные в формате `xFF`, где вместо `F` может быть любая шестнадцатиричная цифра (0…1 или A…F).

Шестнадцатеричная цифра может быть обозначена как `\p{Hex_Digit}`:

```
let regexp = /x\p{Hex_Digit}\p{Hex_Digit}/u;

alert("число: xAF".match(regexp)); // xAF
```

Пример: китайские иероглифы

Поищем китайские иероглифы.

В Юникоде есть свойство `Script` (система написания), которое может иметь значения `Cyrillic` (Кириллическая), `Greek` (Греческая), `Arabic` (Арабская), `Han` (Китайская) и так далее, здесь полный список.

Для поиска символов в нужной системе мы должны установить `Script=<значение>`, например для поиска кириллических букв: `\p{sc=Cyrillic}`, для китайских иероглифов: `\p{sc=Han}`, и так далее:

```
let regexp = /\p{sc=Han}/gu; // вернёт китайские иероглифы

let str = `Hello Привет 你好 123_456`;

alert( str.match(regexp) ); // 你,好
```

Пример: валюта

Символы, обозначающие валюты, такие как `$`, `€`, `¥` и другие, имеют свойство `\p{Currency_Symbol}`, короткая запись `\p{Sc}`.

Используем его, чтобы поискать цены в формате «валюта, за которой идёт цифра»:

```
let regexp = /\p{Sc}\d/gu;

let  str = `Цены: $2, €1, ¥9`;

alert( str.match(regexp) ); // $2,€1,¥9
```

Позже, в главе Квантификаторы +, *, ? и {n} мы изучим, как искать числа из любого количества цифр.

Итого

Флаг `u` включает поддержку Юникода в регулярных выражениях.

Конкретно, это означает, что:

1. Символы из 4 байт воспринимаются как единое целое, а не как два символа по 2 байта.
2. Работает поиск по юникодным свойствам `\p{…}`.

С помощью юникодных свойств мы можем искать слова на нужных языках, специальные символы (кавычки, обозначения валюты) и так далее.

Якоря: начало строки ^ и конец $

У символов каретки `^` и доллара `$` есть специальные значения в регулярных выражениях. Они называются «якоря» (anchors).

Каретка `^` означает совпадение с началом текста, а доллар `$` – с концом.

К примеру, давайте проверим начинается ли текст с `Mary`:

```
let str1 = "Mary had a little lamb";
alert( /^Mary/.test(str1) ); // true
```

Шаблон `^Mary` означает: «начало строки, затем Mary».

Аналогично можно проверить, кончается ли строка словом `snow` при помощи `snow$`:

```
let str1 = "it's fleece was white as snow";
alert( /snow$/.test(str1) ); // true
```

В конкретно этих случаях мы могли бы использовать и методы строк `startsWith/endsWith`. Регулярные выражения следует применять, когда нужна проверка сложнее.

Проверка на полное совпадение

Оба якоря вместе `^...$` часто используются для проверки, совпадает ли строка с шаблоном полностью. Например, чтобы определить, в правильном ли формате пользователь ввёл данные.

Проверим, что строка является временем в формате `12:34`, то есть две цифры, затем двоеточие, затем ещё две цифры.

На языке регулярных выражений это `\d\d:\d\d`:

```
let goodInput = "12:34";
let badInput = "12:345";

let regexp = /^\d\d:\d\d$/;
alert( regexp.test(goodInput) ); // true
alert( regexp.test(badInput) ); // false
```

Здесь совпадение с `\d\d:\d\d` ищется не где-то посередине текста, а сразу после начала строки `^`, и после него должен быть сразу конец строки `$`.

То есть, вся строка – это как раз время в нужном нам формате.

Поведение якорей меняется, если присутствует флаг `m`. Мы рассмотрим этот флаг в следующей статье.

"У

Якоря `^` и `$` – это проверки. У них них нулевая ширина.

Другими словами, они не добавляют к результату поиска символы, а только заставляют движок регулярных выражений проверять условие (начало/конец текста).

Задачи

Регулярное выражение ^$

Какая строка подойдёт под шаблон `^$`?

К решению

Многострочный режим якорей ^ $, флаг "m"

Многострочный режим включается флагом `m`.

Он влияет только на поведение `^` и `$`.

В многострочном режиме они означают не только начало/конец текста, но и начало/конец каждой строки в тексте.

Поиск в начале строки ^

В примере ниже текст состоит из нескольких строк. Шаблон `/^\d/gm` берёт цифру с начала каждой строки:

```
let str = `1е место: Винни
2е место: Пятачок
3е место: Слонопотам`;

alert( str.match(/^\d/gm) ); // 1, 2, 3
```

Обратим внимание – без флага `m` было бы найдено только первое число:

```
let str = `1е место: Винни
2е место: Пятачок
3е место: Слонопотам`;

alert( str.match(/^\d/g) ); // 1
```

Так происходит, потому что в обычном режиме каретка `^` – это только начало текста, а в многострочном – начало любой строки.

> **На заметку:**
>
> «Начало строки», формально, означает «сразу после перевода строки», то есть проверка `^` в многострочном режиме верна на всех позициях, которым предшествует символ перевода строки `\n`.
>
> И в начале текста.

Поиск в конце строки $

Символ доллара `$` ведёт себя аналогично.

Регулярное выражение `\d$` ищет последнюю цифру в каждой строке

```
let str = `Винни: 1
Пятачок: 2
Слонопотам: 3`;

alert( str.match(/\d$/gm) ); // 1,2,3
```

Без флага `m` якорь `$` обозначал бы конец всей строки, и была бы найдена только последняя цифра.

Ищем \n вместо ^ $

Для того, чтобы найти конец строки, можно использовать не только якоря `^` и `$`, но и символ перевода строки `\n`.

В чём разница? Давайте посмотрим на примере.

Поищем `\d\n` вместо `\d$`:

```
let str = `Винни: 1
Пятачок: 2
Слонопотам: 3`;

alert( str.match(/\d\n/gm) ); // 1\n,2\n
```

Как видим, совпадений теперь два, а не три.

Это потому, что после `3` нет перевода строки (а конец текста, подходящий под `$` – есть).

Ещё одно отличие: теперь в каждое найденное совпадение входит символ перевода строки `\n`. В отличие от якорей `^` `$`, которые только проверяют условие (начало/конец строки), `\n` – символ и входит в результат.

Так что `\n` в шаблоне используется, когда нам нужен сам символ перевода строки в результате, а якоря – когда хотим найти что-то в начале/конце строки.

Граница слова: \b

Граница слова `\b` – проверка, как `^` и `$`.

Когда движок регулярных выражений (программный модуль, реализующий поиск по регулярным выражениям) видит `\b`, он проверяет, что позиция в строке является границей слова.

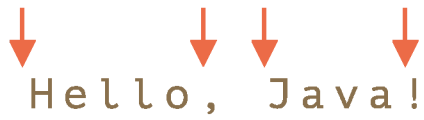
Есть три вида позиций, которые являются границами слова:

- Начало текста, если его первый символ `\w`.
- Позиция внутри текста, если слева находится `\w`, а справа – не `\w`, или наоборот.
- Конец текста, если его последний символ `\w`.

Например, регулярное выражение `\bJava\b` будет найдено в строке `Hello, Java!`, где `Java` – отдельное слово, но не будет найдено в строке `Hello, JavaScript!`.

```
alert( "Hello, Java!".match(/\bJava\b/) ); // Java
alert( "Hello, JavaScript!".match(/\bJava\b/) ); // null
```

В строке `Hello, Java!` следующие позиции соответствуют `\b`:



Так что она соответствует регулярному выражению `\bHello\b`, потому что:

1. В начале строки совпадает первая проверка `\b`.
2. Далее слово `Hello` совпадает.
3. Далее проверка `\b` – снова совпадает, так как мы находимся между `o` и пробелом.

Шаблон `\bJava\b` также совпадёт. Но не `\bHell\b` (потому что после `l` нет границы слова), и не `Java!\b` (восклицательный знак не является «символом слова» `\w`, поэтому после него нет границы слова).

```
alert( "Hello, Java!".match(/\bHello\b/) ); // Hello
alert( "Hello, Java!".match(/\bJava\b/) );  // Java
alert( "Hello, Java!".match(/\bHell\b/) );  // null (нет совпадения)
alert( "Hello, Java!".match(/\bJava!\b/) ); // null (нет совпадения)
```

Мы можем использовать `\b` не только со словами, но и с цифрами.

Например, регулярное выражение `\b\d\d\b` ищет отдельно стоящие двузначные числа. Другими словами, оно требует, чтобы и до и после `\d\d` были символы, отличные от `\w`, такие как побелы или пунктуация (или начало/конец текста).

```
alert( "1 23 456 78".match(/\b\d\d\b/g) ); // 23,78
alert( "12,34,56".match(/\b\d\d\b/g) ); // 12,34,56
```

Граница слова `\b` не работает для алфавитов, не основанных на латинице

Проверка границы слова `\b` проверяет границу, должно быть `\w` с одной стороны и "не `\w`" – с другой.

Но `\w` означает латинскую букву (или цифру или знак подчёркивания), поэтому проверка не будет работать для других символов, например, кириллицы или иероглифов).

Задачи

Найдите время

Время имеет формат: `часы:минуты`. И часы, и минуты имеют две цифры, например, `09:00`.

Введите регулярное выражение, чтобы найти время в строке: `Завтрак в 09:00 в комнате 123:456.`

P.S. В этой задаче пока нет необходимости проверять правильность времени, поэтому `25:99` также может быть верным результатом.

P.P.S. Регулярное выражение не должно находить `123:456`.

К решению

Экранирование, специальные символы

Как мы уже видели, обратная косая черта `\` используется для обозначения классов символов, например `\d`. Это специальный символ в регулярных выражениях (как и в обычных строках).

Есть и другие специальные символы, которые имеют особое значение в регулярном выражении. Они используются для более сложных поисковых конструкций. Вот полный перечень этих символов: `[ \ ^ $ . | ? * + ( )`.

Не надо пытаться запомнить этот список: мы разберёмся с каждым из них по отдельности, и таким образом вы выучите их «автоматически».

Экранирование символов

Допустим, мы хотим найти буквально точку. Не «любой символ», а именно точку.

Чтобы использовать специальный символ как обычный, добавьте к нему обратную косую черту: `\.`.

Это называется «экранирование символа».

К примеру:

```
alert( "Глава 5.1".match(/\d\.\d/) ); // 5.1 (совпадение!)
alert( "Глава 511".match(/\d\.\d/) ); // null ("\." - ищет обычную точку)
```

Круглые скобки также являются специальными символами, поэтому, если нам нужно использовать именно их, нужно указать `\(`. В приведённом ниже примере ищется строка `"g()"`:

```
alert( "function g()".match(/g\(\)/) ); // "g()"
```

Если мы ищем обратную косую черту `\`, это специальный символ как в обычных строках, так и в регулярных выражениях, поэтому мы должны удвоить её.

```
alert( "1\\2".match(/\\/) ); // '\'
```

Косая черта

Символ косой черты `'/'`, так называемый «слэш», не является специальным символом, но в JavaScript он используется для открытия и закрытия регулярного выражения: `/...шаблон.../`, поэтому мы должны экранировать его.

Вот как выглядит поиск самой косой черты `'/'`:

```
alert( "/".match(/\//) ); // '/'
```

С другой стороны, если мы не используем короткую запись `/.../`, а создаём регулярное выражение, используя `new RegExp`, тогда нам не нужно экранировать косую черту:

```
alert( "/".match(new RegExp("/")) ); // находит /
```

new RegExp

Если мы создаём регулярное выражение с помощью `new RegExp`, то нам не нужно учитывать `/`, но нужно другое экранирование.

Например, такой поиск не работает:

```
let regexp = new RegExp("\d\.\d");

alert( "Глава 5.1".match(regexp) ); // null
```

Аналогичный поиск в примере выше с `/\d\.\d/` вполне работал, почему же не работает `new RegExp("\d\.\d")`?

Причина в том, что символы обратной косой черты «съедаются» строкой. Как вы помните, что обычные строки имеют свои специальные символы, такие как `\n`, и для экранирования используется обратная косая черта.

Вот как воспринимается строка «\d.\d»:

```
alert("\d\.\d"); // d.d
```

Строковые кавычки «съедают» символы обратной косой черты для себя, например:

- `\n` – становится символом перевода строки,
- `\u1234` – становится символом Юникода с указанным номером,
- …А когда нет особого значения: как например для `\d` или `\z`, обратная косая черта просто удаляется.

Таким образом, `new RegExp` получает строку без обратной косой черты. Вот почему поиск не работает!

Чтобы исправить это, нам нужно удвоить обратную косую черту, потому что строковые кавычки превращают `\\` в `\`:

```
let regStr = "\\d\\.\\d";
alert(regStr); // \d\.\d (теперь правильно)

let regexp = new RegExp(regStr);

alert( "Глава 5.1".match(regexp) ); // 5.1
```

Итого

- Для поиска специальных символов `[ \ ^ $ . | ? * + ( )`, нам нужно добавить перед ними `\` («экранировать их»).
- Нам также нужно экранировать `/`, если мы используем `/.../` (но не `new RegExp`).
- При передаче строки в `new RegExp` нужно удваивать обратную косую черту: `\\` для экранирования специальных символов, потому что строковые кавычки «съедят» одну черту.

Наборы и диапазоны [...]

Несколько символов или символьных классов в квадратных скобках `[…]` означают «искать любой символ из заданных».

Наборы

Для примера, `[eao]` означает любой из 3-х символов: `'a'`, `'e'` или `'o'`.

Это называется *набором*.

Наборы могут использоваться в регулярных выражениях вместе с обычными символами, например:

```
// найти [т или х], после которых идёт "оп"
alert( "Топ хоп".match(/[тх]оп/gi) ); // "топ", "хоп"
```

Обратите внимание, что в наборе несколько символов, но в результате он соответствует ровно одному символу.

Так что этот пример не даёт совпадений:

```
alert( "Вуаля".match(/В[уа]ля/) ); // null, нет совпадений
// ищет "В", затем [у или а], потом "ля"
// а в строке В, потом у, потом а
```

Шаблон ищет:

- `В`,
- затем *один* из символов `[уа]`,
- потом `ля`.

В этом случае совпадениями могут быть `Вуля` или `Валя`.

Диапазоны

Ещё квадратные скобки могут содержать *диапазоны символов*.

К примеру, `[а-я]` соответствует символу в диапазоне от `а` до `я`, или `[0-5]` – цифра от `0` до `5`.

В приведённом ниже примере мы ищем `"х"`, за которым следуют две цифры или буквы от `A` до `F`:

```
alert( "Exception 0xAF".match(/x[0-9A-F][0-9A-F]/g) ); // xAF
```

Здесь в `[0-9A-F]` сразу два диапазона: ищется символ, который либо цифра от `0` до `9`, либо буква от `A` до `F`.

Если мы хотим найти буквы и в верхнем и в нижнем регистре, то мы можем добавить ещё диапазон `a-f`: `[0-9A-Fa-f]`. Или поставить у регулярного выражения флаг `i`.

Также мы можем использовать символьные классы внутри `[…]`.

Например, если мы хотим найти «символ слова» `\w` или дефис `-`, то набор будет: `[\w-]`.

Можем использовать и несколько классов вместе, например `[\s\d]` означает «пробельный символ или цифра».

> **Символьные классы – сокращения для наборов символов**
>
> Символьные классы – не более чем сокращение для наборов символов.
>
> Например:
>
> - **\d** – то же самое, что и `[0-9]`,
> - **\w** – то же самое, что и `[a-zA-Z0-9_]`,
> - **\s** – то же самое, что и `[\t\n\v\f\r ]`, плюс несколько редких пробельных символов Юникода.

Пример: многоязычный аналог \w

Так как символьный класс `\w` является всего лишь сокращением для `[a-zA-Z0-9_]`, он не найдёт китайские иероглифы, кириллические буквы и т.п.

Давайте сделаем более универсальный шаблон, который ищет символы, используемые в словах, для любого языка. Это очень легко с Юникод-свойствами: `[\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join_C}]`.

Расшифруем его. По аналогии с классом `\w`, мы делаем свой набор, который включает в себя символы со следующими юникодными свойствами:

- `Alphabetic` (`Alpha`) – для букв,
- `Mark` (`M`) – для акцентов,
- `Decimal_Number` (`Nd`) – для цифр,
- `Connector_Punctuation` (`Pc`) – для символа подёркивания `'_'` и подобных ему,
- `Join_Control` (`Join_C`) – два специальных кода `200c` и `200d`, используемые в лигатурах, например, арабских.

Пример использования:

```
let regexp = /[\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join_C}]/gu;

let str = `Hi 你好 12`;

// найдены все буквы и цифры
alert( str.match(regexp) ); // H,i,你,好,1,2
```

Конечно, этот шаблон можно адаптировать: добавить юникодные свойства или убрать. Более подробно о них было рассказано в главе Юникод: флаг "u" и класс \p{...}.

Юникодные свойства не работают в Edge и Firefox

Юникодные свойства `p{…}` пока не реализованы в Edge и Firefox. Если они сильно нужны, можно использовать библиотеку XRegExp .

Или же использовать диапазоны символов в интересующем нас языке, например `[а-я]` для кириллицы.

Исключающие диапазоны

Помимо обычных диапазонов, есть «исключающие» диапазоны, которые выглядят как `[^…]`.

Они обозначаются символом каретки `^` в начале диапазона и соответствуют любому символу *за исключением заданных*.

Например:

- `[^aeyo]` – любой символ, за исключением `'a'`, `'e'`, `'y'` или `'o'`.
- `[^0-9]` – любой символ, за исключением цифры, то же, что и `\D`.
- `[^\s]` – любой непробельный символ, то же, что и `\S`.

Пример ниже ищет любые символы, кроме латинских букв, цифр и пробелов:

```
alert( "alice15@gmail.com".match(/[^\d\sA-Z]/gi) ); // @ и .
```

Экранирование внутри […]

Обычно, когда мы хотим найти специальный символ, нам нужно экранировать его, например `\.`. А если нам нужна обратная косая черта, тогда используем `\\`, т.п.

В квадратных скобках большинство специальных символов можно использовать без экранирования:

- Символы `. + ( )` не нужно экранировать никогда.
- Тире `-` не надо экранировать в начале или в конце (где оно не задаёт диапазон).
- Символ каретки `^` нужно экранировать только в начале (где он означает исключение).
- Закрывающую квадратную скобку `]`, если нужен именно такой символ, экранировать нужно.

Другими словами, разрешены без экранирования все специальные символы, кроме случаев, когда они означают что-то особое в наборах.

Точка `.` внутри квадратных скобок – просто точка. Шаблон `[.,]` будет искать один из символов: точку или запятую.

В приведённом ниже примере регулярное выражение `[-().^+]` ищет один из символов `-().^+`:

```
// Нет необходимости в экранировании
let regexp = /[-().^+]/g;

alert( "1 + 2 - 3".match(regexp) ); // Совпадения +, -
```

…Впрочем, если вы решите экранировать «на всякий случай», то не будет никакого вреда:

```
// Экранирование всех возможных символов
let regexp = /[\-\(\)\.\^\+]/g;

alert( "1 + 2 - 3".match(regexp) ); // также работает: +, -
```

Наборы и флаг «u»

Если в наборе есть суррогатные пары, для корректной работы обязательно нужен флаг `u`.

Например, давайте попробуем найти шаблон `[𝒳𝒴]` в строке `𝒳`:

```
alert( '𝒳'.match(/[𝒳𝒴]/) ); // покажет странный символ, что-то типа [?]
// (поиск был произведён неправильно, вернулась только половина символа)
```

Результат неверный, потому что по умолчанию регулярные выражения «не знают» о существовании суррогатных пар.

Движок регулярных выражений думает, что `[𝒳𝒴]` – это не два, а четыре символа:

1. левая половина от `𝒳` `(1)`,
2. правая половина от `𝒳` `(2)`,
3. левая половина от `𝒴` `(3)`,
4. правая половина от `𝒴` `(4)`.

Мы даже можем вывести их коды:

```
for(let i=0; i<'𝒳𝒴'.length; i++) {
  alert('𝒳𝒴'.charCodeAt(i)); // 55349, 56499, 55349, 56500
};
```

То есть в нашем примере выше ищется и выводится только левая половина от `𝒳`.

Если добавить флаг `u`, то всё будет в порядке:

```
alert( '𝒳'.match(/[𝒳𝒴]/u) ); // 𝒳
```

Аналогичная ситуация произойдёт при попытке искать диапазон: `[𝒳-𝒴]`.

Если мы забудем флаг `u`, то можем нечаянно получить ошибку:

```
'𝒳'.match(/[𝒳-𝒴]/); // Error: Invalid regular expression
```

Причина в том, что без флага `u` суррогатные пары воспринимаются как два символа, так что `[𝒳-𝒴]` воспринимается как `[<55349><56499>-<55349><56500>]` (каждая суррогатная пара заменена на коды). Теперь уже отлично видно, что диапазон `56499-55349` некорректен: его левая граница больше правой, это и есть формальная причина ошибки.

При использовании флага `u` шаблон будет работать правильно:

```
// поищем символы от 𝒳 до 𝒵
alert( '𝒴'.match(/[𝒳-𝒵]/u) ); // 𝒴
```

Задачи

Java[^script]

У нас есть регулярое выражение `/Java[^script]/`.

Найдёт ли оно что-нибудь в строке `Java`? А в строке `JavaScript`?

К решению

Найдите время как hh:mm или hh-mm

Время может быть в формате `часы:минуты` или `часы-минуты`. И часы, и минуты имеют две цифры: `09:00` или `21-30`.

Напишите регулярное выражение, чтобы найти время:

```
let regexp = /your regexp/g;
alert( "Завтрак в 09:00. Ужин в 21-30".match(regexp) ); // 09:00, 21-30
```

P.S. В этой задаче мы предполагаем, что время всегда правильное, нет необходимости отфильтровывать плохие строки, такие как «45:67». Позже мы разберёмся с этим.

К решению

Квантификаторы +, *, ? и {n}

Давайте возьмём строку вида `+7(903)-123-45-67` и найдём все числа в ней. Но теперь нас интересуют не цифры по отдельности, а именно числа: `7`, `903`, `123`, `45`, `67`.

Число — это последовательность из 1 или более цифр `\d`. Чтобы указать количество повторений, нам нужно добавить *квантификатор*.

Количество {n}

Самый простой квантификатор — это число в фигурных скобках: `{n}`.

Он добавляется к символу (или символьному классу, или набору `[...]` и т.д.) и указывает, сколько их нам нужно.

Можно по-разному указать количество, например:

Точное количество: `{5}`

Шаблон `\d{5}` обозначает ровно 5 цифр, он эквивалентен `\d\d\d\d\d`.

Следующий пример находит пятизначное число:

```
alert( "Мне 12345 лет".match(/\d{5}/) ); //  "12345"
```

Мы можем добавить `\b`, чтобы исключить числа длиннее: `\b\d{5}\b`.

Диапазон: `{3,5}`, от 3 до 5

Для того, чтобы найти числа от 3 до 5 цифр, мы можем указать границы в фигурных скобках: `\d{3,5}`

```
alert( "Мне не 12, а 1234 года".match(/\d{3,5}/) ); // "1234"
```

Верхнюю границу можно не указывать.

Тогда шаблон `\d{3,}` найдёт последовательность чисел длиной `3` и более цифр:

```
alert( "Мне не 12, а 345678 лет".match(/\d{3,}/) ); // "345678"
```

Давайте вернёмся к строке `+7(903)-123-45-67`.

Число – это последовательность из одной или более цифр. Поэтому шаблон будет `\d{1,}`:

```
let str = "+7(903)-123-45-67";

let numbers = str.match(/\d{1,}/g);

alert(numbers); // 7,903,123,45,67
```

Короткие обозначения

Для самых востребованных квантификаторов есть сокращённые формы записи:

`+`

Означает «один или более». То же самое, что и `{1,}`.

Например, `\d+` находит числа (из одной или более цифр):

```
let str = "+7(903)-123-45-67";

alert( str.match(/\d+/g) ); // 7,903,123,45,67
```

`?`

Означает «ноль или один». То же самое, что и `{0,1}`. По сути, делает символ необязательным.

Например, шаблон `ou?r` найдёт `o` после которого, возможно, следует `u`, а затем `r`.

Поэтому шаблон `colou?r` найдёт два варианта: `color` и `colour`:

```
let str = "Следует писать color или colour?";

alert( str.match(/colou?r/g) ); // color, colour
```

`*`

Означает «ноль или более». То же самое, что и `{0,}`. То есть символ может повторяться много раз или вообще отсутствовать.

Например, шаблон `\d0*` находит цифру и все нули за ней (их может быть много или ни одного):

```
alert( "100 10 1".match(/\d0*/g) ); // 100, 10, 1
```

Сравните это с `+` (один или более):

```
alert( "100 10 1".match(/\d0+/g) ); // 100, 10
// 1 не подходит, т.к 0+ требует как минимум один ноль
```

Ещё примеры

Квантификаторы используются очень часто. Они служат основными «строительными блоками» сложных регулярных выражений, поэтому давайте рассмотрим ещё примеры.

Регулярное выражение для десятичных дробей (чисел с плавающей точкой): `\d+\.\d+`

В действии:

```
alert( "0 1 12.345 7890".match(/\d+\.\d+/g) ); // 12.345
```

Регулярное выражение для «открывающего HTML-тега без атрибутов», например, `<span>` или `<p>`.

1. Самое простое: `/<[a-z]+>/i`

```
alert( "<body> ... </body>".match(/<[a-z]+>/gi) ); // <body>
```

Это регулярное выражение ищет символ `'<'`, за которым идут одна или более букв латинского алфавита, а затем `'>'`.

2. Улучшенное: `/<[a-z][a-z0-9]*>/i`

   Здесь регулярное выражение расширено: в соответствие со стандартом, в названии HTML-тега цифра может быть на любой позиции, кроме первой, например `<h1>`.

```
alert( "<h1>Привет!</h1>".match(/<[a-z][a-z0-9]*>/gi) ); // <h1>
```

Регулярное выражение для «открывающего или закрывающего HTML-тега без атрибутов»: `/<\/?[a-z][a-z0-9]*>/i`

В начало предыдущего шаблона мы добавили необязательный слеш `/?`. Этот символ понадобилось заэкранировать, чтобы JavaScript не принял его за конец шаблона.

```
alert( "<h1>Привет!</h1>".match(/<\/?[a-z][a-z0-9]*>/gi) ); // <h1>, </h1>
```

Чтобы регулярное выражение было точнее, нам часто приходится делать его сложнее

В этих примерах мы видим общее правило: чем точнее регулярное выражение – тем оно длиннее и сложнее.

Например, для HTML-тегов без атрибутов, скорее всего, подошло бы и более простое регулярное выражение: `<\w+>`. Но стандарт HTML накладывает более жёсткие ограничения на имя тега, так что более точным будет шаблон `<[a-z][a-z0-9]*>`.

Подойдёт ли нам `<\w+>` или нужно использовать `<[a-z][a-z0-9]*>`? А, может быть, нужно ещё его усложнить, добавить атрибуты?

В реальной жизни допустимы разные варианты. Ответ на подобные вопросы зависит от того, насколько реально важна точность и насколько потом будет сложно или несложно отфильтровать лишние совпадения.

Задачи

Как найти многоточие "..." ?

важность: 5

Напишите регулярное выражение, которое ищет многоточие (3 и более точек подряд).

Проверьте его:

```
let regexp = /ваше выражение/g;
alert( "Привет!... Как дела?....."match(regexp) ); // ..., .....
```

К решению

Регулярное выражение для HTML-цветов

Напишите регулярное выражение, которое ищет HTML-цвета в формате `#ABCDEF`: первым идёт символ `#`, и потом – 6 шестнадцатеричных символов.

Пример использования:

```
let regexp = /...ваше выражение.../

let str = "color:#121212; background-color:#AA00ef bad-colors:f#fddee #fd2 #1234567

alert( str.match(regexp) )  // #121212,#AA00ef
```

P.S. В рамках этого задания не нужно искать цвета, записанные в иных форматах типа `#123` или `rgb(1,2,3)`.

К решению

Жадные и ленивые квантификаторы

На первый взгляд квантификаторы – это просто, но на самом деле это не совсем так.

Нужно очень хорошо разбираться, как работает поиск, если планируем искать что-то сложнее, чем `/\d+/`.

Давайте в качестве примера рассмотрим следующую задачу:

У нас есть текст, в котором нужно заменить все кавычки `"..."` на «ёлочки» `«...»`, которые используются в типографике многих стран.

Например: `"Привет, мир"` должно превратиться в `«Привет, мир»`. Есть и другие кавычки, вроде `„Witam, świat!”` (польский язык) или `「你好，世界」` (китайский язык), но для нашей задачи давайте выберем `«...»`.

Первое, что нам нужно – это найти строки с кавычками, а затем мы сможем их заменить.

Регулярное выражение вроде `/".+"/g` (кавычка, какой-то текст, другая кавычка) может выглядеть хорошим решением, но это не так!

Давайте это проверим:

```
let regexp = /".+"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(regexp) ); // "witch" and her "broom"
```

…Как мы видим, регулярное выражение работает не как задумано!

Вместо того, чтобы найти два совпадения `"witch"` и `"broom"`, было найдено одно: `"witch" and her "broom"`.

Причину можно описать, как «жадность – причина всех зол».

Жадный поиск

Чтобы найти совпадение, движок регулярных выражений работает по следующему алгоритму:

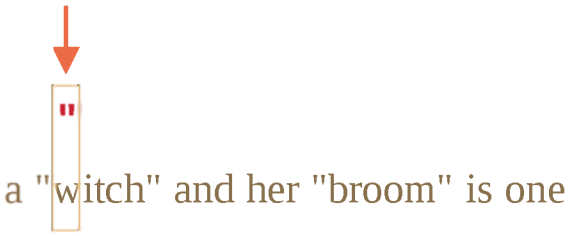
- Для каждой позиции в строке для поиска:
  - Попробовать найти совпадение с шаблоном на этой позиции.
  - Если нет совпадения, переход к следующей позиции.

Эти общие слова никак не объясняют, почему регулярное выражение работает неправильно, так что давайте разберём подробно, как работает шаблон `".+"`.

1. Первый символ шаблона – это кавычка `"`.

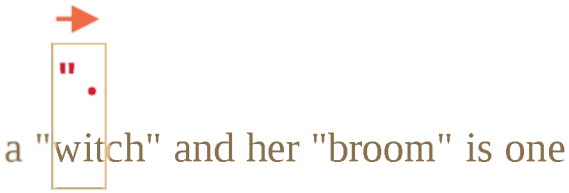
   Движок регулярного выражения пытается найти его на нулевой позиции исходной строки `a "witch" and her "broom" is one`, но там – `a`, так что совпадения нет.

   Он продолжает: двигается к следующей позиции исходной строки и пытается найти первый символ шаблона там. У него не получается, он двигается дальше и, наконец, находит кавычку на третьей позиции:



2. Кавычка замечена, после чего движок пытается найти совпадение для оставшегося шаблона. Смотрит, удовлетворяет ли остаток строки шаблону `.+"`.

   В нашем случае следующий символ шаблона: `.` (точка). Она обозначает «любой символ, кроме новой строки», так что следующая буква строки `'w'` подходит.



3. Затем точка повторяется из-за квантификатора `.+`. Движок регулярного выражения добавляет к совпадению один символ за другим.

   …До каких пор? Точке соответствуют любые символы, так что движок остановится только тогда, когда достигнет конца строки:

4. Тогда он перестанет повторять `.+` и попробует найти следующий символ шаблона. Это кавычка `"`. Но есть проблема: строка для поиска закончилась, больше нет символов!

   Движок регулярного выражения понимает, что захватил слишком много `.+` и начинает *отступать*.

   Другими словами, он сокращает совпадение по квантификатору на один символ:

Теперь он предполагает, что `.+` заканчивается за один символ до конца строки и пытается сопоставить остаток шаблона для этой позиции.

Если бы тут была кавычка, тогда бы поиск закончился, но последний символ – это `'e'`, так что он не подходит.

5. …Поэтому движок уменьшает количество повторений `.+` ещё на один символ:

Кавычка `'"'` не соответствует `'n'`.

6. Движок продолжает возвращаться: он уменьшает количество повторений `'.'`, пока оставшийся шаблон (в нашем случае `'"'`) не совпадёт:

7. Совпадение найдено.

8. Так что первое совпадение: `"witch" and her "broom"`. Если у регулярного выражения стоит флаг `g`, то поиск продолжится с того места, где закончился предыдущий. В оставшейся строке `is one` нет кавычек, так что совпадений больше не будет.

Это, определённо, не то, что мы ожидали. Но так оно работает.

В жадном режиме (по умолчанию) квантификатор повторяется столько раз, сколько это возможно.

Движок регулярного выражения пытается получить максимальное количество символов, соответствующих `.+`, а затем сокращает это количество символ за

символом, если остаток шаблона не совпадает.

В нашей задаче мы хотим другого. И нам поможет ленивый режим квантификатора.

Ленивый режим

«Ленивый» режим противоположен «жадному». Он означает: «повторять квантификатор наименьшее количество раз».

Мы можем включить его, вставив знак вопроса `'?'` после квантификатора, то есть будет `*?` или `+?` или даже `??` для `'?'`.

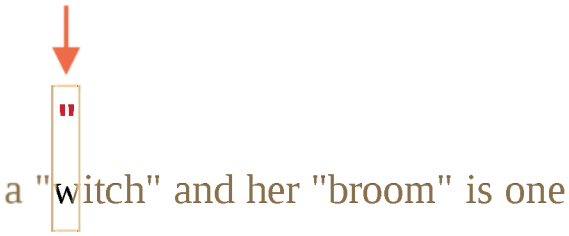
Проясним: обычно знак вопроса `?` сам по себе является квантификатором (ноль или один), но, если он добавлен *после другого квантификтора (или даже после самого себя)*, он получает другое значение – он меняет режим совпадения с жадного на ленивый.

Регулярное выражение `/".+?"/g` работает как задумано, оно находит `"witch"` и `"broom"`:

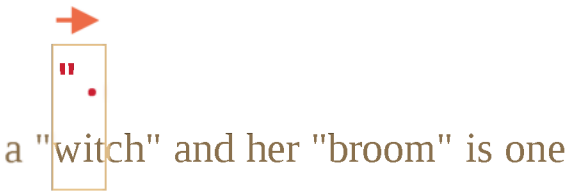
```
let regexp = /".+?"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(regexp) ); // witch, broom
```

Чтобы лучше понять, что поменялось, давайте рассмотрим процесс поиска шаг за шагом.

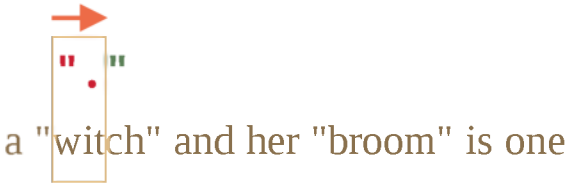
1. Первый шаг будет таким же: движок находит начало шаблона `'"'` на 3-ей позиции:



2. Следующий шаг аналогичен: он найдёт совпадение для точки `'.'`:

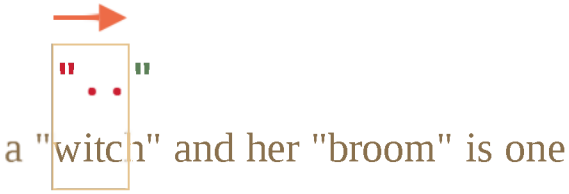


3. А отсюда поиск продолжится по-другому. Из-за того, что у нас включён ленивый режим для `+?`, движок не будет пытаться найти совпадение для точки ещё раз, оно остановится и попробует найти совпадение для оставшегося шаблона `'"'` прямо сейчас:



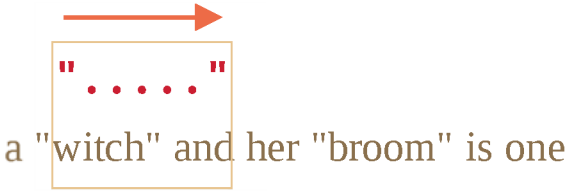
Если бы на этом месте была кавычка, то поиск бы закончился, но там находится `'i'`, то есть совпадения нет.

4. Тогда движок регулярного выражения увеличит количество повторений для точки и попробует ещё раз:



Опять неудача. Тогда количество повторений будет увеличено ещё и ещё…

5. …До тех пор, пока совпадение для оставшегося шаблона не будет найдено:



6. Следующий поиск начнётся с того места, где закончилось текущее совпадение и у нас будет ещё один результат:

В этом примере мы увидели, как ленивый режим работает для `+?`. Квантификаторы `+?` и `??` работают аналогичным образом – движок регулярного выражения увеличит количество совпадений, только если не сможет найти совпадение для оставшегося шаблона на текущей позиции.

Ленивый режим включается только для квантификаторов с `?`.

Остальные квантификаторы остаются жадными.

Например:

```
alert( "123 456".match(/\d+ \d+?/) ); // 123 4
```

1. Шаблон `\d+` пытается найти столько цифр, сколько возможно (жадный режим), так что он находит `123` и останавливается, потому что следующим символом будет пробел `' '`.
2. Дальше в шаблоне пробел и в строке тоже, так что есть совпадение.
3. Затем идёт `\d+?`. Квантификатор находится в ленивом режиме, так что он находит одну цифру `4` и проверяет, есть ли совпадение для оставшегося шаблона с этого места.

   …Но в шаблоне `\d+?` больше ничего нет.

   Ленивый режим ничего не повторяет без необходимости. Шаблон закончился, заканчивается и поиск. Мы получаем `123 4`.

Оптимизации

Современные движки регулярных выражений могут оптимизировать внутренние алгоритмы ради ускорения. Так что их работа может несколько отличаться от описанного алгоритма.

Но эти внутренние оптимизации для нас незаметны, снаружи всё будет работать, как описано.

Сложные регулярные выражения трудно оптимизировать, так что поиск может работать и в точности так, как было описано.

Альтернативный подход

С регулярными выражениями часто есть несколько путей добиться одного и того же результата.

В нашем случаем мы можем найти кавычки без использования ленивого режима с помощью регулярного выражения `"[^"]+"`:

```
let regexp = /"[^"]+"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(regexp) ); // witch, broom
```

Регулярное выражение `"[^"]+"` получит нужный результат, потому что оно ищет кавычку `'"'`, за которой следует один или несколько символов «не-кавычек» `[^"]`, а затем – закрывающая кавычка.

Движок регулярного выражения набирает, сколько может, `[^"]+`, пока не встречает закрывающую кавычку, на которой останавливается.

Обратите внимание, что эта логика не заменяет ленивые квантификаторы!

Просто она работает по-другому. Временами нужен один вариант, временами – другой.

Давайте посмотрим пример, в котором ленивый квантификатор не справляется, а этот вариант работает правильно.

Например, мы хотим найти ссылки вида `<a href="..." class="doc">`, с произвольным `href`.

Какое регулярное выражение нам нужно использовать?

Первой мыслью может быть: `/<a href=".*" class="doc">/g`.

Давайте проверим:

```
let str = '...<a href="link" class="doc">...';
let regexp = /<a href=".*" class="doc">/g;

// Работает!
alert( str.match(regexp) ); // <a href="link" class="doc">
```

Регулярное выражение работает. Но давайте посмотрим, что произойдёт, если в тексте будет много ссылок?

```
let str = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
let regexp = /<a href=".*" class="doc">/g;

// Упс! Две ссылки в одном совпадении!
alert( str.match(regexp) ); // <a href="link1" class="doc">... <a href="link2" clas
```

В данном случае мы получили неправильный результат по той же причине, что в примере с «witches». Квантификатор `.*` забирает слишком много символов.

Совпадение будет выглядеть так:

```
<a href="..................................." class="doc">
<a href="link1" class="doc">... <a href="link2" class="doc">
```

Давайте изменим шаблон, сделав квантификатор ленивым `.*?`:

```
let str = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
let regexp = /<a href=".*?" class="doc">/g;

// Работает!
alert( str.match(regexp) ); // <a href="link1" class="doc">, <a href="link2" class=
```

Теперь кажется, что всё работает правильно. У нас есть два совпадения:

```
<a href="....." class="doc">    <a href="....." class="doc">
<a href="link1" class="doc">... <a href="link2" class="doc">
```

…Но давайте попробуем его на ещё одном тексте:

```
let str = '...<a href="link1" class="wrong">... <p style="" class="doc">...';
let regexp = /<a href=".*?" class="doc">/g;

// Неправильное совпадение!
alert( str.match(regexp) ); // <a href="link1" class="wrong">... <p style="" class=
```

Ну вот, ленивый квантификатор нас подвёл. В совпадении находится не только ссылка, но и текст после неё, включая `<p...>`.

Почему?

Происходит следующее:

1. Первым делом регулярное выражение находит начало ссылки `<a href="`.

2. Затем оно ищет `.*?`, берёт один символ (лениво!) и проверяет, есть ли совпадение для `"` (нет).
3. Затем берёт другой символ для `.*?`, и так далее… пока не достигнет `" class="doc">`. Поиск завершён.

Но с этим есть проблема: конец совпадения находится уже за границей ссылки `<a...>`, вообще в другом теге `<p>`. Что нам не подходит.

Вот как совпадение выглядит по отношению к исходному тексту:

```
<a href="..................................." class="doc">
<a href="link1" class="wrong">... <p style="" class="doc">
```

Итак, нужен шаблон для поиска `<a href="...something..." class="doc">`, но и с ленивым и с жадным режимами есть проблема.

Правильным вариантом может стать: `href="[^"]*"`. Он найдёт все символы внутри атрибута `href` до ближайшей следующей кавычки, как раз то, что нам нужно.

Работающий пример:

```
let str1 = '...<a href="link1" class="wrong">... <p style="" class="doc">...';
let str2 = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
let regexp = /<a href="[^"]*" class="doc">/g;

// Работает!
alert( str1.match(regexp) ); // совпадений нет, всё правильно
alert( str2.match(regexp) ); // <a href="link1" class="doc">, <a href="link2" class=
```

Итого

У квантификаторов есть два режима работы:

Жадный

По умолчанию движок регулярного выражения пытается повторить квантификатор столько раз, сколько это возможно. Например, `\d+` получит все возможные цифры. Когда цифры закончатся или он дойдёт до конца строки, движок продолжит искать совпадение для оставшегося шаблона. Если совпадения не будет, он уменьшит количество повторов (осуществит возврат) и попробует снова.

Ленивый

Включается с помощью знака вопроса `?` после квантификатора. Движок регулярного выражения пытается найти совпадение для оставшегося шаблона перед каждым повторением квантификатора.

Как мы увидели на примере поиска строк в кавычках, ленивый режим не «панацея» от всех проблем жадного поиска. В качестве альтернативы может выступать «хорошо настроенный» жадный поиск, как в шаблоне `"[^"]+"`.

Задачи

Совпадение для /d+? d+?/

Какое здесь будет совпадение?

```
"123 456".match(/\d+? \d+?/g) ); // ?
```

К решению

Поиск HTML-комментариев

Найти все HTML-комментарии в тексте:

```
let regexp = /ваше регулярное выражение/g;

let str = `... <!-- My -- comment
 test --> ..  <!----> ..
`;

alert( str.match(regexp) ); // '<!-- My -- comment \n test -->', '<!---->'
```

К решению

Поиск HTML-тегов

Создайте регулярное выражение, чтобы найти все (открывающие и закрывающие) HTML-теги с их атрибутами.

Пример использования:

```
let regexp = /ваше регулярное выражение/g;

let str = '<> <a href="/"> <input type="radio" checked> <b>';

alert( str.match(regexp) ); // '<a href="/">', '<input type="radio" checked>', '<b>
```

В этой задаче мы предполагаем, что теги выглядят как `<...что угодно...>`, и внутри тегов не может быть символов `<` и `>` (первый встреченный `>` закрывает тег).

К решению

Скобочные группы

Часть шаблона можно заключить в скобки `(...)`. Это называется «скобочная группа».

У такого выделения есть два эффекта:

1. Позволяет поместить часть совпадения в отдельный массив.
2. Если установить квантификтор после скобок, то он будет применяться ко всему содержимому скобки, а не к одному символу.

Примеры

Разберём скобки на примерах.

Пример: gogogo

Без скобок шаблон `go+` означает символ `g` и идущий после него символ `o`, который повторяется один или более раз. Например, `goooo` или `gooooooooo`.

Скобки группируют символы вместе. Так что `(go)+` означает `go`, `gogo`, `gogogo` и т.п.

```
alert( 'Gogogo now!'.match(/(go)+/i) ); // "Gogogo"
```

Пример: домен

Сделаем что-то более сложное – регулярное выражение, которое соответствует домену сайта.

Например:

```
mail.com
users.mail.com
smith.users.mail.com
```

Как видно, домен состоит из повторяющихся слов, причём после каждого, кроме последнего, стоит точка.

На языке регулярных выражений `(\w+\.)+\w+`:

```
let regexp = /(\w+\.)+\w+/g;

alert( "site.com my.site.com".match(regexp) ); // site.com,my.site.com
```

Поиск работает, но такому шаблону не соответствует домен с дефисом, например, `my-site.com`, так как дефис не входит в класс `\w`.

Можно исправить это, заменим `\w` на `[\w-]` везде, кроме как в конце: `([\w-]+\.)+\w+`.

Пример: email

Предыдущий пример можно расширить, создав регулярное выражение для поиска email.

Формат email: `имя@домен`. В качестве имени может быть любое слово, разрешены дефисы и точки. На языке регулярных выражений это `[-.\w]+`.

Итоговый шаблон:

```
let regexp = /[-.\w]+@([\w-]+\.)+[\w-]+/g;

alert("my@mail.com @ his@site.com.uk".match(regexp)); // my@mail.com, his@site.com.
```

Это регулярное выражение не идеальное, но, как правило, работает и помогает исправлять опечатки. Окончательную проверку правильности email, в любом случае, можно осуществить, лишь послав на него письмо.

Содержимое скобок в match

Скобочные группы нумеруются слева направо. Поисковой движок запоминает содержимое, которое соответствует каждой скобочной группе, и позволяет получить его в результате.

Метод `str.match(regexp)`, если у регулярного выражения `regexp` нет флага `g`, ищет первое совпадение и возвращает его в виде массива:

1. На позиции `0` будет всё совпадение целиком.
2. На позиции `1` – содержимое первой скобочной группы.
3. На позиции `2` – содержимое второй скобочнай группы.
4. …и так далее…

Например, мы хотим найти HTML теги `<.*?>` и обработать их. Было бы удобно иметь содержимое тега (то, что внутри уголков) в отдельной переменной.

Давайте заключим внутреннее содержимое в круглые скобки: `<(.*?)>`.

Теперь получим как тег целиком `<h1>`, так и его содержимое `h1` в виде массива:

```
let str = '<h1>Hello, world!</h1>';

let tag = str.match(/<(.*?)>/);

alert( tag[0] ); // <h1>
alert( tag[1] ); // h1
```

Вложенные группы

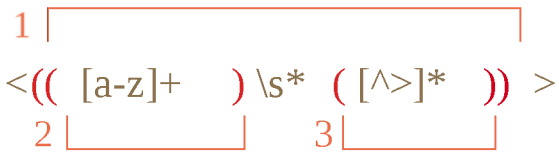
Скобки могут быть и вложенными.

Например, при поиске тега в `<span class="my">` нас может интересовать:

1. Содержимое тега целиком: `span class="my"`.
2. Название тега: `span`.
3. Атрибуты тега: `class="my"`.

Заключим их в скобки в шаблоне: `<(([a-z]+)\s*([^>]*))>`.

Вот их номера (слева направо, по открывающей скобке):



В действии:

```
let str = '<span class="my">';

let regexp = /<(([a-z]+)\s*([^>]*))>/;

let result = str.match(regexp);
alert(result[0]); // <span class="my">
alert(result[1]); // span class="my"
alert(result[2]); // span
alert(result[3]); // class="my"
```

По нулевому индексу в `result` всегда идёт полное совпадение.

Затем следуют группы, нумеруемые слева направо, по открывающим скобкам. Группа, открывающая скобка которой идёт первой, получает первый индекс в результате – `result[1]`. Там находится всё содержимое тега.

Затем в `result[2]` идёт группа, образованная второй открывающей скобкой `([a-z]+)` – имя тега, далее в `result[3]` будет остальное содержимое тега: `([^>]*)`.

Соответствие для каждой группы в строке:

Необязательные группы

Даже если скобочная группа необязательна (например, стоит квантификатор `(...)?`), соответствующий элемент массива `result` существует и равен `undefined`.

Например, рассмотрим регулярное выражение `a(z)?(c)?`. Оно ищет букву `"a"`, за которой идёт необязательная буква `"z"`, за которой, в свою очередь, идёт необязательная буква `"c"`.

Если применить его к строке из одной буквы `a`, то результат будет такой:

```
let match = 'a'.match(/a(z)?(c)?/);

alert( match.length ); // 3
alert( match[0] ); // a (всё совпадение)
alert( match[1] ); // undefined
alert( match[2] ); // undefined
```

Массив имеет длину `3`, но все скобочные группы пустые.

А теперь более сложная ситуация для строки `ac`:

```
let match = 'ac'.match(/a(z)?(c)?/)

alert( match.length ); // 3
alert( match[0] ); // ac (всё совпадение)
alert( match[1] ); // undefined, потому что для (z)? ничего нет
alert( match[2] ); // c
```

Длина массива всегда равна `3`. Для группы `(z)?` ничего нет, поэтому результат: `["ac", undefined, "c"]`.

Поиск всех совпадений с группами: matchAll

`matchAll` является новым, может потребоваться полифил

Метод не поддерживается в старых браузерах.

Может потребоваться полифил, например https://github.com/ljharb/String.prototype.matchAll .

При поиске всех совпадений (флаг `g`) метод `match` не возвращает скобочные группы.

Например, попробуем найти все теги в строке:

```
let str = '<h1> <h2>';

let tags = str.match(/<(.*?)>/g);

alert( tags ); // <h1>,<h2>
```

Результат – массив совпадений, но без деталей о каждом. Но на практике скобочные группы тоже часто нужны.

Для того, чтобы их получать, мы можем использовать метод `str.matchAll(regexp)`.

Он был добавлен в язык JavaScript гораздо позже чем `str.match`, как его «новая и улучшенная» версия.

Он, как и `str.match(regexp)`, ищет совпадения, но у него есть три отличия:

1. Он возвращает не массив, а перебираемый объект.
2. При поиске с флагом `g`, он возвращает каждое совпадение в виде массива со скобочными группами.
3. Если совпадений нет, он возвращает не `null`, а просто пустой перебираемый объект.

Например:

```
let results = '<h1> <h2>'.matchAll(/<(.*?)>/gi);

// results - не массив, а перебираемый объект
alert(results); // [object RegExp String Iterator]

alert(results[0]); // undefined (*)

results = Array.from(results); // превращаем в массив
```

```
alert(results[0]); // <h1>,h1 (первый тег)
alert(results[1]); // <h2>,h2 (второй тег)
```

Как видите, первое отличие – очень важное, это демонстрирует строка `(*)`. Мы не можем получить совпадение как `results[0]`, так как этот объект не является псевдомассивом. Его можно превратить в настоящий массив при помощи `Array.from`. Более подробно о псевдомассивах и перебираемых объектов мы говорили в главе Перебираемые объекты.

В явном преобразовании через `Array.from` нет необходимости, если мы перебираем результаты в цикле, вот так:

```
let results = '<h1> <h2>'.matchAll(/<(.*?)>/gi);

for(let result of results) {
  alert(result);
  // первый вывод: <h1>,h1
  // второй: <h2>,h2
}
```

…Или используем деструктуризацию:

```
let [tag1, tag2] = '<h1> <h2>'.matchAll(/<(.*?)>/gi);
```

Каждое совпадение, возвращаемое `matchAll`, имеет тот же вид, что и при `match` без флага `g`: это массив с дополнительными свойствами `index` (позиция совпадения) и `input` (исходный текст):

```
let results = '<h1> <h2>'.matchAll(/<(.*?)>/gi);

let [tag1, tag2] = results;

alert( tag1[0] ); // <h1>
alert( tag1[1] ); // h1
alert( tag1.index ); // 0
alert( tag1.input ); // <h1> <h2>
```

Почему результат `matchAll` – перебираемый объект, а не обычный массив?

Зачем так сделано? Причина проста – для оптимизации.

При вызове `matchAll` движок JavaScript возвращает перебираемый объект, в котором ещё нет результатов. Поиск осуществляется по мере того, как мы запрашиваем результаты, например, в цикле.

Таким образом, будет найдено ровно столько результатов, сколько нам нужно.

Например, всего в тексте может быть 100 совпадений, а в цикле после 5го результата мы поняли, что нам их достаточно и сделали `break`. Тогда движок не будет тратить время на поиск остальных 95.

Именованные группы

Запоминать группы по номерам не очень удобно. Для простых шаблонов это допустимо, но в сложных регулярных выражениях считать скобки затруднительно. Гораздо лучше – давать скобкам имена.

Это делается добавлением `?<name>` непосредственно после открытия скобки.

Например, поищем дату в формате «день-месяц-год»:

```
let dateRegexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/;
let str = "2019-04-30";

let groups = str.match(dateRegexp).groups;

alert(groups.year); // 2019
alert(groups.month); // 04
alert(groups.day); // 30
```

Как вы можете видеть, группы располагаются в свойстве `groups` результата `match`.

Чтобы найти не только первую дату, используем флаг `g`.

Также нам понадобится `matchAll`, чтобы получить скобочные группы:

```
let dateRegexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/g;

let str = "2019-10-30 2020-01-01";

let results = str.matchAll(dateRegexp);
```

```
for(let result of results) {
  let {year, month, day} = result.groups;

  alert(`${day}.${month}.${year}`);
  // первый вывод: 30.10.2019
  // второй: 01.01.2020
}
```

Скобочные группы при замене

Метод `str.replace(regexp, replacement)`, осуществляющий замену совпадений с `regexp` в строке `str`, позволяет использовать в строке замены содержимое скобок. Это делается при помощи обозначений вида `$n`, где `n` – номер скобочной группы.

Например:

```
let str = "John Bull";
let regexp = /(\w+) (\w+)/;

alert( str.replace(regexp, '$2, $1') ); // Bull, John
```

Для именованных скобок ссылка будет выглядеть как `$<имя>`.

Например, заменим даты в формате «год-месяц-день» на «день.месяц.год»:

```
let regexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/g;

let str = "2019-10-30б 2020-01-01";

alert( str.replace(regexp, '$<day>.$<month>.$<year>') );
// 30.10.2019б 01.01.2020
```

Исключение из запоминания через ?:

Бывает так, что скобки нужны, чтобы квантификатор правильно применился, но мы не хотим, чтобы их содержимое было выделено в результате.

Скобочную группу можно исключить из запоминаемых и нумеруемых, добавив в её начало `?:`.

Например, если мы хотим найти `(go)+`, но не хотим иметь в массиве-результате отдельным элементом содержимое скобок (`go`), то можем написать `(?:go)+`.

В примере ниже мы получим только имя `John` как отдельный элемент совпадения:

```
let str = "Gogogo John!";

// ?: исключает go из запоминания
let regexp = /(?:go)+ (\w+)/i;

let result = str.match(regexp);

alert( result[0] ); // Gogogo John (полное совпадение)
alert( result[1] ); // John
alert( result.length ); // 2 (больше в массиве элементов нет)
```

Как видно, содержимое скобок `(?:go)` не стало отдельным элементом массива `result`.

Итого

Круглые скобки группируют вместе часть регулярного выражения, так что квантификатор применяется к ним в целом.

Скобочные группы нумеруются слева направо. Также им можно дать имя с помощью `(?<name>...)`.

Часть совпадения, соответствующую скобочной группе, мы можем получить в результатах поиска.

- Метод `str.match` возвращает скобочные группы только без флага `g`.
- Метод `str.matchAll` возвращает скобочные группы всегда.

Если скобка не имеет имени, то содержимое группы будет по своему номеру в массиве-результате, если имеет, то также в свойстве `groups`.

Содержимое скобочной группы можно также использовать при замене `str.replace(regexp, replacement)`: по номеру `$n` или по имени `$<имя>`.

Можно исключить скобочную группу из запоминания, добавив в её начало `?:`. Это используется, если необходимо применить квантификатор ко всей группе, но не запоминать их содержимое в отдельном элементе массива-результата. Также мы не можем ссылаться на такие скобки в строке замены.

Задачи

Найти цвет в формате #abc или #abcdef

Напишите регулярное выражение, которое соответствует цветам в формате `#abc` или `#abcdef`. То есть: `#` и за ним 3 или 6 шестнадцатеричных цифр.

Пример использования:

```
let regexp = /ваш шаблон/g;

let str = "color: #3f3; background-color: #AA00ef; and: #abcd";

alert( str.match(regexp) ); // #3f3 #AA00ef
```

P.S. Это должно быть ровно 3 или 6 шестнадцатеричных цифр. При этом значения с 4-мя цифрами типа `#abcd` не должны совпадать в результат.

К решению

Найти все числа

Напишите регулярное выражение, которое ищет любые десятичные числа, включая целочисленные, с плавающей точкой и отрицательные.

Пример использования:

```
let regexp = /ваш шаблон/g;

let str = "-1.5 0 2 -123.4.";

alert( str.match(regexp) ); // -1.5, 0, 2, -123.4
```

К решению

Разобрать выражение

Арифметическое выражение включает два числа и оператор между ними. Например:

- `1 + 2`
- `1.2 * 3.4`
- `-3 / -6`
- `-2 - 2`

Оператором может быть: `"+"`, `"-"`, `"*"` или `"/"`.

В выражении могут быть пробелы в начале, в конце или между частями выражения.

Создайте функцию `parse(expr)`, которая принимает выражение и возвращает массив из трёх элементов:

1. Первое число.
2. Оператор.
3. Второе число.

Например:

```
let [a, op, b] = parse("1.2 * 3.4");

alert(a); // 1.2
alert(op); // *
alert(b); // 3.4
```

К решению

Проверьте MAC-адрес

MAC-адрес сетевого интерфейса состоит из 6-ти двузначных шестнадцатеричных чисел, разделённых двоеточиями.

Например: `'01:32:54:67:89:AB'`.

Напишите регулярное выражение, которое проверит, является ли строка MAC-адресом.

Использование:

```
let regexp = /ваш regexp/;

alert( regexp.test('01:32:54:67:89:AB') ); // true

alert( regexp.test('0132546789AB') ); // false (нет двоеточий)

alert( regexp.test('01:32:54:67:89') ); // false (5 чисел, должно быть 6)

alert( regexp.test('01:32:54:67:89:ZZ') ) // false (ZZ в конце строки)
```

К решению

Обратные ссылки в шаблоне: \N и \k<имя>

Доступ к содержимому скобочных групп `(...)` есть не только в результате поиска и при замене, но и в самом шаблоне.

Обратная ссылка по номеру: \N

К группе можно обратиться в шаблоне, используя `\N`, где `N` – это номер группы.

Чтобы было яснее, зачем это нужно, рассмотрим пример.

Необходимо найти строки в кавычках: либо одинарных `'...'`, либо двойных `"..."` – оба варианта должны подходить.

Как найти такие строки?

Можно попытаться добавить оба вида кавычек в квадратные скобки: `['"]` `(.*?)['"]`, но в таком случае будут находиться строки со смешанными кавычками, например `"...'` и `'..."`. Это приведёт к ошибке, когда одна кавычка окажется внутри других, как в строке `"She's the one!"`:

```
let str = `He said: "She's the one!".`;

let regexp = /['"](.*?)['"]/g;

// Результат - не тот, который хотелось бы
alert( str.match(regexp) ); // "She'
```

Как видно, шаблон нашёл открывающую кавычку `"`, а после нашёл текст вплоть до следующей кавычки `'`, после чего поиск завершился.

Для того, чтобы шаблон искал закрывающую кавычку такую же, как и открывающую, обернём открывающие кавычки в скобочную группу и используем обратную ссылку на неё: `(['"])(.*?)\1`.

Вот верный код:

```
let str = `He said: "She's the one!".`;

let regexp = /(['"])(.*?)\1/g;

alert( str.match(regexp) ); // "She's the one!"
```

Теперь работает! Движок регулярных выражений находит первую кавычку из шаблона `(['"])` и запоминает её содержимое. Это первая скобочная группа.

Далее в шаблоне `\1` означает «найти то же самое, что в первой скобочной группе», а именно – аналогичную кавычку в нашем случае.

Аналогично, `\2` означает содержимое второй скобочной группы, `\3` – третьей, и так далее.

> **На заметку:**
>
> Мы не можем обратиться к группе, которая исключена из запоминания при помощи `?:`.

> **Не перепутайте: в шаблоне `\1`, при замене `$1`**
>
> В строке замены для вставки группы мы используем доллар: `$1`, а в шаблоне обратный слеш `\1`.

Обратная ссылка по имени: `\k<имя>`

Если в регулярном выражении много скобочных групп, то удобно давать им имена.

Для обращения к именованной группе можно использовать синтаксис `\k<имя>`.

В примере ниже кавычки обозначены `?<quote>`, так что обращение будет `\k<quote>`:

```
let str = `He said: "She's the one!".`;

let regexp = /(?<quote>['"])(.*?)\k<quote>/g;

alert( str.match(regexp) ); // "She's the one!"
```

Альтернация (или) |

Альтернация – термин в регулярных выражениях, которому в русском языке соответствует слово «ИЛИ».

В регулярных выражениях она обозначается символом вертикальной черты `|`.

Например, нам нужно найти языки программирования: HTML, PHP, Java и JavaScript.

Соответствующее регулярное выражение: `html|php|java(script)?`.

Пример использования:

```
let regexp = /html|css|java(script)?/gi;

let str = "Сначала появился язык Java, затем HTML, потом JavaScript";

alert( str.match(regexp) ); // Java,HTML,JavaScript
```

Мы уже видели нечто подобное – квадратные скобки. Они позволяют выбирать между несколькими символами, например `gr[ae]y` найдёт `gray`, либо `grey`.

Квадратные скобки работают только с символами или наборами символов. Альтернация мощнее, она работает с любыми выражениями. Регулярное выражение `A|B|C` обозначает поиск одного из выражений: `A`, `B` или `C`.

Например:

- `gr(a|e)y` означает точно то же, что и `gr[ae]y`.
- `gra|ey` означает `gra` или `ey`.

Чтобы применить альтернацию только к части шаблона, можно заключить её в скобки:

- `Люблю HTML|CSS` найдёт `Люблю HTML` или `CSS`.
- `Люблю (HTML|CSS)` найдёт `Люблю HTML` или `Люблю CSS`.

Пример: шаблон для времени

В предыдущих главах было задание написать регулярное выражение для поиска времени в формате `чч:мм`, например `12:00`. Но шаблон `\d\d:\d\d` недостаточно точный. Он принимает `25:99` за время (99 секунд подходят под шаблон, но так не должно быть).

Как сделать лучше?

Мы можем применить более тщательное сравнение. Во-первых, часы:

- Если первая цифра `0` или `1`, тогда следующая цифра может быть любой: `[01]\d`.
- Или если первая цифра `2`, тогда следующая должна быть от 0 до 3: `2[0-3]`.
- (другой первой цифры быть не может)

В виде регулярного выражения оба варианта для часов можно записать при помощи альтернации: `[01]\d|2[0-3]`.

Далее, минуты должны быть от `00` до `59`. На языке регулярных выражений это означает `[0-5]\d`: первая цифра `0-5`, а за ней любая.

Давайте соединим часы и минуты в одно выражение, получится так: `[01]\d|2[0-3]:[0-5]\d`.

Почти готово, но есть проблема. После такого соединения альтернация `|` оказалась между `[01]\d` и `2[0-3]:[0-5]\d`.

То есть, минуты добавились ко второму варианту альтернации, вот более наглядно:

```
[01]\d  |  2[0-3]:[0-5]\d
```

Такой шаблон будет искать `[01]\d` или `2[0-3]:[0-5]\d`.

Но это неверно. Нам нужно, чтобы альтернация использовалась только внутри части регулярного выражения, относящейся к часам, чтобы разрешать `[01]\d` ИЛИ `2[0-3]`. Для этого обернём «часы» в скобки: `([01]\d|2[0-3]):[0-5]\d`.

Пример работы:

```
let regexp = /([01]\d|2[0-3]):[0-5]\d/g;

alert("00:00 10:10 23:59 25:99 1:2".match(regexp)); // 00:00,10:10,23:59
```

Задачи

Найдите языки программирования

Существует много языков программирования, например, Java, JavaScript, PHP, C, C++.

Напишите регулярное выражение, которое найдёт их все в строке `Java JavaScript PHP C++ C`:

```
let regexp = /ваше регулярное выражение/флаги;

alert("Java JavaScript PHP C++ C".match(regexp)); // Java JavaScript PHP C++ C
```

К решению

Найдите пары BB-кодов

BB-код имеет вид `[tag]...[/tag]`, где `tag` – это один из: `b`, `url` или `quote`.

Например:

```
[b]текст[/b]
[url]http://ya.ru[/url]
```

BB-коды могут быть вложенными. Но сам в себя тег быть вложен не может, например:

```
Допустимо:
[url] [b]http://ya.ru[/b] [/url]
[quote] [b]текст[/b] [/quote]

Нельзя:
[b][b]текст[/b][/b]
```

Теги могут содержать переносы строк, это допустимо:

```
[quote]
  [b]текст[/b]
[/quote]
```

Создайте регулярное выражение для поиска всех BB-кодов и их содержимого.

Например:

```
let regexp = /ваше регулярное выражение/флаги;

let str = "..[url]http://ya.ru[/url]..";
alert( str.match(regexp) ); // [url]http://ya.ru[/url]
```

Если теги вложены, то нужно искать самый внешний тег (при желании можно продолжить поиск в его содержимом):

```
let regexp = /ваше регулярное выражение/флаги;

let str = "..[url][b]http://ya.ru[/b][/url]..";
alert( str.match(regexp) ); // [url][b]http://ya.ru[/b][/url]
```

К решению

Найдите строки в кавычках

Создайте регулярное выражение для поиска строк в двойных кавычках `"..."`.

Важно, что строки должны поддерживать экранирование с помощью обратного слеша, по аналогии со строками JavaScript. Например, кавычки могут быть вставлены как `\"`, новая строка как `\n`, а сам обратный слеш как `\\`.

```
let str = "Как вот \"здесь\".";
```

В частности, обратите внимание: двойная кавычка после обратного слеша `\"` не оканчивает строку.

Поэтому мы должны искать от одной кавычки до другой, игнорируя встречающиеся экранированные кавычки.

В этом и состоит основная сложность задачи, которая без этого условия была бы элементарной.

Примеры подходящих строк:

```
.. "test me" ..
.. "Скажи \"Привет\"!" ... (строка с экранированными кавычками)
.. "\\" ..  (внутри двойной слеш)
.. "\\ \"" ..  (внутри двойной слеш и экранированная кавычка)
```

В JavaScript приходится удваивать обратные слеши, чтобы добавлять их в строку, как здесь:

```
let str = ' .. "test me" .. "Скажи \\"Привет\\"!" .. "\\\\ \\"" .. ';

// эта строка в памяти:
alert(str); //  .. "test me" .. "Скажи \"Привет\"!" .. "\\ \"" ..
```

К решению

Найдите весь тег

Напишите регулярное выражение, которое ищет тег `<style...>`. Оно должно искать весь тег: он может как не иметь атрибутов `<style>`, так и иметь несколько `<style type="..." id="...">`.

…Но регулярное выражение не должно находить `<styler>`!

Например:

```
let regexp = /ваше регулярное выражение/g;

alert( '<style> <styler> <style test="...">'.match(regexp) ); // <style>, <style te
```

К решению

Опережающие и ретроспективные проверки

В некоторых случаях нам нужно найти соответствия шаблону, но только те, за которыми или перед которыми следует другой шаблон.

Для этого в регулярных выражениях есть специальный синтаксис: опережающая (lookahead) и ретроспективная (lookbehind) проверка.

В качестве первого примера найдём стоимость из строки `1 индейка стоит 30€`. То есть, найдём число, после которого есть знак валюты `€`.

Опережающая проверка

Синтаксис опережающей проверки: `X(?=Y)`.

Он означает: найди `X` при условии, что за ним следует `Y`. Вместо `X` и `Y` здесь может быть любой шаблон.

Для целого числа, за которым идёт знак `€`, шаблон регулярного выражения будет `\d+(?=€)`:

```
let str = "1 индейка стоит 30€";

alert( str.match(/\d+(?=€)/) ); // 30, число 1 проигнорировано, так как за ним НЕ с
```

Обратим внимание, что проверка – это именно проверка, содержимое скобок `(?=...)` не включается в результат `30`.

При поиске `X(?=Y)` движок регулярных выражений, найдя `X`, проверяет есть ли после него `Y`. Если это не так, то игнорирует совпадение и продолжает поиск дальше.

Возможны и более сложные проверки, например `X(?=Y)(?=Z)` означает:

1. Найти `X`.

2. Проверить, идёт ли `Y` сразу после `X` (если нет – не подходит).
3. Проверить, идёт ли `Z` сразу после `X` (если нет – не подходит).
4. Если обе проверки прошли – совпадение найдено.

То есть, этот шаблон означает, что мы ищем `X` при условии, что за ним идёт и `Y` и `Z`.

Такое возможно только при условии, что шаблоны `Y` и `Z` не являются взаимно исключающими.

Например, `\d+(?=\s)(?=.*30)` ищет `\d+` при условии, что за ним идёт пробел, и где-то впереди есть `30`:

```
let str = "1 индейка стоит 30€";

alert( str.match(/\d+(?=\s)(?=.*30)/) ); // 1
```

В нашей строке это как раз число `1`.

Негативная опережающая проверка

Допустим, нам нужно узнать из этой же строки количество индеек, то есть число `\d+`, за которым НЕ следует знак `€`.

Для этой задачи мы можем применить негативную опережающую проверку.

Синтаксис: `X(?!Y)`

Он означает: найди такой `X`, за которым НЕ следует `Y`.

```
let str = "2 индейки стоят 60€";

alert( str.match(/\d+(?!€)/) ); // 2 (в этот раз проигнорирована цена)
```

Ретроспективная проверка

Опережающие проверки позволяют задавать условия на то, что «идёт после».

Ретроспективная проверка выполняет такую же функцию, но с просмотром назад. Другими словами, она находит соответствие шаблону, только если перед ним есть что-то заранее определённое.

Синтаксис:

- Позитивная ретроспективная проверка: `(?<=Y)X`, ищет совпадение с `X` при условии, что перед ним ЕСТЬ `Y`.
- Негативная ретроспективная проверка: `(?<!Y)X`, ищет совпадение с `X` при условии, что перед ним НЕТ `Y`.

Чтобы протестировать ретроспективную проверку, давайте поменяем валюту на доллары США. Знак доллара обычно ставится перед суммой денег, поэтому для того чтобы найти `$30`, мы используем `(?<=\$)\d+` – число, перед которым идёт `$`:

```
let str = "1 индейка стоит $30";

// знак доллара экранируем \$, так как это специальный символ
alert( str.match(/(?<=\$)\d+/) ); // 30, одинокое число игнорируется
```

Если нам необходимо найти количество индеек – число, перед которым не идёт `$`, мы можем использовать негативную ретроспективную проверку `(?<!\$)\d+`:

```
let str = "2 индейки стоят $60";

alert( str.match(/(?<!\$)\d+/) ); // 2 (проигнорировалась цена)
```

Скобочные группы

Как правило, то что находится внутри скобок, задающих опережающую и ретроспективную проверку, не включается в результат совпадения.

Например, в шаблоне `\d+(?=€)` знак `€` не будет включён в результат. Это логично, ведь мы ищем число `\d+`, а `(?=€)` – это всего лишь проверка, что за ним идёт знак `€`.

Но в некоторых ситуациях нам может быть интересно захватить и то, что в проверке. Для этого нужно обернуть это в дополнительные скобки.

В следующем примере знак валюты `(€|kr)` будет включён в результат вместе с суммой:

```
let str = "1 индейка стоит 30€";
let regexp = /\d+(?=(€|kr))/; // добавлены дополнительные скобки вокруг €|kr

alert( str.match(regexp) ); // 30, €
```

Тоже самое можно применить к ретроспективной проверке:

```
let str = "1 индейка стоит $30";
let regexp = /(?<=(\$|£))\d+/;

alert( str.match(regexp) ); // 30, $
```

Итого

Опережающая и ретроспективная проверки удобны, когда мы хотим искать шаблон по дополнительному условию на контекст, в котором он находится.

Для простых регулярных выражений мы можем сделать похожую вещь «вручную». То есть, найти все совпадения, независимо от контекста, а затем в цикле отфильтровать подходящие.

Как мы помним, `regexp.match` (без флага `g`) и `str.matchAll` (всегда) возвращают совпадения со свойством `index`, которое содержит позицию совпадения в строке, так что мы можем посмотреть на контекст.

Но обычно регулярные выражения удобнее.

Виды проверок:

Шаблон	Тип	Совпадение
`X(?=Y)`	Позитивная опережающая	`X`, если за ним следует `Y`
`X(?!Y)`	Негативная опережающая	`X`, если за ним НЕ следует `Y`
`(?<=Y)X`	Позитивная ретроспективная	`X`, если следует за `Y`
`(?<!Y)X`	Негативная ретроспективная	`X`, если НЕ следует за `Y`

Задачи

Найдите неотрицательные целые

Есть строка с целыми числами.

Создайте регулярное выражение, которое ищет только неотрицательные числа. Ноль разрешён.

Пример использования:

```
let regexp = /ваше регулярное выражение/g;

let str = "0 12 -5 123 -18";
```

```
alert( str.match(regexp) ); // 0, 12, 123
```

К решению

Вставьте после фрагмента

Есть строка с HTML-документом.

Вставьте после тега `<body>` (у него могут быть атрибуты) строку `<h1>Hello</h1>`.

Например:

```
let regexp = /ваше регулярное выражение/;

let str = `
<html>
  <body style="height: 200px">
  ...
  </body>
</html>
`;

str = str.replace(regexp, `<h1>Hello</h1>`);
```

После этого значение `str`:

```
<html>
  <body style="height: 200px"><h1>Hello</h1>
  ...
  </body>
</html>
```

К решению

Катастрофический возврат

Некоторые регулярные выражения, простые с виду, могут выполняться оооочень долго, и даже «подвешивать» интерпретатор JavaScript.

Рано или поздно с этим сталкивается любой разработчик, потому что нечаянно создать такое регулярное выражение –- проще простого.

Типичный симптом: регулярное выражение обычно работает нормально, но иногда, с некоторыми строками, «подвешивает» интерпретатор и потребляет 100% процессора.

Как правило, веб-браузер при этом предлагает «убить» скрипт и перезагрузить зависшую страницу. Явно плохая ситуация.

Ну а для серверного JavaScript это может стать серьёзной уязвимостью, если регулярные выражения используются для обработки пользовательских данных.

Пример

Допустим, у нас есть строка, и мы хотим проверить, что она состоит из слов `\w+`, после каждого слова может быть пробел `\s?`.

Используем регулярное выражение `^(\w+\s?)*$`, которое задаёт 0 или более таких слов.

Проверим, чтобы убедиться, что оно работает:

```
let regexp = /^(\w+\s?)*$/;

alert( regexp.test("A good string") ); // true
alert( regexp.test("Bad characters: $@#") ); // false
```

Результат верный. Однако, на некоторых строках оно выполняется очень долго. Так долго, что интерпретатор JavaScript «зависает» с потреблением 100% процессора.

Если вы запустите пример ниже, то, скорее всего, ничего не увидите, так как JavaScript «подвиснет». В браузере он перестанет реагировать на другие события и, скорее всего, понадобится перезагрузить страницу, так что осторожно с этим:

```
let regexp = /^(\w+\s?)*$/;
let str = "An input string that takes a long time or even makes this regexp to hang

// этот поиск будет выполняться очень, очень долго
alert( regexp.test(str) );
```

Некоторые движки регулярных выражений могут справиться с таким поиском, но большинство из них – нет.

Упрощённый пример

В чём же дело? Почему регулярное выражение «зависает»?

Чтобы это понять, упростим пример: уберём из него пробелы `\s?`. Получится `^(\w+)*$`.

И, для большей наглядности, заменим `\w` на `\d`. Получившееся регулярное выражение тоже будет «зависать», например:

```
let regexp = /^(\d+)*$/;

let str = "012345678901234567890123456789!";

// этот поиск будет выполняться очень, очень долго
alert( regexp.test(str) );
```

В чём же дело, что не так с регулярным выражением?

Внимательный читатель, посмотрев на `(\d+)*`, наверняка удивится, ведь оно какое-то странное. Квантификатор `*` здесь выглядит лишним. Если хочется найти число, то с тем же успехом можно искать `\d+`.

Действительно, это регулярное выражение носит искусственный характер, но, разобравшись с ним, мы поймём и практический пример, данный выше. Причина их медленной работы одинакова. Поэтому оставим как есть.

Что же происходит во время поиска `^(\d+)*$` в строке `123456789!` (укоротим для ясности), почему всё так долго?

1. Первым делом, движок регулярных выражений пытается найти `\d+`. Плюс `+` является жадным по умолчанию, так что он хватает все цифры, какие может:

   ```
   \d+.......
   (123456789)!
   ```

   Затем движок пытается применить квантификатор `*`, но больше цифр нет, так что звёздочка ничего не даёт.

   Далее по шаблону ожидается конец строки `$`, а в тексте символ `!`, так что соответствий нет:

   ```
             X
   \d+........$
   (123456789)!
   ```

2. Так как соответствие не найдено, то «жадный» квантификатор `+` уменьшает количество повторений, возвращается на один символ назад.

   Теперь `\d+` – это все цифры, за исключением последней:

```
\d+.......
(12345678)9!
```

3. Далее движок снова пытается продолжить поиск, начиная уже с позиции (`9`).

   Звёздочка `(\d+)*` теперь может быть применена – она даёт второе число `9`:

```
\d+.......\d+
(12345678)(9)!
```

   Затем движок ожидает найти `$`, но это ему не удаётся, ведь строка оканчивается на `!`:

```
           X
\d+.......\d+
(12345678)(9)!
```

4. Так как совпадения нет, то поисковой движок продолжает отступать назад. Общее правило таково: последний жадный квантификатор уменьшает количество повторений до тех пор, пока это возможно. Затем понижается предыдущий «жадный» квантификатор и т.д.

   Перебираются все возможные комбинации. Вот их примеры.

   Когда первое число `\d+` содержит 7 цифр, а дальше число из 2 цифр:

```
           X
\d+......\d+
(1234567)(89)!
```

   Когда первое число содержит 7 цифр, а дальше два числа по 1 цифре:

```
             X
\d+......\d+\d+
(1234567)(8)(9)!
```

Когда первое число содержит 6 цифр, а дальше одно число из 3 цифр:

```
           X
\d+.......\d+
(123456)(789)!
```

Когда первое число содержит 6 цифр, а затем два числа:

```
             X
\d+.....\d+ \d+
(123456)(78)(9)!
```

…И так далее.

Существует много способов как разбить на числа набор цифр `123456789`. Если быть точным, их 2^n-1, где `n` – длина набора.

В случае `n=20` их порядка миллиона, при `n=30` – ещё в тысячу раз больше. На их перебор и тратится время.

Что же делать?

Может нам стоит использовать «ленивый» режим?

К сожалению, нет: если мы заменим `\d+` на `\d+?`, то регулярное выражение всё ещё будет «зависать». Поменяется только порядок перебора, но не общее количество комбинаций.

Некоторые движки регулярных выражений содержат хитрые проверки и конечные автоматы, которые позволяют избежать полного перебора в таких ситуациях или кардинально ускорить его, но не все движки и не всегда.

Назад к словам и строкам

В начальном примере, когда мы ищем слова по шаблону `^(\w+\s?)*$` в строке вида `An input that hangs!`, происходит то же самое.

Дело в том, что каждое слово может быть представлено как в виде одного `\w+`, так и нескольких:

```
(input)
(inpu)(t)
(inp)(u)(t)
```

```
(in)(p)(ut)
...
```

Человеку очевидно, что совпадения быть не может, так как эта строка заканчивается на восклицательный знак `!`, а по регулярному выражению в конце должен быть символ `\w` или пробел `\s`. Но движок этого не знает.

Он перебирает все комбинации того, как регулярное выражение `(\w+\s?)*` может «захватить» каждое слово, включая варианты как с пробелами `(\w+\s)*`, так и без `(\w+)*` (пробелы `\s?` ведь не обязательны). Этих вариантов очень много, отсюда и сверхдолгое время выполнения.

Как исправить?

Есть два основных подхода, как это исправить.

Первый – уменьшить количество возможных комбинаций.

Перепишем регулярное выражение так: `^(\w+\s)*\w*?` – то есть, будем искать любое количество слов с пробелом `(\w+\s)*`, после которых идёт (но не обязательно) обычное слово `\w*`.

Это регулярное выражение эквивалентно предыдущему (ищет то же самое), и на этот раз всё работает:

```
let regexp = /^(\w+\s)*\w*$/;
let str = "An input string that takes a long time or even makes this regex to hang!

alert( regexp.test(str) ); // false
```

Почему же проблема исчезла?

Теперь звёздочка `*` стоит после `\w+\s` вместо `\w+\s?`. Стало невозможно разбить одно слово на несколько разных `\w+`. Исчезли и потери времени на перебор таких комбинаций.

Например, с предыдущим шаблоном `(\w+\s?)*` слово `string` могло быть представлено как два подряд `\w+`:

```
\w+\w+
string
```

Предыдущий шаблон из-за необязательности `\s` допускал варианты `\w+`, `\w+\s`, `\w+\w+` и т.п.

С переписанным шаблоном `(\w+\s)*`, такое невозможно: может быть `\w+\s` или `\w+\s\w+\s`, но не `\w+\w+`. Так что общее количество комбинаций сильно уменьшается.

Запрет возврата

Переписывать регулярное выражение не всегда удобно, и не всегда очевидно, как это сделать.

Альтернативный подход заключается в том, чтобы запретить возврат для квантификатора.

Движок регулярных выражений проверяет множество вариантов, которые для человека являются очевидно ошибочными.

Например, в шаблоне `(\d+)*$` для человека очевидно, что в `(\d+)*` не нужно «откатывать» `+`. От того, что вместо одного `\d+` у нас будет два независимых `\d+\d+`, ничего не изменится:

```
\d+........
(123456789)!

\d+...\d+....
(1234)(56789)!
```

Если говорить об изначальном примере `^(\w+\s?)*$`, то хорошо бы исключить возврат для `\w+`. То есть, для `\w+` нужно искать только одно слово целиком, максимально возможной длины. Не нужно уменьшать количество повторений `\w+`, пробовать разбить слово на два `\w+\w+`, и т.п.

В современных регулярных выражениях для решения этой проблемы придумали захватывающие (possessive) квантификаторы, которые такие же как жадные, но не делают возврат (то есть, по сути, они даже проще, чем жадные).

Также есть «атомарные скобочные группы» – средство, запрещающее возврат внутри скобок.

К сожалению, в JavaScript они не поддерживаются, но есть другое средство.

Опережающая проверка в помощь!

Мы можем исключить возврат с помощью опережающей проверки.

Шаблон, захватывающий максимальное количество повторений `\w` без возврата, выглядит так: `(?=(\w+))\1`.

Расшифруем его:

- Опережающая проверка `?=` ищет максимальное количество `\w+`, доступных с текущей позиции.
- Содержимое скобок вокруг `?=...` не запоминается движком, поэтому оборачиваем `\w+` внутри в дополнительные скобки, чтобы движок регулярных выражений запомнил их содержимое.
- …И чтобы далее в шаблоне на него сослаться обратной ссылкой `\1`.

То есть, мы смотрим вперед – и если там есть слово `\w+`, то ищем его же `\1`.

Зачем? Всё дело в том, что опережающая проверка находит слово `\w+` целиком, и мы захватываем его в шаблон посредством `\1`. Поэтому мы реализовали, по сути, захватывающий квантификатор `+`. Такой шаблон захватывает только полностью слово `\w+`, не его часть.

Например, в слове `JavaScript` он не может захватить только `Java`, и оставить `Script` для совпадения с остатком шаблона.

Вот, посмотрите, сравнение двух шаблонов:

```
alert( "JavaScript".match(/\w+Script/)); // JavaScript
alert( "JavaScript".match(/(?=(\w+))\1Script/)); // null
```

1. В первом варианте `\w+` сначала забирает слово `JavaScript` целиком, потом `+` постепенно отступает, чтобы попробовать найти оставшуюся часть шаблона, и в конце концов находит (при этом `\w+` будет соответствовать `Java`).
2. Во втором варианте `(?=(\w+))` осуществляет опережающую проверку и видит сразу слово `JavaScript`, которое `\1` целиком захватывает в совпадение, так что уже нет возможности найти `Script`.

Внутрь `(?=(\w+))\1` можно вместо `\w` вставить и более сложное регулярное выражение, при поиске которого квантификатор `+` не должен делать возврат.

> **На заметку:**
>
> Больше о связи между захватывающих квантификаторов и опережающей проверки вы можете найти в статьях Regex: Emulate Atomic Grouping (and Possessive Quantifiers) with LookAhead и Mimicking Atomic Groups.

Перепишем исходный пример, используя опережающую проверку для запрета возврата:

```
let regexp = /^((?=(\w+))\2\s?)*$/;

alert( regexp.test("A good string") ); // true

let str = "An input string that takes a long time or even makes this regex to hang!

alert( regexp.test(str) ); // false, работает и быстро
```

Здесь внутри скобок стоит `\2` вместо `\1`, так как есть ещё внешние скобки. Чтобы избежать путаницы с номерами скобок, можно дать скобкам имя, например `(?<word>\w+)`.

```
// скобки названы ?<word>, ссылка на них \k<word>
let regexp = /^((?=(?<word>\w+))\k<word>\s?)*$/;

let str = "An input string that takes a long time or even makes this regex to hang!

alert( regexp.test(str) ); // false

alert( regexp.test("A correct string") ); // true
```

Проблему, которой была посвящена эта глава, называют «катастрофический возврат» (catastrophic backtracking).

Мы разобрали два способа её решения:

- Уменьшение возможных комбинаций переписыванием шаблона.
- Запрет возврата.

Поиск на заданной позиции, флаг "y"

Флаг `y` позволяет произвести поиск на определённой позиции в исходной строке.

Чтобы разобрать флаг `y` и понять, чем же он хорош, рассмотрим практический пример.

Одна из часто встречающихся задач регулярных выражений – лексический разбор: мы имеем текст, например, на каком-то языке программирования и получаем его структурные элементы.

Например, в HTML есть теги и атрибуты, в JavaScript-коде – переменные и функции, и т.п.

Мы не будем погружаться глубоко в тему написания таких анализаторов (это специализированная область со своим набором инструментов и алгоритмов).

Но в процессе их работы, вообще, в процессе анализа текста, очень часто возникает задача «прочитать что-то на заданной позиции».

Например, у нас есть строка кода `let varName = "value"`, и нам надо прочитать из неё имя переменной, которое начинается с позиции `4`.

Имя переменной будем искать как слово `\w+`. Вообще, в языке JavaScript для имени переменной нужно чуть более сложное регулярное выражение, но здесь это не важно.

Вызов `str.match(/\w+/)` найдёт только первое слово в строке или все слова (с флагом `g`), а нам нужно одно слово именно на позиции `4`.

Для поиска, начиная с нужной позиции, можно использовать метод `regexp.exec(str)`.

Если у регулярного выражения `regexp` нет флагов `g` или `y`, то этот метод ищет первое совпадение в строке `str`, точно так же, как `str.match(regexp)`. Здесь нас этот простейший вариант без флагов не интересует.

Если флаг `g` есть, то он осуществляет поиск в строке `str`, начиная с позиции, заданной свойством `regexp.lastIndex`. И, когда находит, обновляет `regexp.lastIndex` на позицию после совпадения.

При создании регулярного выражения его свойство `lastIndex` равно `0`.

Так что повторные вызовы `regexp.exec` возвращают совпадения по очереди, одно за другим.

Например (с флагом `g`):

```
let str = 'let varName';

let regexp = /\w+/g;
alert(regexp.lastIndex); // 0 (при создании lastIndex=0)

let word1 = regexp.exec(str);
alert(word1[0]); // let (первое слово)
alert(regexp.lastIndex); // 3 (позиция за первым совпадением)

let word2 = regexp.exec(str);
alert(word2[0]); // varName (второе слово)
alert(regexp.lastIndex); // 11 (позиция за вторым совпадением)

let word3 = regexp.exec(str);
alert(word3); // null (больше совпадений нет)
alert(regexp.lastIndex); // 0 (сбрасывается по окончании поиска)
```

Заметим, что каждое совпадение возвращается в виде массива, со всеми скобочными группами и дополнительными свойствами.

Можно перебрать все совпадения в цикле:

```
let str = 'let varName';
let regexp = /\w+/g;

let result;

while (result = regexp.exec(str)) {
  alert( `Найдено ${result[0]} на позиции ${result.index}` );
  // Найдено let на позиции 0, затем
  // Найдено varName на позиции 4
}
```

Такое использование `regexp.exec` представляет собой альтернативу методу `str.matchAll`.

Таким образом, последовательные вызовы `regexp.exec` могут найти все совпадения, представляя собой альтернативу методам `str.match/matchAll`.

Но, в отличие от других методов, мы можем поставить самостоятельно `lastIndex`, начав тем самым поиск именно с нужной позиции.

Например, найдём слово, начиная с позиции `4`:

```
let str = 'let varName = "value"';

let regexp = /\w+/g; // без флага g свойство lastIndex игнорируется

regexp.lastIndex = 4;

let word = regexp.exec(str);
alert(word); // varName
```

Поиск `\w+` произведён, начиная с позиции `regexp.lastIndex = 4`.

Заметим, что такой поиск лишь начинается с позиции `lastIndex` и идёт дальше. Если слова на позиции `lastIndex` нет, но оно есть позже, оно всё равно будет найдено:

```
let str = 'let varName = "value"';

let regexp = /\w+/g;
```

```
regexp.lastIndex = 3;

let word = regexp.exec(str);
alert(word[0]); // varName
alert(word.index); // 4
```

…То есть, при флаге `g` свойство `lastIndex` задаёт начальную позицию поиска.

Флаг `y` заставляет `regexp.exec` искать ровно на позиции `lastIndex`, ни до и ни после.

Вот тот же поиск с флагом `y`:

```
let str = 'let varName = "value"';

let regexp = /\w+/y;

regexp.lastIndex = 3;
alert( regexp.exec(str) ); // null (на позиции 3 пробел, а не слово)

regexp.lastIndex = 4;
alert( regexp.exec(str) ); // varName (слово на позиции 4)
```

Как можно видеть, регулярное выражение `/\w+/y` не найдено на позиции `3` (в отличие от флага `g`), но найдено на позиции `4`.

Представим себе, что у нас большой текст, и в нём нет ни одного совпадения. В таком случае регэксп с флагом `g` будет идти до самого конца текста, и это займёт гораздо больше времени, чем поиск с флагом `y`.

В задачах, подобных лексическому анализу, обычно много поисков на конкретной позиции. Использование флага `y` – ключ к хорошей производительности.

Методы RegExp и String

В этой главе мы рассмотрим все детали методов для работы с регулярными выражениями.

str.match(regexp)

Метод `str.match(regexp)` ищет совпадения с `regexp` в строке `str`.

У него есть три режима работы:

1. Если у регулярного выражения нет флага `g`, то он возвращает первое совпадение в виде массива со скобочными группами и свойствами `index` (позиция совпадения), `input` (строка поиска, равна `str`):

```
let str = "I love JavaScript";

let result = str.match(/Java(Script)/);

alert( result[0] );     // JavaScript (всё совпадение)
alert( result[1] );     // Script (первые собки)
alert( result.length ); // 2

// Дополнительная информация:
alert( result.index );  // 0 (позиция совпадения)
alert( result.input );  // I love JavaScript (исходная строка)
```

2. Если у регулярного выражения есть флаг `g`, то он возвращает массив всех совпадений, без скобочных групп и других деталей.

```
let str = "I love JavaScript";

let result = str.match(/Java(Script)/g);

alert( result[0] ); // JavaScript
alert( result.length ); // 1
```

3. Если совпадений нет, то, вне зависимости от наличия флага `g`, возвращается `null`.

   Это очень важный нюанс. При отсутствии совпадений возвращается не пустой массив, а именно `null`. Если об этом забыть, можно легко допустить ошибку, например:

```
let str = "I love JavaScript";

let result = str.match(/HTML/);

alert(result); // null
alert(result.length); // Ошибка: у null нет свойства length
```

   Если хочется, чтобы результатом всегда был массив, можно написать так:

```
let result = str.match(regexp) || [];
```

str.matchAll(regexp)

> **Новая возможность**
>
> Эта возможность была добавлена в язык недавно. В старых браузерах может понадобиться полифил.

Метод `str.matchAll(regexp)` – «новый, улучшенный» вариант метода `str.match`.

Он используется, в первую очередь, для поиска всех совпадений вместе со скобочными группами.

У него 3 отличия от `match`:

1. Он возвращает не массив, а перебираемый объект с результатами, обычный массив можно сделать при помощи `Array.from`.
2. Каждое совпадение возвращается в виде массива со скобочными группами (как `str.match` без флага `g`).
3. Если совпадений нет, то возвращается не `null`, а пустой перебираемый объект.

Пример использования:

```
let str = '<h1>Hello, world!</h1>';
let regexp = /<(.*?)>/g;

let matchAll = str.matchAll(regexp);

alert(matchAll); // [object RegExp String Iterator], не массив, а перебираемый объе

matchAll = Array.from(matchAll); // теперь массив

let firstMatch = matchAll[0];
alert( firstMatch[0] );  // <h1>
alert( firstMatch[1] );  // h1
alert( firstMatch.index );  // 0
alert( firstMatch.input );  // <h1>Hello, world!</h1>
```

При переборе результатов `matchAll` в цикле `for..of` вызов `Array.from`, разумеется, не нужен.

str.split(regexp|substr, limit)

Разбивает строку в массив по разделителю – регулярному выражению regexp или подстроке substr.

Обычно мы используем метод `split` со строками, вот так:

```
alert('12-34-56'.split('-')) // массив [12, 34, 56]
```

Но мы можем разделить по регулярному выражению аналогичным образом:

```
alert('12, 34, 56'.split(/,\s*/)) // массив [12, 34, 56]
```

str.search(regexp)

Метод `str.search(regexp)` возвращает позицию первого совпадения с `regexp` в строке `str` или `-1`, если совпадения нет.

Например:

```
let str = "Я люблю JavaScript!";

let regexp = /Java.+/;

alert( str.search(regexp) ); // 8
```

Важное ограничение: `str.search` умеет возвращать только позицию первого совпадения.

Если нужны позиции других совпадений, то следует использовать другой метод, например, найти их все при помощи `str.matchAll(regexp)`.

str.replace(str|regexp, str|func)

Это универсальный метод поиска-и-замены, один из самых полезных. Этакий швейцарский армейский нож для поиска и замены в строке.

Мы можем использовать его и без регулярных выражений, для поиска-и-замены подстроки:

```
// заменить тире двоеточием
alert('12-34-56'.replace("-", ":")) // 12:34-56
```

Хотя есть подводный камень.

Когда первый аргумент `replace` является строкой, он заменяет только первое совпадение.

Вы можете видеть это в приведённом выше примере: только первый `"-"` заменяется на `":"`.

Чтобы найти все дефисы, нам нужно использовать не строку `"-"`, а регулярное выражение `/-/g` с обязательным флагом `g`:

```
// заменить все тире двоеточием
alert( '12-34-56'.replace( /-/g, ":" ))  // 12:34:56
```

Второй аргумент – строка замены. Мы можем использовать специальные символы в нем:

Спецсимволы	Действие в строке замены
`$$`	вставляет `"$"`
`$&`	вставляет всё найденное совпадение
`$``	вставляет часть строки до совпадения
`$'`	вставляет часть строки после совпадения
`$n`	если `n` это 1-2 значное число, то вставляет содержимое n-й скобки
`$<имя>`	вставояет содержимое скобки с указанным именем

Например:

```
let str = "John Smith";

// поменять местами имя и фамилию
alert(str.replace(/(\w+) (\w+)/i, '$2, $1')) // Smith, John
```

Для ситуаций, которые требуют «умных» замен, вторым аргументом может быть функция.

Она будет вызываться для каждого совпадения, и её результат будет вставлен в качестве замены.

Функция вызывается с аргументами `func(match, p1, p2, ..., pn, offset, input, groups)`:

1. `match` – найденное совпадение,
2. `p1, p2, ..., pn` – содержимое скобок (см. главу Скобочные группы).
3. `offset` – позиция, на которой найдено совпадение,
4. `input` – исходная строка,

5. `groups` – объект с содержимым именованных скобок (см. главу Скобочные группы).

Если скобок в регулярном выражении нет, то будет только 3 аргумента: `func(match, offset, input)`.

Например, переведём выбранные совпадения в верхний регистр:

```
let str = "html and css";

let result = str.replace(/html|css/gi, str => str.toUpperCase());

alert(result); // HTML and CSS
```

Заменим каждое совпадение на его позицию в строке:

```
alert("Xo-Xo-xo".replace(/xo/gi, (match, offset) => offset)); // 0-3-6
```

В примере ниже двое скобок, поэтому функция замены вызывается с 5-ю аргументами: первый – всё совпадение, затем два аргумента содержимое скобок, затем (в примере не используются) индекс совпадения и исходная строка:

```
let str = "John Smith";

let result = str.replace(/(\w+) (\w+)/, (match, name, surname) => `${surname}, ${na

alert(result); // Smith, John
```

Если в регулярном выражении много скобочных групп, то бывает удобно использовать остаточные аргументы для обращения к ним:

```
let str = "John Smith";

let result = str.replace(/(\w+) (\w+)/, (...match) => `${match[2]}, ${match[1]}`);

alert(result); // Smith, John
```

Или, если мы используем именованные группы, то объект `groups` с ними всегда идёт последним, так что можно получить его так:

```
let str = "John Smith";

let result = str.replace(/(?<name>\w+) (?<surname>\w+)/, (...match) => {
  let groups = match.pop();

  return `${groups.surname}, ${groups.name}`;
});

alert(result); // Smith, John
```

Использование функции даёт нам максимальные возможности по замене, потому что функция получает всю информацию о совпадении, имеет доступ к внешним переменным и может делать всё, что угодно.

regexp.exec(str)

Метод `regexp.exec(str)` ищет совпадение с `regexp` в строке `str`. В отличие от предыдущих методов, вызывается на регулярном выражении, а не на строке.

Он ведёт себя по-разному в зависимости от того, имеет ли регулярное выражение флаг `g`.

Если нет `g`, то `regexp.exec(str)` возвращает первое совпадение в точности как `str.match(regexp)`. Такое поведение не даёт нам ничего нового.

Но если есть `g`, то:

- Вызов `regexp.exec(str)` возвращает первое совпадение и *запоминает* позицию после него в свойстве `regexp.lastIndex`.
- Следующий такой вызов начинает поиск с позиции `regexp.lastIndex`, возвращает следующее совпадение и запоминает позицию после него в `regexp.lastIndex`.
- …И так далее.
- Если совпадений больше нет, то `regexp.exec` возвращает `null`, а для `regexp.lastIndex` устанавливается значение `0`.

Таким образом, повторные вызовы возвращают одно за другим все совпадения, используя свойство `regexp.lastIndex` для отслеживания текущей позиции поиска.

В прошлом, до появления метода `str.matchAll` в JavaScript, вызов `regexp.exec` в цикле использовали для получить всех совпадений с их позициями и группами скобок в цикле:

```
let str = 'Больше о JavaScript на https://javascript.info';
let regexp = /javascript/ig;

let result;

while (result = regexp.exec(str)) {
  alert( `Найдено ${result[0]} на позиции ${result.index}` );
  // Найдено JavaScript на позиции 9, затем
  // Найдено javascript на позиции 31
}
```

Это работает и сейчас, хотя для современных браузеров `str.matchAll`, как правило, удобнее.

Мы можем использовать `regexp.exec` для поиска совпадения, начиная с нужной позиции, если вручную поставим `lastIndex`.

Например:

```
let str = 'Hello, world!';

let regexp = /\w+/g; // без флага g свойство lastIndex игнорируется
regexp.lastIndex = 5; // ищем с 5-й позиции (т.е с запятой и далее)

alert( regexp.exec(str) ); // world
```

Если у регулярного выражения стоит флаг `y`, то поиск будет вестись не начиная с позиции `regexp.lastIndex`, а только на этой позиции (не далее в тексте).

В примере выше заменим флаг `g` на `y`. Ничего найдено не будет, поскольку именно на позиции `5` слова нет:

```
let str = 'Hello, world!';

let regexp = /\w+/y;
regexp.lastIndex = 5; // ищем ровно на 5-й позиции

alert( regexp.exec(str) ); // null
```

Это удобно в тех ситуациях, когда мы хотим «прочитать» что-то из строки по регулярному выражению именно на конкретной позиции, а не где-то далее.

regexp.test(str)

Метод `regexp.test(str)` ищет совпадение и возвращает `true/false`, в зависимости от того, находит ли он его.

Например:

```
let str = "Я люблю JavaScript";

// эти два теста делают одно и же
alert( /люблю/i.test(str) ); // true
alert( str.search(/люблю/i) != -1 ); // true
```

Пример с отрицательным ответом:

```
let str = "Ля-ля-ля";

alert( /люблю/i.test(str) ); // false
alert( str.search(/люблю/i) != -1 ); // false
```

Если регулярное выражение имеет флаг `g`, то `regexp.test` ищет, начиная с `regexp.lastIndex` и обновляет это свойство, аналогично `regexp.exec`.

Таким образом, мы можем использовать его для поиска с заданной позиции:

```
let regexp = /люблю/gi;

let str = "Я люблю JavaScript";

// начать поиск с 10-й позиции:
regexp.lastIndex = 10;
alert( regexp.test(str) ); // false (совпадений нет)
```

⚠️ **Одно и то же регулярное выражение, использованное повторно на другом тексте, может дать другой результат**

Если мы применяем одно и то же регулярное выражение последовательно к разным строкам, это может привести к неверному результату, поскольку вызов `regexp.test` обновляет свойство `regexp.lastIndex`, поэтому поиск в новой строке может начаться с ненулевой позиции.

Например, здесь мы дважды вызываем `regexp.test` для одного и того же текста, и второй раз поиск завершается уже неудачно:

```
let regexp = /javascript/g;  // (regexp только что создан: regexp.lastIndex=0)

alert( regexp.test("javascript") ); // true (теперь regexp.lastIndex=10)
alert( regexp.test("javascript") ); // false
```

Это именно потому, что во втором тесте `regexp.lastIndex` не равен нулю.

Чтобы обойти это, можно присвоить `regexp.lastIndex = 0` перед новым поиском. Или вместо методов на регулярном выражении вызывать методы строк `str.match/search/...`, они не используют `lastIndex`.

CSS для JavaScript-разработчика

О чём пойдёт речь

Неужели мы сейчас будем учить CSS? Ничего подобного. Предполагается, что вы *уже* знаете CSS, во всяком случае понимаете его на таком уровне, который позволяет делать Web-страницы.

Особенность квалификации JavaScript-разработчика заключается в том, что он не обязан выбирать цвета, рисовать иконки, «делать красиво». Он также не обязан верстать макет в HTML, разве что если является по совместительству специалистом-верстальщиком.

Вот что он должен уметь абсолютно точно – так это и разработать такую структуру HTML/CSS для элементов управления, которая не сломается, и с которой ему же потом удобно будет взаимодействовать.

Это требует отличного знания CSS в области позиционирования элементов, включая тонкости работы `display`, `margin`, `border`, `outline`, `position`, `float`, `border-box` и остальных свойств, а также подходы к построению структуры компонент (CSS layouts).

Многое можно сделать при помощи JavaScript. И зачастую, не зная CSS, так и делают. Но мы на это ловиться не будем.

> Если что-то можно сделать через CSS – лучше делать это через CSS.

Причина проста – обычно, даже если CSS на вид сложнее – поддерживать и развивать его проще, чем JS. Поэтому овчинка стоит выделки.

Кроме того, есть ещё одно наблюдение.

> Знание JavaScript не может заменить знание CSS.

Жить становится приятнее и проще, если есть хорошее знание и CSS, и JavaScript.

Чек-лист

Ниже находится «чек-лист». Если хоть одно свойство незнакомо – это стоп-сигнал для дальнейшего чтения этого раздела.

- Блочная модель, включая:
  - `margin`, `padding`, `border`, `overflow`
  - а также `height/width` и `min-height/min-width`.
- Текст:
  - `font`
  - `line-height`.
- Различные курсоры `cursor`.
- Позиционирование:
  - `position`, `float`, `clear`, `display`, `visibility`
  - Центрирование при помощи CSS
  - Перекрытие `z-index` и прозрачность `opacity`
- Селекторы:
  - Приоритет селекторов
  - Селекторы `#id`, `.class`, `a > b`
- Сброс браузерных стилей, reset.css

Почитать

Книжек много, но хороших – как всегда, мало.

С уверенностью могу рекомендовать следующие:

- Большая книга CSS3. *Дэвид Макфарланд.*
- CSS. Каскадные таблицы стилей. Подробное руководство. *Эрик Мейер*

Дальнейшие статьи раздела не являются *учебником* CSS, поэтому пожалуйста, изучите эту технологию до них.

Это очерки для лучшей систематизации и дополнения уже существующих знаний.

Единицы измерения: px, em, rem и другие

В этом очерке я постараюсь не только рассказать о различных единицах измерения, но и построить общую картину – что и когда выбирать.

Пиксели: px

Пиксель `px` – это самая базовая, абсолютная и окончательная единица измерения.

Количество пикселей задаётся в настройках разрешения экрана, один `px` – это как раз один такой пиксель на экране. Все значения браузер в итоге пересчитает в пиксели.

Пиксели могут быть дробными, например размер можно задать в `16.5px`. Это совершенно нормально, браузер сам использует дробные пиксели для внутренних вычислений. К примеру, есть элемент шириной в `100px`, его нужно разделить на три части – волей-неволей появляются `33.333...px`. При окончательном отображении дробные пиксели, конечно же, округляются и становятся целыми.

Для мобильных устройств, у которых много пикселей на экране, но сам экран маленький, чтобы обеспечить читаемость, браузер автоматически применяет масштабирование.

Достоинства

- Главное достоинство пикселя – чёткость и понятность

Недостатки

- Другие единицы измерения – в некотором смысле «мощнее», они являются относительными и позволяют устанавливать соотношения

	между различными размерами

Давно на свалке: `mm`, `cm`, `pt`, `pc`

Существуют также «производные» от пикселя единицы измерения: `mm`, `cm`, `pt` и `pc`, но они давно отправились на свалку истории.

Вот, если интересно, их значения:

- `1mm` (мм) = `3.8px`
- `1cm` (см) = `38px`
- `1pt` (типографский пункт) = `4/3 px`
- `1pc` (типографская пика) = `16px`

Так как браузер пересчитывает эти значения в пиксели, то смысла в их употреблении нет.

Почему в сантиметре `cm` содержится ровно 38 пикселей?

В реальной жизни сантиметр – это эталон длины, одна сотая метра. А пиксель может быть разным, в зависимости от экрана.

Но в формулах выше под пикселем понимается «сферический пиксель в вакууме», точка на «стандартизованном экране», характеристики которого описаны в спецификации.

Поэтому ни о каком соответствии `cm` реальному сантиметру здесь нет и речи. Это полностью синтетическая и производная единица измерения, которая не нужна.

Относительно шрифта: em

`1em` – текущий размер шрифта.

Можно брать любые пропорции от текущего шрифта: `2em`, `0.5em` и т.п.

Размеры в `em` – *относительные*, они определяются по текущему контексту.

Например, давайте сравним `px` с `em` на таком примере:

```
<div style="font-size:24px">
  Страусы
  <div style="font-size:24px">Живут также в Африке</div>
</div>
```

Страусы
Живут также в Африке

`24` пикселей – и в Африке `24` пикселей, поэтому размер шрифта в `<div>` одинаков.

А вот аналогичный пример с `em` вместо `px`:

```
<div style="font-size:1.5em">
  Страусы
  <div style="font-size:1.5em">Живут также в Африке</div>
</div>
```

Страусы
Живут также в Африке

Так как значение в `em` высчитывается относительно *текущего шрифта*, то вложенная строка в `1.5` раза больше, чем первая.

Выходит, размеры, заданные в `em`, будут уменьшаться или увеличиваться вместе со шрифтом. С учётом того, что размер шрифта обычно определяется в родителе, и может быть изменён ровно в одном месте, это бывает очень удобно.

Что такое размер шрифта?

Что такое «размер шрифта»? Это вовсе не «размер самой большой буквы в нём», как можно было бы подумать.

Размер шрифта – это некоторая «условная единица», которая встроена в шрифт.

Она обычно чуть больше, чем расстояние от верха самой большой буквы до низа самой маленькой. То есть, предполагается, что в эту высоту помещается любая буква или их сочетание. Но при этом «хвосты» букв, таких как `p`, `g` могут заходить за это значение, то есть вылезать снизу. Поэтому обычно высоту строки делают чуть больше, чем размер шрифта.

Единицы `ex` и `ch`

В спецификации указаны также единицы ex и ch, которые означают размер символа `"x"` и размер символа `"0"`.

Эти размеры присутствуют в шрифте всегда, даже если по коду этих символов в шрифте находятся другие значения, а не именно буква `"x"` и ноль `"0"`. В этом случае они носят более условный характер.

Эти единицы используются чрезвычайно редко, так как «размер шрифта» `em` обычно вполне подходит.

Проценты %

Проценты `%`, как и `em` – относительные единицы.

Когда мы говорим «процент», то возникает вопрос – «Процент от чего?»

Как правило, процент будет от значения свойства родителя с тем же названием, но не всегда.

Это очень важная особенность процентов, про которую, увы, часто забывают.

Отличный источник информации по этой теме – стандарт, Visual formatting model details.

Вот пример с `%`, он выглядит в точности так же, как с `em`:

```
<div style="font-size:150%">
  Страусы
  <div style="font-size:150%">Живут также в Африке</div>
</div>
```

Страусы

Живут также в Африке

В примере выше процент берётся от размера шрифта родителя.

А вот примеры-исключения, в которых `%` берётся не так:

`margin-left`

При установке свойства `margin-left` в `%`, процент берётся от *ширины* родительского блока, а вовсе не от его `margin-left`.

`line-height`

При установке свойства `line-height` в `%`, процент берётся от текущего *размера шрифта*, а вовсе не от `line-height` родителя. Детали по `line-height` и размеру шрифта вы также можете найти в статье Свойства font-size и line-height.

`width/height`

Для `width/height` обычно процент от ширины/высоты родителя, но при `position:fixed`, процент берётся от ширины/высоты *окна* (а не родителя и не документа). Кроме того, иногда `%` требует соблюдения дополнительных условий, за примером – обратитесь к главе Особенности свойства height в %.

Единица rem: смесь px и em

Итак, мы рассмотрели:

- `px` – абсолютные, чёткие, понятные, не зависящие ни от чего.
- `em` – относительно размера шрифта.
- `%` – относительно такого же свойства родителя (а может и не родителя, а может и не такого же – см. примеры выше).

Может быть, пора уже остановиться, может этого достаточно?

Э-э, нет! Не все вещи делаются удобно.

Вернёмся к теме шрифтов. Бывают задачи, когда мы хотим сделать на странице большие кнопки «Шрифт больше» и «Шрифт меньше». При нажатии на них будет срабатывать JavaScript, который будет увеличивать или уменьшать шрифт.

Вообще-то это можно сделать без JavaScript, в браузере обычно есть горячие клавиши для масштабирования вроде `Ctrl+`, но они работают слишком тупо – берут и увеличивают всю страницу, вместе с изображениями и другими элементами, которые масштабировать как раз не надо. А если надо увеличить только шрифт, потому что посетитель хочет комфортнее читать?

Какую единицу использовать для задания шрифтов? Наверно не `px`, ведь значения в `px` абсолютны, если менять, то во всех стилевых правилах. Вполне возможна ситуация, когда мы в одном правиле размер поменяли, а другое забыли.

Следующие кандидаты – `em` и `%`.

Разницы между ними здесь нет, так как при задании `font-size` в процентах, эти проценты берутся от `font-size` родителя, то есть ведут себя так же, как и `em`.

Вроде бы, использовать можно, однако есть проблема.

Попробуем использовать этот подход для `<li>`.

Протестируем на таком списке:

```
<ul>
<li>Собака
  <ul>
  <li>бывает
    <ul>
    <li>кусачей
      <ul>
      <li>только
        <ul>
        <li>от жизни
          <ul>
          <li>собачей</li>
          </ul>
        </li>
        </ul>
      </li>
      </ul>
    </li>
    </ul>
  </li>
  </ul>
</li>
</ul>
```

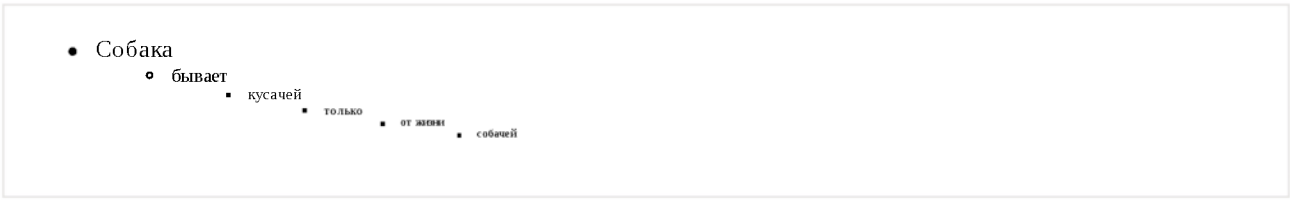
- Собака
  - бывает
    - кусачей
      - только
        - от жизни
          - собачей

Пока это обычный вложенный список.

Теперь уменьшим размер шрифта до `0.8em`, вот что получится:

```
<style>
  li {
    font-size: 0.8em;
  }
</style>

<ul>
<li>Собака
  <ul>
```

```
  <li>бывает
    <ul>
    <li>кусачей
      <ul>
      <li>только
        <ul>
        <li>от жизни
          <ul>
          <li>собачей</li>
          </ul>
        </li>
        </ul>
      </li>
      </ul>
    </li>
    </ul>
  </li>
  </ul>
</li>
</ul>
```



Проблема очевидна. Хотели, как лучше, а получилось… Мелковато. Каждый вложенный `<li>` получил размер шрифта `0.8` от родителя, в итоге уменьшившись до нечитаемого состояния. Это не совсем то, чего мы бы здесь хотели.

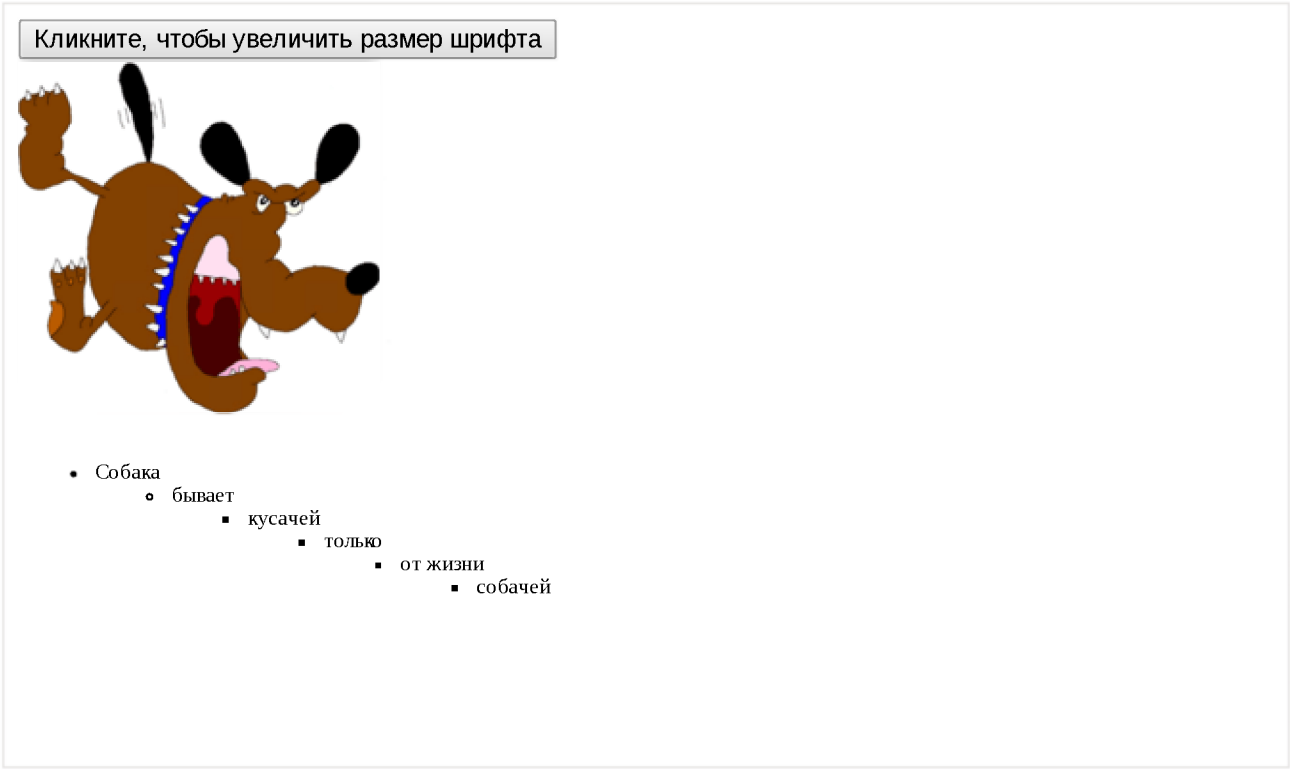
Можно уменьшить размер шрифта только на одном «корневом элементе»… Или воспользоваться единицей `rem`, которая, можно сказать, специально придумана для таких случаев!

Единица `rem` задаёт размер относительно размера шрифта элемента `<html>`.

Как правило, браузеры ставят этому элементу некоторый «разумный» (reasonable) размер по-умолчанию, который мы, конечно, можем переопределить и использовать `rem` для задания шрифтов внутри относительно него:

```
<style>
  html {
    font-size: 14px;
  }
  li {
    font-size: 0.8rem;
  }
```

```
</style>

<div><button id="up">Кликните, чтобы увеличить размер шрифта</button></div>

<img src="https://js.cx/clipart/angry_dog.png">

<ul>
<li>Собака
  <ul>
  <li>бывает
    <ul>
    <li>кусачей
      <ul>
      <li>только
        <ul>
        <li>от жизни
          <ul>
          <li>собачей</li>
          </ul>
        </li>
        </ul>
      </li>
      </ul>
    </li>
    </ul>
  </li>
  </ul>
</li>
</ul>

<script>
let html = document.documentElement;
up.onclick = function() {
  // при помощи JS увеличить размер шрифта html на 2px
  html.style.fontSize = parseInt(getComputedStyle(html, '').fontSize) + 2 + 'px';
};
</script>
```



Получилось удобное масштабирование для шрифтов, не влияющее на другие элементы.

Элементы, размер которых задан в `rem`, не зависят друг от друга и от контекста – и этим похожи на `px`, а с другой стороны они все заданы относительно размера шрифта `<html>`.

Единица `rem` не поддерживается в IE8-.

Относительно экрана: vw, vh, vmin, vmax

Во всех современных браузерах, исключая IE8-, поддерживаются новые единицы из черновика стандарта CSS Values and Units 3 :

- `vw` – 1% ширины окна
- `vh` – 1% высоты окна
- `vmin` – наименьшее из (`vw`, `vh`), в IE9 обозначается `vm`
- `vmax` – наибольшее из (`vw`, `vh`)

Эти значения были созданы, в первую очередь, для поддержки мобильных устройств.

Их основное преимущество – в том, что любые размеры, которые в них заданы, автоматически масштабируются при изменении размеров окна.

Итого

Мы рассмотрели единицы измерения:

- `px` – абсолютные пиксели, к которым привязаны и потому не нужны `mm`, `cm`, `pt` и `pc`. Используется для максимально конкретного и точного задания размеров.
- `em` – задаёт размер относительно шрифта родителя, можно относительно конкретных символов: `"x"` (`ex`) и `"0"` (`ch`), используется там, где нужно упростить масштабирование компоненты.
- `rem` – задаёт размер относительно шрифта `<html>`, используется для удобства глобального масштабирования: элементы которые планируется масштабировать, задаются в `rem`, а JS меняет шрифт у `<html>`.
- `%` – относительно такого же свойства родителя (как правило, но не всегда), используется для ширин, высот и так далее, без него никуда, но надо знать, относительно чего он считает проценты.
- `vw`, `vh`, `vmin`, `vmax` – относительно размера экрана.

Все значения свойства display

Свойство `display` имеет много разных значений. Обычно, используются только три из них: `none`, `inline` и `block`, потому что когда-то браузеры другие не поддерживали.

Но после ухода IE7-, стало возможным использовать и другие значения тоже. Рассмотрим здесь весь список.

Значение none

Самое простое значение. Элемент не показывается, вообще. Как будто его и нет.

```
<div style="border:1px solid black">
Невидимый div (
  <div style="display: none">Я - невидим!</div>
) Стоит внутри скобок
</div>
```

Невидимый div ( ) Стоит внутри скобок

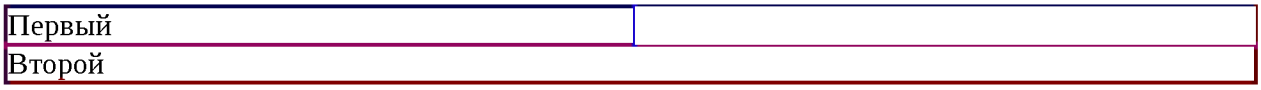
Значение block

- Блочные элементы располагаются один над другим, вертикально (если нет особых свойств позиционирования, например `float`).

- Блок стремится расшириться на всю доступную ширину. Можно указать ширину и высоту явно.

Это значение `display` многие элементы имеют по умолчанию: `<div>`, заголовок `<h1>`, параграф `<p>`.

```
<div style="border:1px solid black">
  <div style="border:1px solid blue; width: 50%">Первый</div>
  <div style="border:1px solid red">Второй</div>
</div>
```



Блоки прилегают друг к другу вплотную, если у них нет `margin`.

Значение inline

- Элементы располагаются на той же строке, последовательно.
- Ширина и высота элемента определяются по содержимому. Поменять их нельзя.

Например, инлайновые элементы по умолчанию: `<span>`, `<a>`.

```
<span style="border:1px solid black">
  <span style="border:1px solid blue; width:50%">Ширина</span>
  <a style="border:1px solid red">Игнорируется</a>
</span>
```

Если вы присмотритесь внимательно к примеру выше, то увидите, что между внутренними `<span>` и `<a>` есть пробел. Это потому, что он есть в HTML.

Если расположить элементы вплотную – его не будет:

```
<span style="border:1px solid black">
  <span style="border:1px solid blue; width:50%">Без</span><a style="border:1px sol
</span>
```

Содержимое инлайн-элемента может переноситься на другую строку.

При этом каждая строка в смысле отображения является отдельным прямоугольником («line box»). Так что инлайн-элемент состоит из объединения прямоугольников, но в целом, в отличие от блока, прямоугольником не является.

Это проявляется, например, при назначении фона.

Например, три прямоугольника подряд:

```
<div style="width:400px">
...<span style="background: lightgreen">
  Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля
  Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля
</span>...
</div>
```

... Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля
Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля Ля
Ля Ля Ля Ля Ля ...

Если инлайн-элемент граничит с блоком, то между ними обязательно будет перенос строки:

```
<div style="border:1px solid black">
  <span style="border:1px solid red">Инлайн</span>
  <div style="border:1px solid blue; width:50%">Блок</div>
  <span style="border:1px solid red">Инлайн</span>
</div>
```

Инлайн
Блок
Инлайн

Значение inline-block

Это значение – означает элемент, который продолжает находиться в строке (`inline`), но при этом может иметь важные свойства блока.

Как и инлайн-элемент:

- Располагается в строке.
- Размер устанавливается по содержимому.

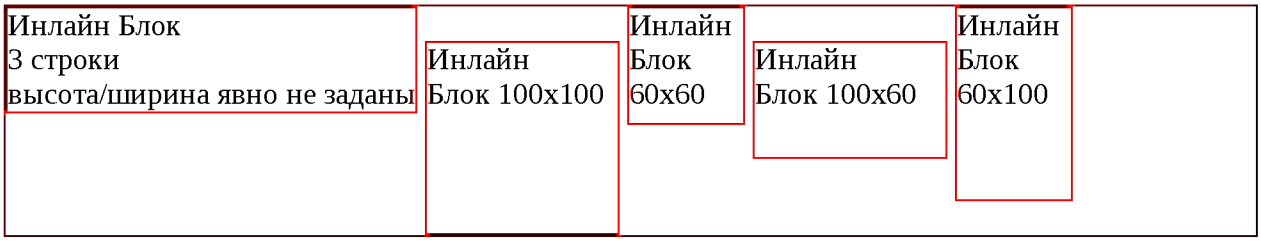
Во всём остальном – это блок, то есть:

- Элемент всегда прямоугольный.
- Работают свойства `width/height`.

Это значение `display` используют, чтобы отобразить в одну строку блочные элементы, в том числе разных размеров.

Например:

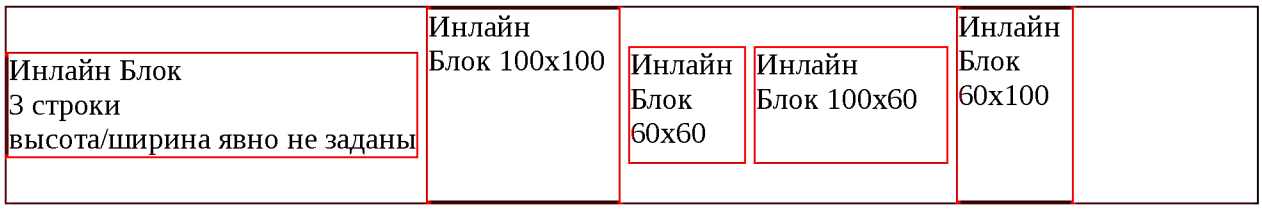
```
<style>
li {
  display: inline-block;
  list-style: none;
  border: 1px solid red;
}
</style>

<ul style="border:1px solid black; padding:0">
  <li>Инлайн Блок<br>3 строки<br>высота/ширина явно не заданы</li>
  <li style="width:100px;height:100px">Инлайн<br>Блок 100x100</li>
  <li style="width:60px;height:60px">Инлайн<br>Блок 60x60</li>
  <li style="width:100px;height:60px">Инлайн<br>Блок 100x60</li>
  <li style="width:60px;height:100px">Инлайн<br>Блок 60x100</li>
</ul>
```



Свойство `vertical-align` позволяет выровнять такие элементы внутри внешнего блока:

```
<style>
li {
  display: inline-block;
  list-style: none;
  border:1px solid red;
  vertical-align: middle;
}
</style>

<ul style="border:1px solid black; padding:0">
  <li>Инлайн Блок<br>3 строки<br>высота/ширина явно не заданы</li>
  <li style="width:100px;height:100px">Инлайн<br>Блок 100x100</li>
  <li style="width:60px;height:60px">Инлайн<br>Блок 60x60</li>
  <li style="width:100px;height:60px">Инлайн<br>Блок 100x60</li>
```

```
  <li style="width:60px;height:100px">Инлайн<br>Блок 60x100</li>
</ul>
```



Как и в случае с инлайн-элементами, пробелы между блоками появляются из-за пробелов в HTML. Если элементы списка идут вплотную, например, генерируются в JavaScript – их не будет.

Значения table-*

Современные браузеры (IE8+) позволяют описывать таблицу любыми элементами, если поставить им соответствующие значения `display`.

Для таблицы целиком `table`, для строки – `table-row`, для ячейки – `table-cell` и т.д.

Пример использования:

```
<form style="display: table">
  <div style="display: table-row">
    <label style="display: table-cell">Имя:</label>
    <input style="display: table-cell">
  </div>
  <div style="display: table-row">
    <label style="display: table-cell">Фамилия:</label>
    <input style="display: table-cell">
  </div>
</form>
```

Важно то, что это действительно полноценная таблица. Используются табличные алгоритмы вычисления ширины и высоты элемента, описанные в стандарте.

Это хорошо для семантической вёрстки и позволяет избавиться от лишних тегов.

С точки зрения современного CSS, обычные `<table>`, `<tr>`, `<td>` и т.д. – это просто элементы с предопределёнными значениями `display`:

```
table    { display: table }
tr       { display: table-row }
thead    { display: table-header-group }
tbody    { display: table-row-group }
tfoot    { display: table-footer-group }
col      { display: table-column }
colgroup { display: table-column-group }
td, th   { display: table-cell }
caption  { display: table-caption }
```

Очень подробно об алгоритмах вычисления размеров и отображении таблиц рассказывает стандарт CSS 2.1 – Tables .

Вертикальное центрирование с table-cell

Внутри ячеек свойство vertical-align выравнивает содержимое по вертикали.

Это можно использовать для центрирования:

```
<style>
  div { border:1px solid black }
</style>

<div style="height:100px; display: table-cell; vertical-align: middle">
  <div>Элемент<br>С неизвестной<br>Высотой</div>
</div>
```

CSS не требует, чтобы вокруг `table-cell` была структура таблицы: `table-row` и т.п. Может быть просто такой одинокий `DIV`, это допустимо.

При этом он ведёт себя как ячейка `TD`, то есть подстраивается под размер содержимого и умеет вертикально центрировать его при помощи `vertical-align`.

Значения list-item, run-in и flex

У свойства `display` есть и другие значения. Они используются реже, поэтому посмотрим на них кратко:

`list-item`

Этот display по умолчанию используется для элементов списка. Он добавляет к блоку содержимым ещё и блок с номером(значком) списка, который стилизуется стандартными списочными свойствами:

```
<div style="display: list-item; list-style:inside square">Пункт 1</div>
```

- Пункт 1

`run-in`

Если после `run-in` идёт `block`, то `run-in` становится его первым инлайн-элементом, то есть отображается в начале `block`.

Если ваш браузер поддерживает это значение, то в примере ниже `h3`, благодаря `display:run-in`, окажется визуально внутри `div`:

```
<h3 style="display: run-in; border:2px solid red">Про пчёл.</h3>
<div style="border:2px solid black">Пчёлы - отличные создания, они делают мёд.</div
```

Про пчёл.

Пчёлы - отличные создания, они делают мёд.

Если же вы видите две строки, то ваш браузер НЕ поддерживает `run-in`.

Вот, для примера, правильный вариант отображения `run-in`, оформленный другим кодом:

```
<div style="border:2px solid black">
  <h3 style="display: inline; border:2px solid red">Про пчёл.</h3>Пчёлы - отличные
</div>
```

Про пчёл. Пчёлы - отличные создания, они делают мёд.

Если этот вариант отличается от того, что вы видите выше – ваш браузер не поддерживает `run-in`. На момент написания этой статьи только IE поддерживал `display:run-in`.

`flex-box`

Flexbox позволяет удобно управлять дочерними и родительскими элементами на странице, располагая их в необходимом порядке. Официальная

спецификация находится здесь: CSS Flexible Box Layout Module

Свойство float

Свойство `float` в CSS занимает особенное место. До его появления расположить два блока один слева от другого можно было лишь при помощи таблиц. Но в его работе есть ряд особенностей. Поэтому его иногда не любят, но при их понимании `float` станет вашим верным другом и помощником.

Далее мы рассмотрим, как работает `float`, разберём решения сопутствующих проблем, а также ряд полезных рецептов.

Как работает float

Синтаксис:

```
float: left | right | none | inherit;
```

При применении этого свойства происходит следующее:

1. Элемент позиционируется как обычно, а затем *вынимается из ~~документа~~ потока* и сдвигается влево (для `left`) или вправо (для `right`) до того как коснётся либо границы родителя, либо другого элемента с `float`.
2. Если пространства по горизонтали не хватает для того, чтобы вместить элемент, то он сдвигается вниз до тех пор, пока не начнёт помещаться.
3. Другие непозиционированные блочные элементы без `float` ведут себя так, как будто элемента с `float` нет, так как он убран из потока.
4. Строки (iniine-элементы), напротив, «знают» о `float` и обтекают элемент по сторонам.

Ещё детали:

1. Элемент при наличии `float` получает `display:block`.

   То есть, указав элементу, у которого `display:inline` свойство `float: left/right`, мы автоматически сделаем его блочным. В частности, для него будут работать `width/height`.

   Исключением являются некоторые редкие `display` наподобие `inline-table` и `run-in` (см. Relationships between „display“, „position“, and „float“)

2. Ширина `float`-блока определяется по содержимому. («CSS 2.1, 10.3.5»).

3. Вертикальные отступы `margin` элементов с `float` не сливаются с отступами соседей, в отличие от обычных блочных элементов.

Это пока только теория. Далее мы рассмотрим происходящее на примере.

Текст с картинками

Одно из первых применений `float`, для которого это свойство когда-то было придумано – это вёрстка текста с картинками, отжатыми влево или вправо.

Например, вот страница о Винни-Пухе с картинками, которым поставлено свойство `float`:

Её HTML-код выглядит примерно так:

```
<img src="1.jpg" style="float:right">
<p>Текст...</p>
<p>Текст...</p>

<img src="2.jpg" style="float:left">
<p>Текст...</p>
```

```
<img src="3.jpg" style="float:right">
<p>Текст...</p>
```

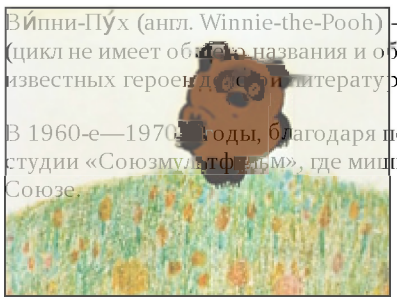
Каждая картинка, у которой есть `float`, обрабатывается в точности по алгоритму, указанному выше.

Посмотрим, например, как выглядело бы начало текста без float:

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге). Один из самых известных героев детской литературы XX века.

В 1960-е—1970-е годы, благодаря пересказу Бориса Заходера «Винни-Пух и все-все-все», а затем и фильмам студии «Союзмультфильм», где мишку озвучивал Евгений Леонов, Винни-Пух стал очень популярен и в Советском Союзе.

1. Элемент `IMG` вынимается из ~~документа~~ потока. Иначе говоря, последующие блоки начинают вести себя так, как будто его нет, и заполняют освободившееся место (изображение для наглядности полупрозрачно):



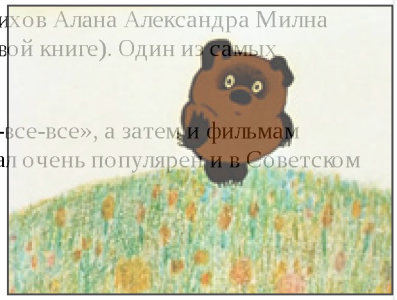
Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге). Один из самых известных героев детской литературы XX века.

В 1960-е—1970-е годы, благодаря пересказу Бориса Заходера «Винни-Пух и все-все-все», а затем и фильмам студии «Союзмультфильм», где мишку озвучивал Евгений Леонов, Винни-Пух стал очень популярен и в Советском Союзе.

2. Элемент `IMG` сдвигается максимально вправо(при `float:right`):

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге). Один из самых известных героев детской литературы XX века.

В 1960-е—1970-е годы, благодаря пересказу Бориса Заходера «Винни-Пух и все-все-все», а затем и фильмам студии «Союзмультфильм», где мишку озвучивал Евгений Леонов, Винни-Пух стал очень популярен и в Советском Союзе.



3. Строки, в отличие от блочных элементов, «чувствуют» `float` и уступают ему место, обтекая картинку слева:

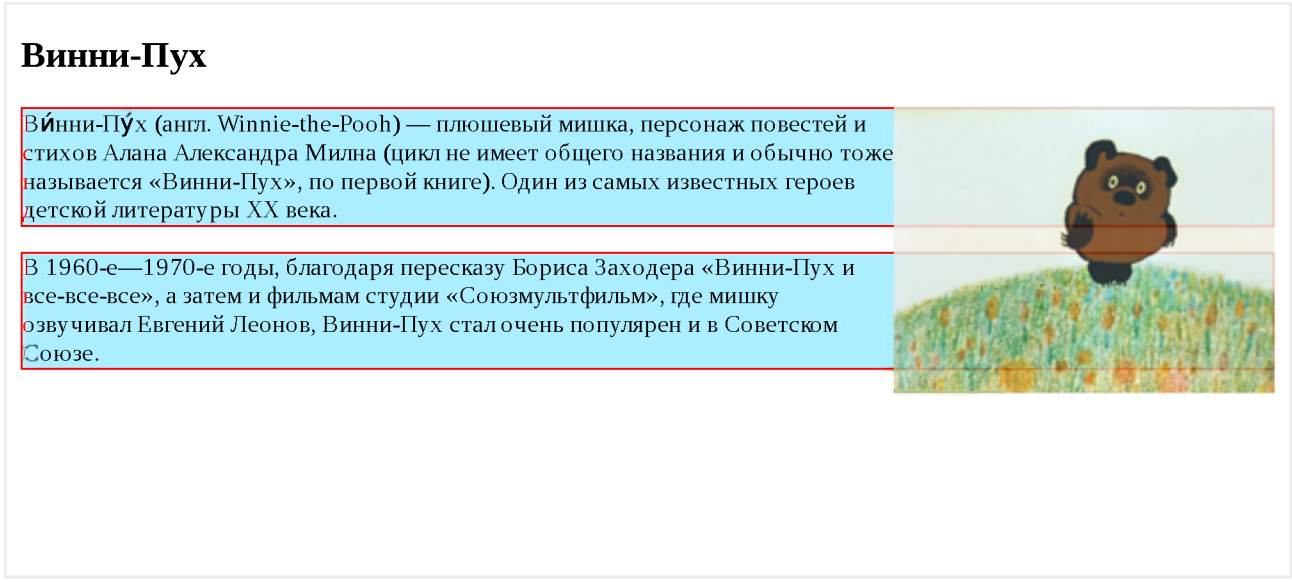
Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге). Один из самых известных героев детской литературы XX века.

В 1960-е—1970-е годы, благодаря пересказу Бориса Заходера «Винни-Пух и все-все-все», а затем и фильмам студии «Союзмультфильм», где мишку озвучивал Евгений Леонов, Винни-Пух стал очень популярен и в Советском Союзе.

При `float:left` – всё то же самое, только `IMG` смещается влево (или не смещается, если он и так у левого края), а строки – обтекают справа

Строки и инлайн-элементы смещаются, чтобы уступить место `float`. Обычные блоки – ведут себя так, как будто элемента нет.

Чтобы это увидеть, добавим параграфам фон и рамку, а также сделаем изображение немного прозрачным:



Как видно из рисунка, параграфы проходят «за» `float`. При этом строки в них о `float'ax` знают и обтекают их, поэтому соответствующая часть параграфа пуста.

Блок с float

Свойство `float` можно поставить любому элементу, не обязательно картинке. При этом элемент станет блочным.

Посмотрим, как это работает, на конкретной задаче – сделать рамку с названием вокруг картинки с Винни.

HTML будет такой:

```
<h2>Винни-Пух</h2>

<div class="left-picture">
  <img src="winnie-mult.jpg" width="200" height="150">
  <div>Кадр из советского мультфильма</div>
</div>

<p>Текст...</p>
```

...То есть, `div.left-picture` включает в себя картинку и заголовок к ней. Добавим стиль с `float`:

```
.left-picture {
  float: left;

  /* рамочка и отступ для красоты (не обязательно) */
  margin: 0 10px 5px 0;
  text-align: center;
  border: 1px solid black;
}
```

Результат:

Винни-Пух

Кадр из советского мультфильма

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге).

Один из самых известных героев детской литературы XX века.

Заметим, что блок `div.left-picture` «обернул» картинку и текст под ней, а не растянулся на всю ширину. Это следствие того, что ширина блока с `float` определяется по содержимому.

Очистка под float

Разберём ещё одну особенность использования свойства `float`.

Для этого выведем персонажей из мультфильма «Винни-Пух». Цель:

Винни-Пух

Кадр из советского мультфильма

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге).

Один из самых известных героев детской литературы XX века.

Сова

Кадр из советского мультфильма

Персонаж мультфильма про Винни-Пуха. Очень умная.

Говорит редко, но чрезвычайно мудро.

Реализуем её, шаг за шагом.

Шаг 1. Добавляем информацию

Попробуем просто добавить Сову после Винни-Пуха:

```
<h2>Винни-Пух</h2>
<div class="left">Картинка</div>
<p>..Текст о Винни..</p>

<h2>Сова</h2>
<div class="left">Картинка</div>
<p>..Текст о Сове..</p>
```

Результат такого кода будет странным, но предсказуемым:

Винни-Пух

Кадр из советского мультфильма

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге).

Один из самых известных героев детской литературы XX века.

Сова

Кадр из советского мультфильма

Персонаж мультфильма про Винни-Пуха. Очень умная.

Говорит редко, но чрезвычайно мудро.

Произошло следующее:

- **Заголовок `<h2>Сова</h2>` не заметил `float`** (он же блочный элемент) и расположился сразу после предыдущего параграфа `<p>..Текст о Винни..</p>`.
- После него идёт `float`-элемент – картинка «Сова». Он был сдвинут влево. Согласно алгоритму, он двигается до левой границы или до касания с другим `float`-элементом, что и произошло (картинка «Винни-Пух»).
- Так как у совы `float:left`, то **последующий текст обтекает её справа**.

Шаг 2. Свойство clear

Мы, конечно же, хотели бы расположить заголовок «Сова» и остальную информацию ниже Винни-Пуха.

Для решения возникшей проблемы придумано свойство `clear`.

Синтаксис:

```
clear: left | right | both;
```

Применение этого свойства сдвигает элемент вниз до тех пор, пока не закончатся `float`'ы слева/справа/с обеих сторон.

Применим его к заголовку `H2`:

```
h2 {
  clear: left;
}
```

Результат получившегося кода будет ближе к цели, но всё ещё не идеален:

Элементы теперь в нужном порядке. Но куда пропал отступ `margin-top` у заголовка «Сова»?

Теперь заголовок «Сова» прилегает снизу почти вплотную к картинке, с учётом её `margin-bottom`, но без своего большого отступа `margin-top`.

Таково поведение свойства `clear`. Оно сдвинуло элемент `h2` вниз ровно настолько, чтобы элементов `float` не было *сбоку от его верхней границы*.

Если посмотреть на элемент заголовка внимательно в инструментах разработчика, то можно заметить отступ `margin-top` у заголовка по-прежнему есть, но он располагается «за» элементом `float` и не учитывается при работе в `clear`.

Чтобы исправить ситуацию, можно добавить перед заголовком пустой промежуточный элемент без отступов, с единственным свойством `clear:both`. Тогда уже под ним отступ заголовка будет работать нормально:

```
<h2>Винни-Пух</h2>
<div class="left">Картинка</div>
<p>Текст</p>

<div style="clear:both"></div>

<h2>Сова</h2>
<div class="left">Картинка</div>
<p>Текст</p>
```

Результат получившегося кода :

Винни-Пух

Кадр из советского мультфильма

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге).

Один из самых известных героев детской литературы XX века.

Сова

Кадр из советского мультфильма

Персонаж мультфильма про Винни-Пуха. Очень умная.

Говорит редко, но чрезвычайно мудро.

- Свойство `clear` гарантировало, что `<div style="clear:both">` будет под картинкой с `float`.

- Заголовок `<h2>Сова</h2>` идёт после этого `<div>`. Так что его отступ учитывается.

Заполнение блока-родителя

Итак, мы научились располагать другие элементы *под* `float`. Теперь рассмотрим следующую особенность.

Из-за того, что блок с `float` удалён из потока, родитель не выделяет под него места.

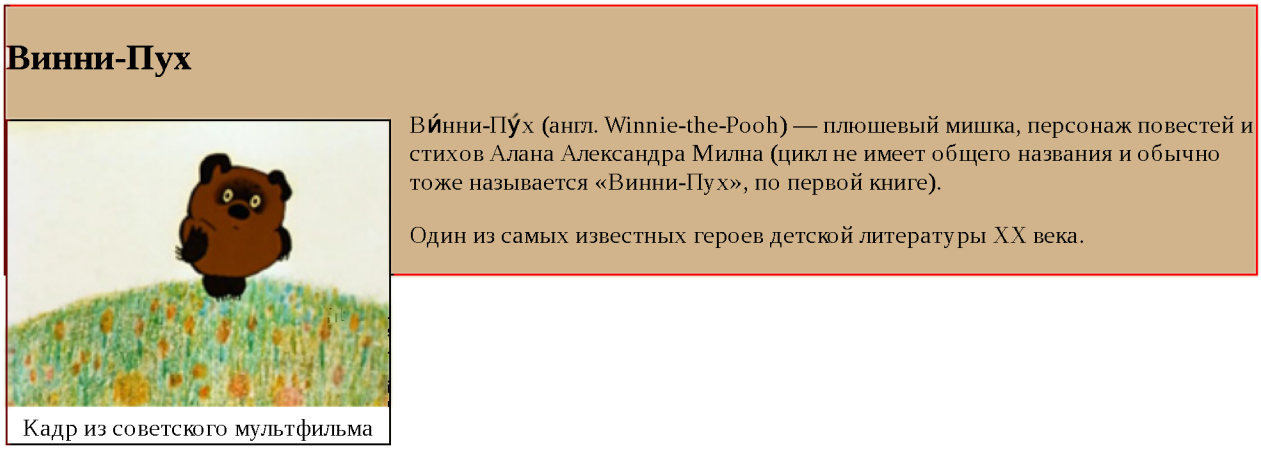
Например, выделим для информации о Винни-Пухе красивый элемент-контейнер `<div class="hero">`:

```
<div class="hero">

  <h2>Винни-Пух</h2>

  <div class="left">Картинка</div>

  <p>Текст.</p>
</div>
```

Стиль контейнера:

```
.hero {
  background: #D2B48C;
  border: 1px solid red;
}
```

Результат получившегося кода :



Элемент с `float` оказался выпавшим за границу родителя `.hero`.

Чтобы этого не происходило, используют одну из следующих техник.

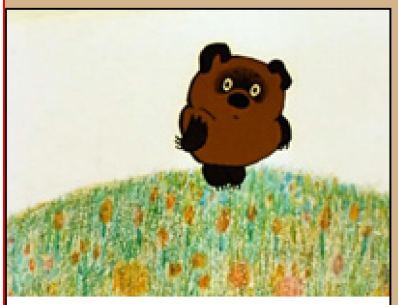
Поставить родителю float

Элемент с `float` обязан расшириться, чтобы вместить вложенные `float`.

Поэтому, если это допустимо, то установка `float` контейнеру всё исправит:

```
.hero {
  background: #D2B48C;
  border: 1px solid red;
  float: left;
}
```

Винни-Пух



Кадр из советского мультфильма

Ви́нни-Пу́х (англ. Winnie-the-Pooh) — плюшевый мишка, персонаж повестей и стихов Алана Александра Милна (цикл не имеет общего названия и обычно тоже называется «Винни-Пух», по первой книге).

Один из самых известных героев детской литературы XX века.

Разумеется, не всегда можно поставить родителю `float`, так что смотрим дальше.

Добавить в родителя элемент с clear

Добавим элемент `div style="clear:both"` в самый конец контейнера `.hero`.

Он с одной стороны будет «нормальным» элементом, в потоке, и контейнер будет обязан выделить под него пространство, с другой – он знает о `float` и сместится вниз.

Соответственно, и контейнер вырастет в размере:

```
<div class="hero">

  <h2>Винни-Пух</h2>

  <div class="left">Картинка</div>
```

```
  <p>Текст.</p>

  <div style="clear:both"></div>
</div>
```

Результат – правильное отображение, как и в примере выше. Открыть код .

Единственный недостаток этого метода – лишний HTML-элемент в разметке.

Универсальный класс clearfix

Чтобы не добавлять в HTML-код лишний элемент, можно задать его через `:after`.

```
.clearfix:after {
  content: "."; /* добавить содержимое: "." */
  display: block;  /* сделать блоком, т.к. inline не может иметь clear */
  clear: both;  /* с обеих сторон clear */
  visibility: hidden; /* сделать невидимым, зачем нам точка внизу? */
  height: 0;  /* сделать высоту 0, чтобы не занимал место */
}
```

Добавив этот класс к родителю, получим тот же результат, что и выше. Открыть код .

overflow:auto/hidden

Если добавить родителю `overflow: hidden` или `overflow: auto`, то всё станет хорошо.

```
.hero {
  overflow: auto;
}
```

Этот метод работает во всех браузерах, полный код в песочнице .

Несмотря на внешнюю странность, этот способ не является «хаком». Такое поведение прописано в спецификации CSS.

Однако, установка `overflow` может привести к появлению полосы прокрутки, способ с псевдоэлементом `:after` более безопасен.

float вместо display:inline-block

При помощи `float` можно размещать блочные элементы в строке, похоже на `display: inline-block`:

https://plnkr.co/edit/3bwnqJjFLMGnYjMA5LJC?p=preview

Стиль здесь:

```
.gallery li {
  float: left;
  width: 130px;
  list-style: none;
}
```

Элементы `float:left` двигаются влево, а если это невозможно, то вниз, автоматически адаптируясь под ширину контейнера, получается эффект, аналогичный `display: inline-block`, но с особенностями `float`.

Вёрстка в несколько колонок

Свойство `float` позволяет делать несколько вертикальных колонок.

float:left + float:right

Например, для вёрстки в две колонки можно сделать два `<div>`. Первому указать `float:left` (левая колонка), а второму – `float:right` (правая колонка).

Чтобы они не ссорились, каждой колонке нужно дополнительно указать ширину:

```
<div>Шапка</div>
<div class="column-left">Левая колонка</div>
<div class="column-right">Правая колонка</div>
<div class="footer">Низ</div>
```

Стили:

```
.column-left {
  float: left;
  width: 30%;
}

.column-right {
  float: left;
  width: 70%;
}

.footer {
  clear: both;
}
```

Результат (добавлены краски):

https://plnkr.co/edit/mRhQDYwCT9ArV0EGklWg?p=preview

В эту структуру легко добавить больше колонок с разной шириной. Правой колонке можно было бы указать и `float:right`.

float + margin

Ещё вариант – сделать `float` для левой колонки, а правую оставить в потоке, но с отбивкой через `margin`:

```
.column-left {
  float: left;
  width: 30%;
}

.column-right {
  margin-left: 30%;
}

.footer {
  clear: both;
}
```

Результат (добавлены краски):

https://plnkr.co/edit/bjkaZcfpp6O8NbmcKdLQ?p=preview

В примере выше – показана небольшая проблема. Колонки не растягиваются до одинаковой высоты. Конечно, это не имеет значения, если фон одинаковый, но что, если он разный?

В современных браузерах (кроме IE10-) эту же задачу лучше решает flexbox.

Для старых есть различные обходы и трюки, которые позволяют обойти проблему в ряде ситуаций, но они выходят за рамки нашего обсуждения. Если интересно – посмотрите, например, Faux Columns .

Задачи

Разница inline-block и float

важность: 5

Галерея изображений состоит из картинок в рамках с подписями (возможно, с другой дополнительной информацией).

Пример галереи:

Технически вывод такой галереи можно реализовать при помощи списка UL/LI, где:

1. каждый LI имеет `display:inline-block`
2. каждый LI имеет `float:left`

Какие различия между этими подходами? Какой вариант выбрали бы вы?

К решению

Дерево с многострочными узлами

важность: 3

Сделайте дерево при помощи семантической вёрстки и CSS-спрайта с иконками (есть готовый).

Выглядеть должно так (не кликабельно):

- Поддержка многострочных названий узлов
- Над иконкой курсор становится указателем.

Исходный документ содержит список UL/LI и ссылку на картинку.

P.S. Достаточно сделать HTML/CSS-структуру, действия добавим позже.

Открыть песочницу для задачи.

К решению

Постраничная навигация (CSS)

важность: 5

Оформите навигацию, центрированную внутри `DIV'a`:

Требования:

- Левая стрелка – слева, правая – справа, список страниц – по центру.
- Список страниц центрирован вертикально.
- Текст сверху и снизу ни на что не наползает.
- Курсор при наведении на стрелку или элемент списка становится стрелкой `pointer`.

P.S. Без использования таблиц.

Открыть песочницу для задачи.

К решению

Добавить рамку, сохранив ширину

важность: 4

Есть две колонки `30%/70%`:

```
<style>
  .left {
    float:left;
    width:30%;
    background: #aef;
  }

  .right {
    float:right;
    width:70%;
    background: tan;
  }
</style>

<div class="left">
 Левая<br>Колонка
</div>
<div class="right">
 Правая<br>Колонка<br>...
</div>
```

Добавьте к правой колонке рамку `border-left` и отступ `padding-left`.

Двухколоночная вёрстка при этом не должна сломаться!

Желательно не трогать свойство `width` ни слева ни справа и не создавать дополнительных элементов.

К решению

Свойство position

Свойство `position` позволяет сдвигать элемент со своего обычного места. Цель этой главы – не только напомнить, как оно работает, но и разобрать ряд частых заблуждений и граблей.

position: static

Статическое позиционирование производится по умолчанию, в том случае, если свойство `position` не указано.

Его можно также явно указать через CSS-свойство:

```
position: static;
```

Такая запись встречается редко и используется для переопределения других значений `position`.

Здесь и далее, для примеров мы будем использовать следующий документ:

```
<div style="background: #fee; width: 500px">
    Без позиционирования ("position: static").

    <h2 style="background: #aef; margin: 0">Заголовок</h2>

    <div>А тут - всякий разный текст... <br/>
        ... В две строки!</div>
</div>
```

Без позиционирования ("position: static").

Заголовок

А тут - всякий разный текст...
... В две строки!

В этом документе сейчас все элементы отпозиционированы статически, то есть никак.

> Элемент с `position: static` ещё называют *не позиционированым*.

position: relative

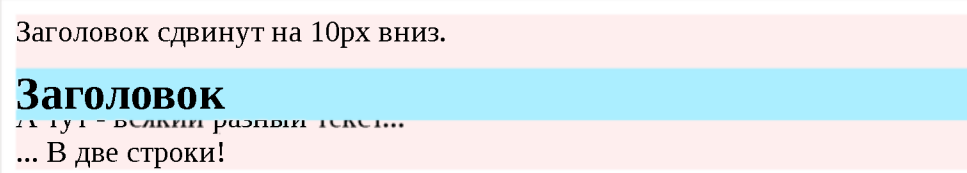
Относительное позиционирование сдвигает элемент относительно его обычного положения.

Для того, чтобы применить относительное позиционирование, необходимо указать элементу CSS-свойство `position: relative` и координаты `left/right/top/bottom`.

Этот стиль сдвинет элемент на 10 пикселей относительно обычной позиции по вертикали:

```
position: relative;
top: 10px;
```

```
<style>
  h2 {
    position: relative;
    top: 10px;
  }
</style>

<div style="background: #fee; width: 500px">
    Заголовок сдвинут на 10px вниз.

    <h2 style="background: #aef; margin: 0;">Заголовок</h2>

    <div>А тут - всякий разный текст... <br/>
        ... В две строки!</div>
</div>
```



Координаты

Для сдвига можно использовать координаты:

- `top` – сдвиг от «обычной» верхней границы
- `bottom` – сдвиг от нижней границы
- `left` – сдвиг слева
- `right` – сдвиг справа

Не будут работать одновременно указанные `top` и `bottom`, `left` и `right`. Нужно использовать только одну границу из каждой пары.

Возможны отрицательные координаты и координаты, использующие другие единицы измерения. Например, `left: 10%` сдвинет элемент на 10% его

ширины вправо, а `left: -10%` – влево. При этом часть элемента может оказаться за границей окна:

```
<style>
  h2 {
    position: relative;
    left: -10%;
  }
</style>
```

```
<div style="background: #fee; width: 500px">
    Заголовок сдвинут на 10% влево.

    <h2 style="background: #aef; margin: 0;">Заголовок</h2>

    <div>А тут - всякий разный текст... <br/>
         ... В две строки!</div>
</div>
```

Заголовок сдвинут на 10% влево.

оловок

А тут - всякий разный текст...
... В две строки!

Свойства `left/top` не будут работать для `position:static`. Если их все же поставить, браузер их проигнорирует. Эти свойства предназначены для работы только с позиционированными элементами.

position: absolute

Синтаксис:

```
position: absolute;
```

Абсолютное позиционирование делает две вещи:

1. **Элемент исчезает с того места, где он должен быть и позиционируется заново.** Остальные элементы, располагаются так, как будто этого элемента никогда не было.
2. **Координаты `top/bottom/left/right` для нового местоположения отсчитываются от ближайшего позиционированного родителя**, т.е.

родителя с позиционированием, отличным от `static`. Если такого родителя нет – то относительно документа.

Кроме того:

- **Ширина элемента с `position: absolute` устанавливается по содержимому.** Детали алгоритма вычисления ширины описаны в стандарте .
- **Элемент получает `display:block`**, который перекрывает почти все возможные `display` (см. Relationships between „display“, „position“, and „float“).

Например, отпозиционируем заголовок в правом-верхнем углу документа:

```
<style>
  h2 {
    position: absolute;
    right: 0;
    top: 0;
  }
</style>
```

```
<div style="background: #fee; width: 500px">
    Заголовок в правом-верхнем углу документа.

    <h2 style="background: #aef; margin: 0;">Заголовок</h2>

    <div>А тут - всякий разный текст... <br/>
         ... В две строки!</div>
</div>
```

Заголовок в правом-верхнем углу документа.
А тут - всякий разный текст...
... В две строки!

Заголовок

Важное отличие от `relative`: **так как элемент удаляется со своего обычного места, то элементы под ним сдвигаются, занимая освободившееся пространство**. Это видно в примере выше: строки идут одна за другой.

Так как при `position:absolute` размер блока устанавливается по содержимому, то широкий `Заголовок` «съёжился» до прямоугольника в углу.

Иногда бывает нужно поменять элементу `position` на `absolute`, но так, чтобы элементы вокруг не сдвигались. Как правило это делают, меняя соседей – добавляют `margin/padding` или вставляют в документ пустой элемент с такими же размерами.

Одновременное указание `left/right`, `top/bottom`

В абсолютно позиционированном элементе можно одновременно задавать противоположные границы.

Браузер растянет такой элемент до границ.

```
<style>
div {
  position: absolute;
  left: 10px; right: 10px; top: 10px; bottom: 10px;
}
</style>
<div style="background:#aef;text-align:center">10px от границ</div>
```

10px от границ

Внешним блоком является окно

Как растянуть абсолютно позиционированный блок на всю ширину документа?

Первое, что может прийти в голову:

```
/*+ no-beautify */
div {
  position: absolute;
  left: 0; top: 0; /* в левый-верхний угол */
  width: 100%; height: 100%; /* .. и растянуть */
}
```

Но это будет работать лишь до тех пор, пока у страницы не появится скроллинг!

Прокрутите вниз ифрейм:

Вы увидите, что голубой фон оканчивается задолго до конца документа.

Дело в том, что в CSS `100%` относится к ширине внешнего блока («containing block»). А какой внешний блок имеется в виду здесь, ведь элемент изъят со своего обычного места?

В данном случае им является так называемый ("«initial containing block»"), которым является окно, *а не документ*.

То есть, координаты и ширины вычисляются относительно окна, а не документа.

Может быть, получится так?

```
/*+ no-beautify */
div {
  position: absolute;
  left: 0; top: 0; /* в левый-верхний угол, и растянуть..  */
```

```
  right: 0; bottom: 0; /* ..указанием противоположных границ */
}
```

С виду логично, но нет, не получится!

Координаты `top/right/left/bottom` вычисляются относительно *окна*. Значение `bottom: 0` – нижняя граница окна, а не документа, блок растянется до неё. То есть, будет то же самое, что и в предыдущем примере.

position: absolute в позиционированном родителе

Если у элемента есть позиционированный предок, то `position: absolute` работает относительно него, а не относительно документа.

То есть, достаточно поставить родительскому `div` позицию `relative`, даже без координат – и заголовок будет в его правом-верхнем углу, вот так:

```
<style>
  h2 {
    position: absolute;
    right: 0;
    top: 0;
  }
</style>

<div style="background: #fee; width: 500px; position: relative">
    Заголовок в правом-верхнем углу DIV'a.

    <h2 style="background: #aef; margin: 0;">Заголовок</h2>

    <div>А тут - всякий разный текст... <br/>
        ... В две строки!</div>
</div>
```

Заголовок в правом-верхнем углу DIV'a.
А тут - всякий разный текст...
... В две строки!

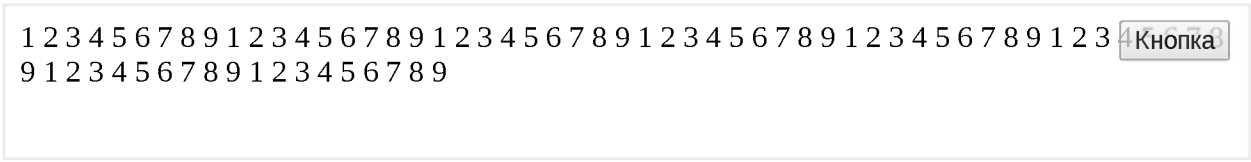
Заголовок

Нужно пользоваться таким позиционированием с осторожностью, т.к оно может перекрыть текст. Этим оно отличается от `float`.

Сравните:

- Используем `position` для размещения элемента управления:

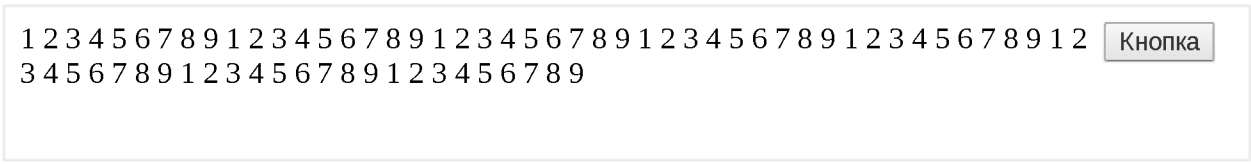
```
<button style="position: absolute; right: 10px; opacity: 0.8">
  Кнопка
</button>
1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
```



Часть текста перекрывается. Кнопка более не участвует в потоке.

- Используем `float` для размещения элемента управления:

```
<button style="float: right; margin-right: 10px; opacity: 0.8;">
  Кнопка
</button>
1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
```



Браузер освобождает место справа, текст перенесён. Кнопка продолжает находиться в потоке, просто сдвинута.

position: fixed

Это подвид абсолютного позиционирования.

Синтаксис:

```
position: fixed;
```

Позиционирует объект точно так же, как `absolute`, но относительно `window`.

Разница в нескольких словах:

Когда страницу прокручивают, фиксированный элемент остаётся на своём месте и не прокручивается вместе со страницей.

В следующем примере, при прокрутке документа, ссылка `#top` всегда остаётся на своём месте.

```
<style>
  #top {
    position: fixed;
    right: 10px;
    top: 10px;
    background: #fee;
  }
</style>

<a href="#" id="top">Наверх (остаётся при прокрутке)</a>

Фиксированное позиционирование.

<p>Текст страницы.. Прокрути меня...</p>
<p>Много строк..</p><p>Много строк..</p>
<p>Много строк..</p><p>Много строк..</p>
<p>Много строк..</p><p>Много строк..</p>
<p>Много строк..</p><p>Много строк..</p>
```

Фиксированное позиционирование. Наверх (остаётся при прокрутке)

Текст страницы.. Прокрути меня...

Много строк..

Много строк..

Много строк..

Много строк..

Итого

Виды позиционирования и их особенности.

`static`

Иначе называется «без позиционирования». В явном виде задаётся только если надо переопределить другое правило CSS.

`relative`

Сдвигает элемент относительно текущего места.

- Противоположные границы `left/right` (`top/bottom`) одновременно указать нельзя.
- Окружающие элементы ведут себя так, как будто элемент не сдвигался.

`absolute`

Визуально переносит элемент на новое место.

Новое место вычисляется по координатам `left/top/right/bottom` относительно ближайшего позиционированного родителя. Если такого родителя нет, то им считается окно.

- Ширина элемента по умолчанию устанавливается по содержимому.
- Можно указать противоположные границы `left/right` (`top/bottom`). Элемент растянется.
- Окружающие элементы заполняют освободившееся место.

`fixed`

Подвид абсолютного позиционирования, при котором элемент привязывается к координатам окна, а не документа.

При прокрутке он остаётся на том же месте.

Почитать

CSS-позиционирование по-настоящему глубоко в спецификации Visual Formatting Model, 9.3 и ниже .

Ещё есть хорошее руководство CSS Positioning in 10 steps , которое охватывает основные типы позиционирования.

Задачи

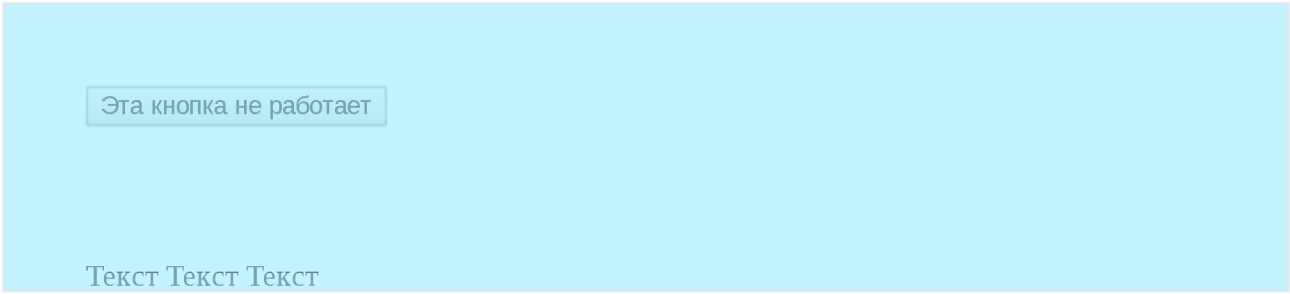
Модальное окно

важность: 5

Создайте при помощи HTML/CSS «модальное окно», то есть `DIV`, который полностью перекрывает документ и находится над ним.

При этом все элементы управления на документе перестают работать, т.к. клики попадают в `DIV`.

В примере ниже `DIV'у` дополнительно поставлен цвет фона и прозрачность, чтобы было видно перекрытие:



Браузеры: все основные, IE8+. Должно работать при прокрутке окна (проверьте).

Открыть песочницу для задачи.

К решению

Центрирование горизонтальное и вертикальное

В CSS есть всего несколько техник центрирования элементов. Если их знать, то большинство задач решаются просто.

Горизонтальное

text-align

Для центрирования инлайновых элементов – достаточно поставить родителю `text-align: center`.:

```
<style>
  .outer {
    text-align: center;
    border: 1px solid blue;
  }
</style>

<div class="outer">Текст</div>
```

Для центрирования блока это уже не подойдёт, свойство просто не подействует. Например:

```
<style>
  .outer {
    text-align: center;
```

```
    border: 1px solid blue;
  }
  .inner {
    width: 100px;
    border: 1px solid red;
  }
</style>

<div class="outer">
  <div class="inner">Текст</div>
</div>
```

margin: auto

Блок по горизонтали центрируется `margin: auto`:

```
<style>
  .outer {
    border: 1px solid blue;
  }
  .inner {
    width: 100px;
    border: 1px solid red;
    margin: auto;
  }
</style>

<div class="outer">
  <div class="inner">Текст</div>
</div>
```

В отличие от `width/height`, значение `auto` для `margin` само не появляется. Обычно `margin` равно конкретной величине для элемента, например `0` для `DIV`. Нужно поставить его явно.

Значение `margin-left:auto/margin-right:auto` заставляет браузер выделять под `margin` всё доступное сбоку пространство. А если и то и другое `auto`, то слева и справа будет одинаковый отступ, таким образом элемент окажется в середине. Детали вычислений описаны в разделе спецификации Calculating widths and margins .

Вертикальное

Для горизонтального центрирования всё просто. Вертикальное же изначальное не было предусмотрено в спецификации CSS и по сей день вызывает ряд проблем.

Есть три основных решения.

position:absolute + margin

Центрируемый элемент позиционируем абсолютно и опускаем до середины по вертикали при помощи `top:50%`:

```
<style>
  .outer {
    position: relative;
    height: 5em;
    border: 1px solid blue;
  }

  .inner {
    position: absolute;
    top: 50%;
    border: 1px solid red;
  }
</style>

<div class="outer">
  <div class="inner">Текст</div>
</div>
```

Это, конечно, не совсем центр. По центру находится верхняя граница. Нужно ещё приподнять элемент на половину своей высоты.

Высота центрируемого элемента должна быть известна. Родитель может иметь любую высоту.

Если мы знаем, что это ровно одна строка, то её высота равна `line-height`.

Приподнимем элемент на пол-высоты при помощи `margin-top`:

```
<style>
  .outer {
    position: relative;
    height: 5em;
    border: 1px solid blue;
  }
```

```
  .inner {
    position: absolute;
    top: 50%;
    margin-top: -0.625em;
    border: 1px solid red;
  }
</style>

<div class="outer">
  <div class="inner">Текст</div>
</div>
```

Почему‑0.625em?

При стандартных настройках браузера высота строки `line-height: 1.25`, если поделить на два `1.25em / 2 = 0.625em`.

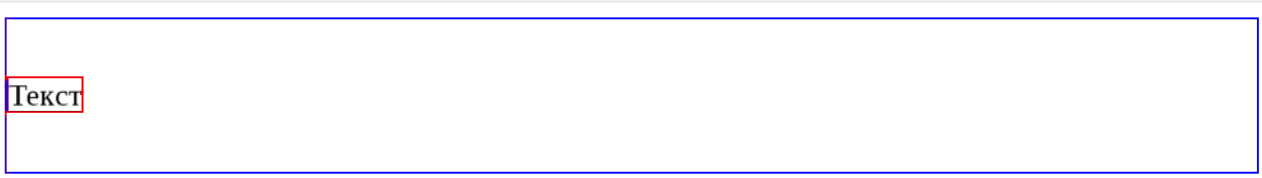
Конечно, высота может быть и другой, главное чтобы мы её знали заранее.

Можно аналогично центрировать и по горизонтали, если известен горизонтальный размер, при помощи `left:50%` и отрицательного `margin-left`.

Одна строка: line-height

Вертикально отцентрировать одну строку в элементе с известной высотой `height` можно, указав эту высоту в свойстве `line-height`:

```
<style>
  .outer {
    height: 5em;
    line-height: 5em;
    border: 1px solid blue;
  }
</style>

<div class="outer">
  <span style="border:1px solid red">Текст</span>
</div>
```



Это работает, но лишь до тех пор, пока строка одна, а если содержимое вдруг переносится на другую строку, то начинает выглядеть довольно уродливо.

Таблица с vertical-align

У свойства vertical-align , которое управляет вертикальным расположением элемента, есть два режима работы.

В таблицах свойство `vertical-align` указывает расположение *содержимого* ячейки.

Его возможные значения:

`baseline`

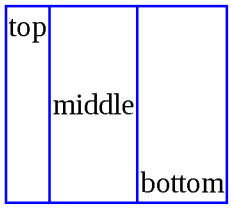
Значение по умолчанию.

`middle`, `top`, `bottom`

Располагать содержимое посередине, вверху, внизу ячейки.

Например, ниже есть таблица со всеми 3-мя значениями:

```
<style>
  table {  border-collapse: collapse; }
  td {
    border: 1px solid blue;
    height: 100px;
  }
</style>

<table>
<tr>
  <td style="vertical-align: top">top</td>
  <td style="vertical-align: middle">middle</td>
  <td style="vertical-align: bottom">bottom</td>
</tr>
</table>
```

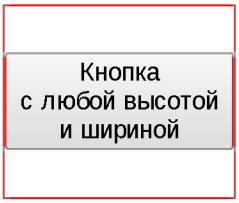


Обратим внимание, что в ячейке с `vertical-align: middle` содержимое находится по центру. Таким образом, можно обернуть нужный элемент в таблицу размера `width:100%;height:100%` с одной ячейкой, у которой указать `vertical-align:middle`, и он будет отцентрирован.

Но мы рассмотрим более красивый способ, который поддерживается во всех современных браузерах, и в IE8+. В них не обязательно делать таблицу, так как доступно значение `display:table-cell`. Для элемента с таким `display` используются те же алгоритмы вычисления ширины и центрирования, что и в `TD`. И, в том числе, работает `vertical-align`:

Пример центрирования:

```
<div style="display: table-cell; vertical-align: middle;  height: 100px; border: 1p
  <button>Кнопка<br>с любой высотой<br>и шириной</button>
</div>
```



Этот способ замечателен тем, что он не требует знания высоты элементов.

Однако у него есть особенность. Вместе с `vertical-align` родительский блок получает табличный алгоритм вычисления ширины и начинает подстраиваться под содержимое. Это не всегда желательно.

Чтобы его растянуть, нужно указать `width` явно, например: `300px`:

```
<div style="display: table-cell; vertical-align: middle; height: 100px; width: 300p
  <button>Кнопка<br>с любой высотой<br>и шириной</button>
</div>
```

Можно и в процентах, но в примере выше они не сработают, потому что структура таблицы «сломана» – ячейка есть, а собственно таблицы-то нет.

Это можно починить, завернув «псевдоячейку» в элемент с `display:table`, которому и поставим ширину:

```
<div style="display: table; width: 100%">
<div style="display: table-cell; vertical-align: middle; height: 100px; border: 1px
  <button>Кнопка<br>с любой высотой<br>и шириной</button>
</div>
</div>
```

Если дополнительно нужно горизонтальное центрирование – оно обеспечивается другими средствами, например `margin: 0 auto` для блочных элементов или `text-align:center` на родителе – для других.

Центрирование в строке с vertical-align

Для инлайновых элементов (`display:inline/inline-block`), включая картинки, свойство `vertical-align` центрирует *сам инлайн-элемент в окружающем его тексте*.

В этом случае набор значений несколько другой:

Картинка размером в `30px`, значения `vertical-align`:

Это можно использовать и для центрирования, если высота родителя известна, а центрируемого элемента – нет.

Допустим, высота внешнего элемента `120px`. Укажем её в свойстве `line-height`:

```
<style>
  .outer {
    line-height: 120px;
  }
  .inner {
    display: inline-block; /* центрировать..*/
    vertical-align: middle;  /* ..по вертикали */
    line-height: 1.25; /* переопределить высоту строки на обычную */
    border: 1px solid red;
  }
</style>
<div class="outer" style="height: 120px;border: 1px solid blue">
  <span class="inner">Центрирован<br>вертикально</span>
</div>
```

Работает во всех браузерах и IE8+.

Свойство `line-height` наследуется, поэтому надо знать «правильную» высоту строки и переопределять её для `inner`.

Центрирование с vertical-align без таблиц

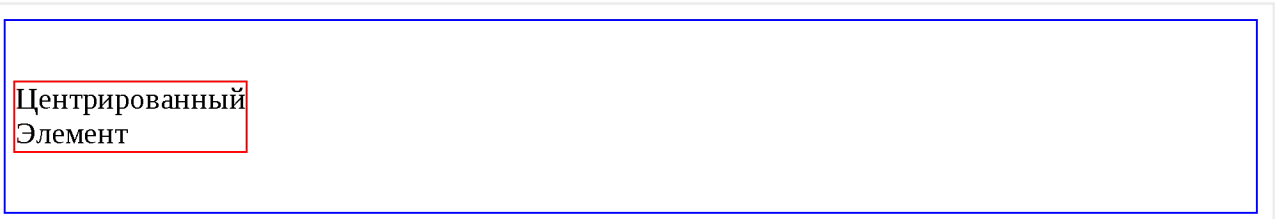
Если центрирование должно работать для любой высоты родителя и центрируемого элемента, то обычно используют таблицы или `display:table-cell` с `vertical-align`.

Если центрируются не-блочные элементы, например `inline` или `inline-block`, то `vertical-align` может решить задачу без всяких таблиц. Правда, понадобится вспомогательный элемент (можно через `:before`).

Пример:

```
<style>
.before {
  display: inline-block;
  height: 100%;
  vertical-align: middle;
}

.inner {
  display: inline-block;
  vertical-align: middle;
}
```

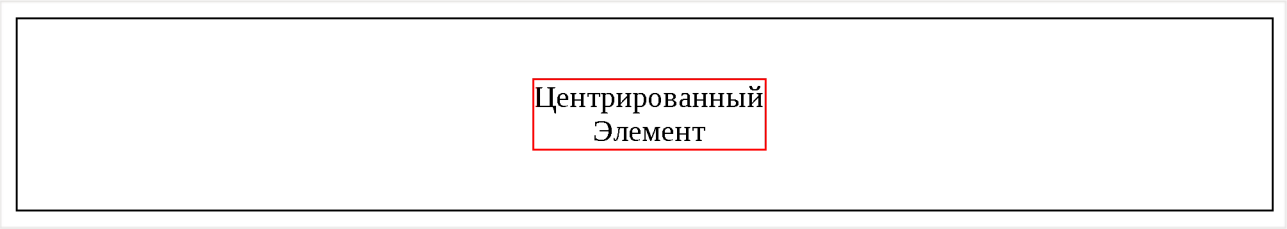
```
</style>

<div class="outer" style="height:100px;border:1px solid blue">
  <span class="before"></span>
  <span class="inner" style="border:1px solid red">
    Центрированный<br>Элемент
  </span>
</div>
```



- Перед центрируемым элементом помещается вспомогательный инлайн-блок `before`, занимающий всю возможную высоту.
- Центрируемый блок выровнен по его середине.

Для всех современных браузеров и IE8 можно добавить вспомогательный элемент через `:before`:

```
<style>
.outer:before {
  content: '';
  display: inline-block;
  height: 100%;
  vertical-align: middle;
}

.inner {
  display: inline-block;
  vertical-align: middle;
}

/* добавим горизонтальное центрирование */
.outer {
  text-align: center;
}
</style>

<div class="outer" style="height:100px; width: 100%; border:1px solid black">
  <span class="inner" style="border:1px solid red">
    Центрированный<br>Элемент
  </span>
</div>
```



В пример выше добавлено также горизонтальное центрирование `text-align: center`. Но вы можете видеть, что на самом деле внутренний элемент не центрирован горизонтально, он немного сдвинут вправо.

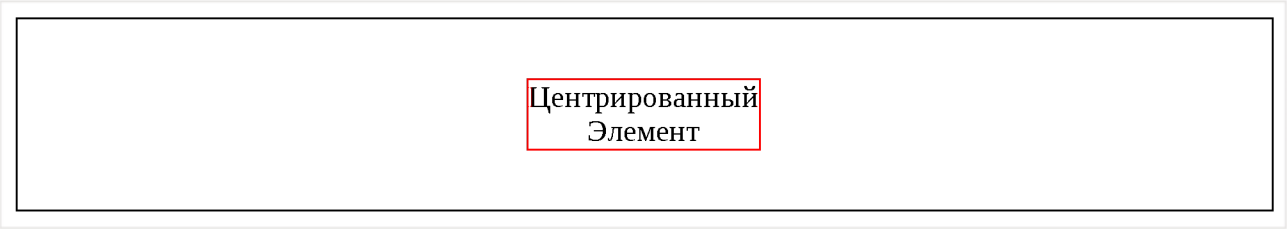
Это происходит потому, что центрируется *весь текст*, а перед `inner` находится пробел, который занимает место.

Варианта два:

1. Убрать лишний пробел между `div` и началом `inner`, будет `<div class="outer"><span class="inner">...`.
2. Оставить пробел, но сделать отрицательный `margin-left` у `inner`, равный размеру пробела, чтобы `inner` сместился левее.

Второе решение:

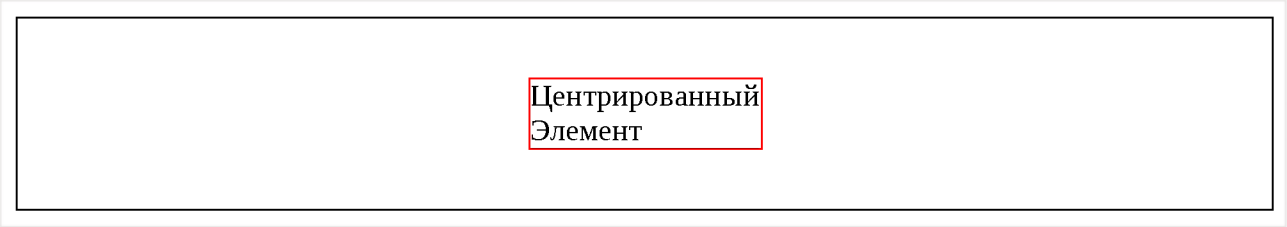
```
<style>
.outer:before {
  content: '';
  display: inline-block;
  height: 100%;
  vertical-align: middle;
}

.inner {
  display: inline-block;
  vertical-align: middle;
  margin-left: -0.35em;
}

.outer {
  text-align: center;
}
</style>

<div class="outer" style="height:100px; width: 100%; border:1px solid black">
  <span class="inner" style="border:1px solid red">
    Центрированный<br>Элемент
  </span>
</div>
```



Центрирование с использованием модели flexbox

Данный метод поддерживается всеми современными браузерами.

```
<style>
.outer {
    display: flex;
    justify-content: center; /*Центрирование по горизонтали*/
    align-items: center;     /*Центрирование по вертикали */
}
</style>

<div class="outer" style="height:100px; width: 100%; border:1px solid black">
    <span class="inner" style="border:1px solid red">
        Центрированный<br>Элемент
    </span>
</div>
```



Плюсы:

- Не требуется знания высоты центрирумого элемента.
- CSS чистый, короткий и не требует дополнительных элементов.

Минусы:

- Не поддерживается IE9-, IE10 поддерживает предыдущую версию flexbox.

Итого

Обобщим решения, которые обсуждались в этой статье.

Для горизонтального центрирования:

- `text-align: center` – центрирует инлайн-элементы в блоке.
- `margin: 0 auto` – центрирует блок внутри родителя. У блока должна быть указана ширина.

Для вертикального центрирования одного блока внутри другого:

Если размер центрируемого элемента известен, а родителя – нет

Родителю `position:relative`, потомку `position:absolute; top:50%` и `margin-top:-<половина-высоты-потомка>`. Аналогично можно отцентрировать и по горизонтали.

Если нужно отцентрировать одну строку в блоке, высота которого известна

Поставить блоку `line-height: <высота>`. Нужны конкретные единицы высоты (`px`, `em` ...). Значение `line-height:100%` не будет работать, т.к. проценты берутся не от высоты блока, а от текущей `line-height`.

Высота родителя известна, а центрируемого элемента – нет.

Поставить `line-height` родителю во всю его высоту, а потомку поставить `display:inline-block`.

Высота обоих элементов неизвестна.

Три варианта:

1. Сделать элемент-родитель ячейкой таблицы при помощи `display:table-cell` (IE8) или реальной таблицы, и поставить ему `vertical-align:middle`. Отлично работает, но мы имеем дело с таблицей вместо обычного блока.

2. Решение с вспомогательным элементом `outer:before` и инлайн-блоками. Вполне универсально и не создаёт таблицу.
3. Решение с использованием flexbox.

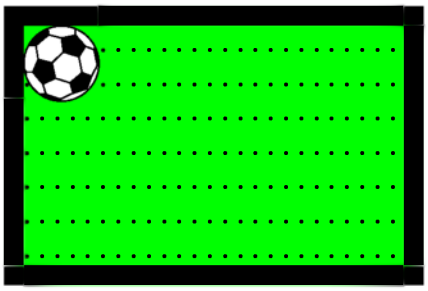
Задачи

Поместите мяч в центр поля (CSS)

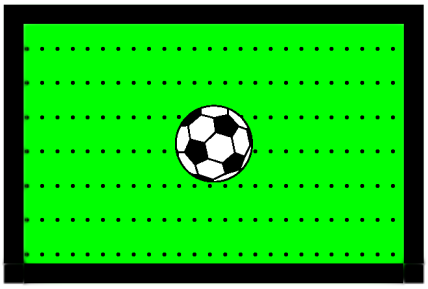
важность: 5

Поместите мяч в центр поля при помощи CSS.

Исходный код:



Используйте CSS, чтобы поместить мяч в центр:



- CSS для центрирования может использовать размеры мяча.
- CSS для центрирования не должен опираться на конкретный размер поля.

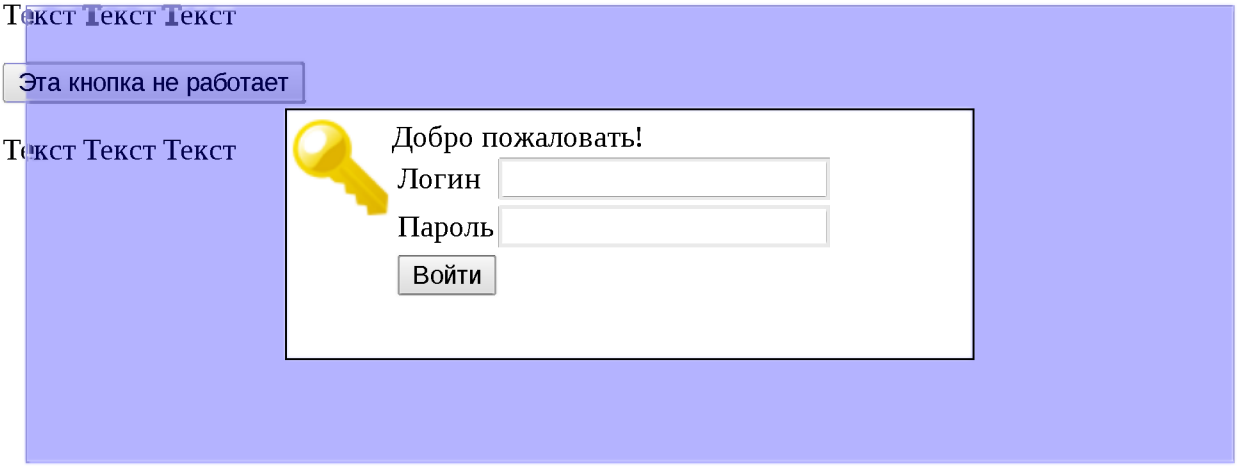
Открыть песочницу для задачи.

К решению

Форма + модальное окно

важность: 5

Создайте при помощи HTML/CSS форму для логина в модальном окне.



Требования:

- Кнопки окна вне формы не работают (даже на левый край нажать нельзя).
- Полупрозрачный голубой «экран» отстоит от границ на `20px`.
- Форма центрирована вертикально и горизонтально, её размеры фиксированы.
- Посетитель может менять размер окна браузера, геометрия должна сохраняться.
- Не ломается при прокрутке.

Браузеры: все основные, IE8+.

Открыть песочницу для задачи.

К решению

---

vertical-align + table-cell + position = ?

важность: 5

В коде ниже используется вертикальное центрирование при помощи `table-cell + vertical-align`.

Почему оно не работает? Нажмите на просмотр, чтобы увидеть (стрелка должна быть в центре по вертикали).

```
<style>
  .arrow {
    position: absolute;
    height: 60px;
    border: 1px solid black;
    font-size: 28px;

    display: table-cell;
    vertical-align: middle;
  }
</style>

<div class="arrow">«</div>
```

Как починить центрирование при помощи CSS? Свойства `position/height` менять нельзя.

К решению

Свойства font-size и line-height

Здесь мы рассмотрим, как соотносятся размеры шрифта и строки, и как их правильно задавать.

font-size и line-height

- `font-size` – *размер шрифта*, в частности, определяющий высоту букв.
- `line-height` – *высота строки*.

Для наглядности посмотрим пример HTML, в котором шрифт и размер строки одинаковы:

```
<style>
body {
    font-size: 30px;
    font-family: monospace;
    line-height: 30px;
  }
</style>

<div style="outline: 1px dotted red">Ёрш p</div>
<div style="outline: 1px dotted red">Ёрш Ё</div>
```

Ёрш p
Ёрш Ё

Размер шрифта `font-size` – это абстрактное значение, которое привязано к шрифту, и даётся в типографских целях.

Обычно оно равно расстоянию от самой верхней границы букв до самой нижней, исключая «нижние хвосты» букв, таких как `p`, `g`. Как видно из примера выше, при размере строки, равном `font-size`, строка не будет размером точно «под букву».

В зависимости от шрифта, «хвосты» букв при этом могут вылезать, правые буквы `Ё` и `p` в примере выше пересекаются как раз поэтому.

В некоторых особо размашистых шрифтах «хвосты букв» могут быть размером с саму букву, а то и больше. Но это, всё же исключение.

Обычно размер строки делают чуть больше, чем шрифт.

По умолчанию в браузерах используется специальное значение `line-height:normal`.

Оно означает, что браузер может принимать решение о размере строки самостоятельно. Как правило, оно будет в диапазоне `1.1 - 1.25`, но стандарт не гарантирует этого, он говорит лишь, что оно должно быть «разумным» (дословно – англ. reasonable).

Множитель для line-height

Значение `line-height` можно указать при помощи `px` или `em`, но гораздо лучше – задать его числом.

Значение-число интерпретируется как множитель относительно размера шрифта. Например, значение с множителем `line-height: 2` при `font-size: 16px` будет аналогично `line-height: 32px (=16px*2)`.

Однако, между множителем и точным значением есть одна существенная разница.

- **Значение, заданное множителем, наследуется и применяется в каждом элементе относительно его размера шрифта.**

  То есть, при `line-height: 2` означает, что высота строки будет равна удвоенному размеру шрифта, не важно какой шрифт.

- **Значение, заданное в единицах измерения, запоминается и наследуется «как есть».**

  Это означает, что `line-height: 32px` будет всегда жёстко задавать высоту строки, даже если шрифт во вложенных элементах станет больше или меньше текущего.

Давайте посмотрим, как это выглядит, на примерах:

Множитель, `line-height:1.25`

```
<div style="line-height: 1.25">
  стандартная строка
  <div style="font-size:2em">
    шрифт в 2 раза больше<br>
    шрифт в 2 раза больше
  </div>
</div>
```

стандартная строка

шрифт в 2 раза больше
шрифт в 2 раза больше

Конкретное значение, `line-height:1.25em`

```
<div style="line-height: 1.25em">
  стандартная строка
  <div style="font-size:2em">
    шрифт в 2 раза больше<br>
    шрифт в 2 раза больше
  </div>
</div>
```

стандартная строка
шрифт в 2 раза больше
шрифт в 2 раза больше

Какой вариант выглядит лучше? Наверно, первый. В нём размер строки более-менее соответствует шрифту, поскольку задан через множитель.

В обычных ситуациях рекомендуется использовать именно множитель, за исключением особых случаев, когда вы действительно знаете что делаете.

Синтаксис font: size/height family

Установить `font-size` и `line-height` можно одновременно.

Соответствующий синтаксис выглядит он так:

```
font: 20px/1.5 Arial,sans-serif;
```

При этом нужно обязательно указать сам шрифт, например `Arial,sans-serif`. Укороченный вариант `font: 20px/1.5` работать не будет.

Дополнительно можно задать и свойства `font-style`, `font-weight`:

```
font: italic bold 20px/1.5 Arial,sans-serif;
```

Итого

`line-height`

Размер строки, обычно он больше размера шрифта. При установке множителем рассчитывается каждый раз относительно текущего шрифта, при установке в единицах измерения – фиксируется.

`font-size`

Размер шрифта. Если сделать блок такой же высоты, как шрифт, то хвосты букв будут вылезать из-под него.

`font: 125%/1.5 FontFamily`

Даёт возможность одновременно задать размер, высоту строки и, собственно, сам шрифт.

Свойство white-space

Свойство `white-space` позволяет сохранять пробелы и переносы строк.

У него есть два известных значения:

- `white-space: normal` – обычное поведение
- `white-space: pre` – текст ведёт себя, как будто оформлен в тег `pre`.

Но браузеры поддерживают и другие, которые также бывают очень полезны.

pre / nowrap

Встречаем первую «сладкую парочку» – `pre` и `nowrap`.

Оба этих значения меняют стандартное поведение HTML при работе с текстом:

`pre`:

- Сохраняет пробелы.
- Переносит текст при явном разрыве строки.

`nowrap`

- Не сохраняет пробелы.
- Игнорирует явные разрывы строки (не переносит текст).

Оба этих значения поддерживаются кросс-браузерно.

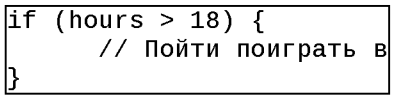
Их основной недостаток – текст может вылезти из контейнера.

Для примера, рассмотрим следующий фрагмент кода:

```
if (hours > 18) {
      // Пойти поиграть в теннис
}
```

`white-space: pre`:

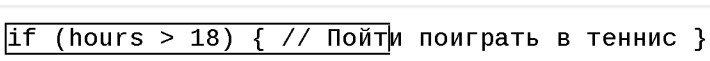
```
<style>
  div { font-family: monospace; width: 200px; border: 1px solid black; }
</style>

<div style="white-space:pre">if (hours > 18) {
      // Пойти поиграть в теннис
}
</div>
```



Здесь пробелы и переводы строк сохранены. В HTML этому значению `white-space` соответствует тег `PRE`.

`white-space: nowrap`:

```
<style>
  div { font-family: monospace; width: 200px; border: 1px solid black; }
</style>

<div style="white-space:nowrap">if (hours > 18) {
      // Пойти поиграть в теннис
}
</div>
```



Здесь переводы строки проигнорированы, а подряд идущие пробелы, если присмотреться – сжаты в один (например, перед комментарием `//`).

Допустим, мы хотим разрешить посетителям публиковать код на сайте, с сохранением разметки. Но тег `PRE` и описанные выше значения `white-space` для этого не подойдут!

Злой посетитель Василий Пупкин может написать такой текст, который вылезет из контейнера и поломает вёрстку страницы.

Можно скрыть вылезшее значение при помощи `overflow-x: hidden` или сделать так, чтобы была горизонтальная прокрутка, но, к счастью, есть другие значения `white-space`, специально для таких случаев.

pre-wrap/pre-line

`pre-wrap`

То же самое, что `pre`, но переводит строку, если текст вылезает из контейнера.

`pre-line`

То же самое, что `pre`, но переводит строку, если текст вылезает из контейнера и не сохраняет пробелы.

Оба поведения отлично прослеживаются на примерах:

`white-space: pre-wrap`:

```
<style>
  div { font-family: monospace; width: 200px; border: 1px solid black; }
</style>

<div style="white-space:pre-wrap">if (hours > 18) {
      // Пойти поиграть в теннис
}
</div>
```

```
if (hours > 18) {
      // Пойти поиграть в
теннис
}
```

Отличный выбор для безопасной вставки кода посетителями.

`white-space: pre-line`:

```
<style>
  div { font-family: monospace; width: 200px; border: 1px solid black; }
</style>

<div style="white-space:pre-line">if (hours > 18) {
      // Пойти поиграть в теннис
}
</div>
```

```
if (hours > 18) {
// Пойти поиграть в
теннис
}
```

Сохранены переводы строк, ничего не вылезает, но пробелы интерпретированы в режиме обычного HTML.

Свойство outline

Свойство `outline` задаёт дополнительную рамку вокруг элемента, *за пределами его CSS-блока*. Поддерживается во всех браузерах, IE8+.

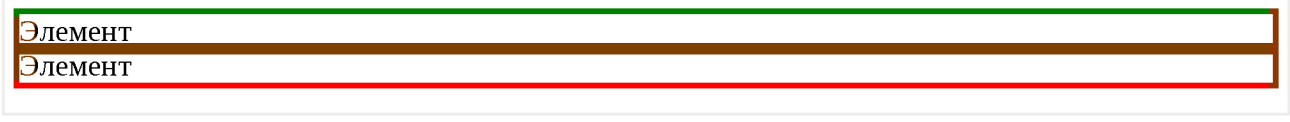
Для примера, рассмотрим его вместе с обычной рамкой `border`:

```
<div style="border:3px solid blue; outline: 3px solid red">
 Элемент
</div>
```

- **В отличие от `border`, рамка `outline` не участвует в блочной модели CSS.** Она не занимает места и не меняет размер элемента. Поэтому его используют, когда хотят добавить рамку без изменения других CSS-параметров.
- Также, в отличие от `border`, рамку `outline` можно задать только со всех сторон: свойств `outline-top`, `outline-left` не существует.

Так как `outline` находится за границами элемента – **`outline`-рамки соседей могут перекрывать друг друга**:

```
<div style="outline: 3px solid green">
 Элемент
</div>
<div style="outline: 3px solid red">
 Элемент
</div>
```



В примере выше верхняя рамка нижнего элемента находится на территории верхнего и наоборот.

Все браузеры, кроме IE9-, также поддерживают свойство `outline-offset`, задающее отступ `outline` от внешней границы элемента:

```
<div style="border:3px solid blue; outline: 3px solid red; outline-offset:5px">
 Везде, кроме IE9-, между рамками будет расстояние 5px
</div>
```

Ещё раз заметим, что основная особенность `outline` – в том, что при наличии `outline-offset` или без него – он не занимает места в блоке.

Поэтому его часто используют для стилей `:hover` и других аналогичных, когда нужно выделить элемент, но чтобы ничего при этом не прыгало.

Свойство box-sizing

Свойство `box-sizing` может принимать одно из двух значений – `border-box` или `content-box`. В зависимости от выбранного значения браузер по-разному трактует значение свойств `width/height`.

Значения box-sizing

`content-box`

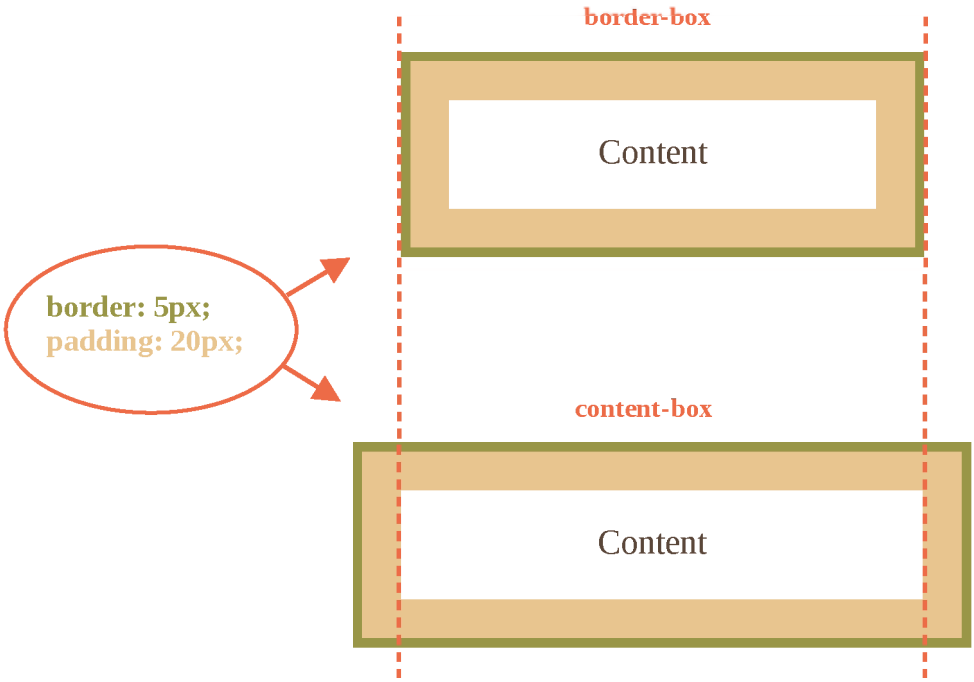
Это значение по умолчанию. В этом случае свойства `width/height` обозначают то, что находится *внутри* `padding`.

`border-box`

Значения `width/height` задают высоту/ширину *всего элемента*.

Для большей наглядности посмотрим на картинку этого `div` в зависимости от `box-sizing`:

```
/*+ no-beautify */
div {
  width: 200px;
  height: 100px;
  box-sizing: border-box (вверху) | content-box (внизу);

  padding: 20px;
  border:5px solid brown;
}
```



В верхнем случае браузер нарисовал весь элемент размером в `width x height`, в нижнем – интерпретировал `width/height` как размеры внутренней области.

Исторически сложилось так, что по умолчанию принят `content-box`, а `border-box` некоторые браузеры используют если не указан `DOCTYPE`, в режиме совместимости.

Но есть как минимум один случай, когда явное указание `border-box` может быть полезно: растягивание элемента до ширины родителя.

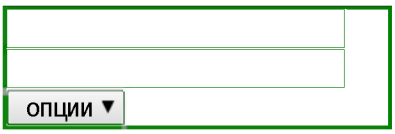
Пример: подстроить ширину к родителю

Задача: подогнать элемент по ширине внешнего элемента, чтобы он заполнял всё его пространство. Без привязки к конкретному размеру элемента в пикселях.

Например, мы хотим, чтобы элементы формы ниже были одинакового размера:

```
<style>
  form {
    width: 200px;
    border: 2px solid green;
  }

  form input,
  form select {
    display: block;
    padding-left: 5px;
```

```
    /* padding для красоты */
  }
</style>

<form>
  <input>
  <input>
  <select>
    <option>опции</option>
  </select>
</form>
```



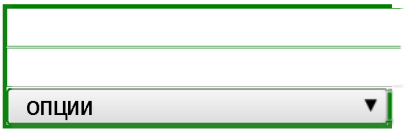
Как сделать, чтобы элементы растянулись чётко по ширине `FORM`? Попробуйте добиться этого самостоятельно, перед тем как читать дальше.

Попытка width:100%

Первое, что приходит в голову – поставить всем `INPUT'ам` ширину `width: 100%`.

Попробуем:

```
<style>
  form {
    width: 200px;
    border: 2px solid green;
  }

  form input, form select {
    display: block;
    padding-left: 5px;
    width: 100%;
  }
</style>

<form>
  <input>
  <input>
  <select><option>опции</option></select>
</form>
```



Как видно, не получается. **Элементы вылезают за пределы родителя.**

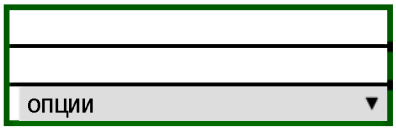
Причина – ширина элемента 100% по умолчанию относится к внутренней области, не включающей `padding` и `border`. То есть, внутренняя область растягивается до `100%` родителя, и к ней снаружи прибавляются `padding/border`, которые и вылезают.

Есть два решения этой проблемы.

Решение: дополнительный элемент

Можно убрать `padding/border` у элементов `INPUT/SELECT` и завернуть каждый из них в дополнительный `DIV`, который будет обеспечивать дизайн:

```
<style>
  form {
    width: 200px;
    border: 2px solid green;
  }
  /* убрать padding/border */

  form input,
  form select {
    padding: 0;
    border: 0;
    width: 100%;
  }
  /* внешний div даст дизайн */

  form div {
    padding-left: 5px;
    border: 1px solid black;
  }
</style>

<form>
  <div>
    <input>
  </div>
  <div>
    <input>
  </div>
  <div>
    <select>
      <option>опции</option>
    </select>
  </div>
</form>
```



В принципе, это работает. Но нужны дополнительные элементы. А если мы делаем дерево или большую редактируемую таблицу, да и вообще – любой интерфейс, где элементов и так много, то лишние нам точно не нужны.

Кроме того, такое решение заставляет пожертвовать встроенным в браузер дизайном элементов `INPUT/SELECT`.

Решение: box-sizing

Существует другой способ, гораздо более естественный, чем предыдущий.

При помощи `box-sizing: border-box` мы можем сказать браузеру, что ширина, которую мы ставим, относится к элементу полностью, включая `border` и `padding`:

```
<style>
  form {
    width: 200px;
    border: 2px solid green;
  }

  form input, form select {
    display: block;
    padding-left: 5px;
    -moz-box-sizing: border-box; /* в Firefox нужен префикс */
    box-sizing: border-box;
    width: 100%;
  }
</style>

<form>
  <input>
  <input>
  <select><option>опции</option></select>
</form>
```

Мы сохранили «родную» рамку вокруг `INPUT/SELECT` и не добавили лишних элементов. Всё замечательно.

Свойство `box-sizing` поддерживается в IE начиная с версии 8.

Свойство margin

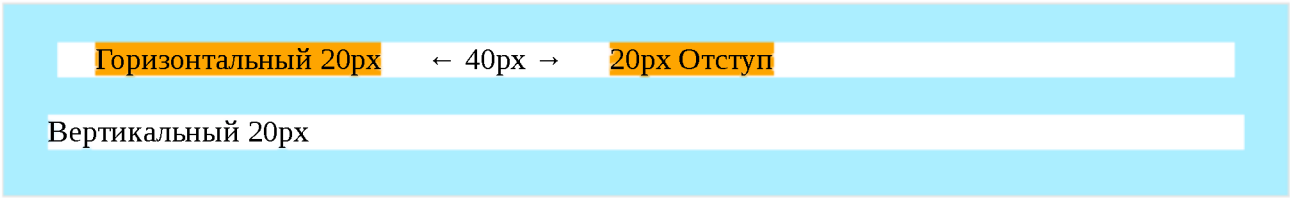
Свойство `margin` задаёт отступы вокруг элемента. У него есть несколько особенностей, которые мы здесь рассмотрим.

Объединение отступов

Вертикальные отступы поглощают друг друга, горизонтальные – нет.

Например, вот документ с вертикальными и горизонтальными отступами:

```
<body style="background: #aef">
  <p style="margin:20px; background:white">

    <span style="margin:20px; background:orange">Горизонтальный 20px</span>
    ← 40px →
    <span style="margin:20px; background:orange">20px Отступ </span>

  </p>
  <p style="margin:15px; background:white">Вертикальный 20px</p>
</body>
```



Расстояние по горизонтали между элементами `SPAN` равно `40px`, так как горизонтальные отступы по `20px` сложились.

А вот по вертикали расстояние от `SPAN` до `P` равно `20px`: из двух вертикальных отступов выбирается больший `max(20px, 15px) = 20px` и применяется.

Отрицательные margin-top/left

Отрицательные значения `margin-top/margin-left` смещают элемент со своего обычного места.

В CSS есть другой способ добиться похожего эффекта – а именно, `position:relative`. Но между ними есть одно принципиальное различие.

При сдвиге через `margin` соседние элементы занимают освободившееся пространство, в отличие от `position: relative`, при котором элемент визуально сдвигается, но место, где он был, остаётся «занятым».

То есть, элемент продолжает полноценно участвовать в потоке.

Пример: вынос заголовка

Например, есть документ с информационными блоками:

```
<style>
  div {
    border: 1px solid blue;
    margin: 2em;
    font: .8em/1.25 sans-serif;
  }

  h2 {
    background: #aef;
    margin: 0 0 0.8em 0;
  }
</style>

<div>
  <h2>Общие положения</h2>

  <p>Настоящие Правила дорожного движения устанавливают единый порядок дорожного дв
</div>

<div>
  <h2>Общие обязанности водителей</h2>

  <p>Водитель механического транспортного средства обязан иметь при себе и по требо
  <ul>
    <li>водительское удостоверение на право управления транспортным средством соотв
    <li>...и так далее...</li>
  </ul>
</div>
```

Общие положения

Настоящие Правила дорожного движения устанавливают единый порядок дорожного движения на всей территории Российской Федерации. Другие нормативные акты, касающиеся дорожного движения, должны основываться на требованиях Правил и не противоречить им.

Общие обязанности водителей

Водитель механического транспортного средства обязан иметь при себе и по требованию сотрудников милиции передавать им для проверки:

- водительское удостоверение на право управления транспортным средством соответствующей категории;
- ...и так далее...

Использование отрицательного `margin-top` позволяет вынести заголовки над блоком.

```
/* вверх чуть больше, чем на высоту строки (1.25em) */

h2 {
  margin-top: -1.3em;
}
```

Результат:

Общие положения

Настоящие Правила дорожного движения устанавливают единый порядок дорожного движения на всей территории Российской Федерации. Другие нормативные акты, касающиеся дорожного движения, должны основываться на требованиях Правил и не противоречить им.

Общие обязанности водителей

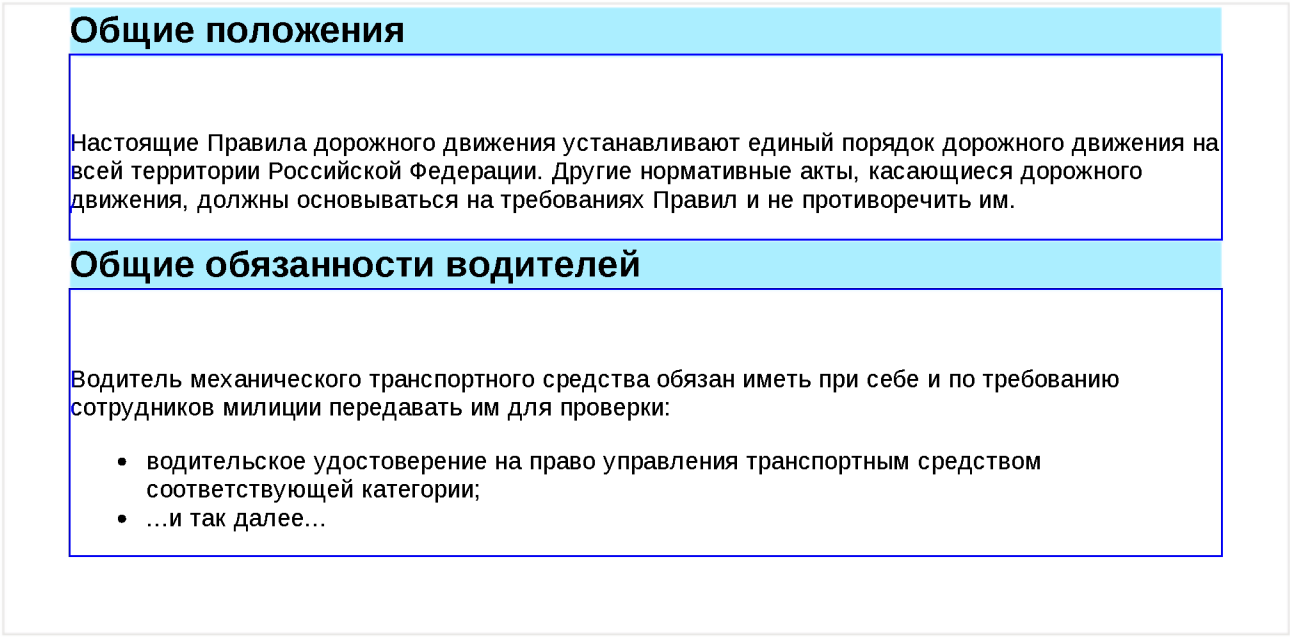
Водитель механического транспортного средства обязан иметь при себе и по требованию сотрудников милиции передавать им для проверки:

- водительское удостоверение на право управления транспортным средством соответствующей категории;
- ...и так далее...

А вот, что бы было при использовании `position`:

```
h2 {
  position: relative;
  top: -1.3em;
}
```

Результат:



При использовании `position`, в отличие от `margin`, на месте заголовков, внутри блоков, осталось пустое пространство.

Пример: вынос отчерка

Организуем информацию чуть по-другому. Пусть после каждого заголовка будет отчерк:

```
<div>
  <h2>Заголовок</h2>
  <hr>

  <p>Текст Текст Текст.</p>
</div>
```

Пример документа с такими отчерками:

Общие положения

Настоящие Правила дорожного движения устанавливают единый порядок дорожного движения на всей территории Российской Федерации. Другие нормативные акты, касающиеся дорожного движения, должны основываться на требованиях Правил и не противоречить им.

Общие обязанности водителей

Водитель механического транспортного средства обязан иметь при себе и по требованию сотрудников милиции передавать им для проверки:

- водительское удостоверение на право управления транспортным средством соответствующей категории;
- ...и так далее...

Для красоты мы хотим, чтобы отчерк `HR` начинался левее, чем основной текст. Отрицательный `margin-left` нам поможет:

```
/*+ no-beautify */
hr.margin { margin-left: -2em; }

/* для сравнения */
hr.position { position: relative; left: -2em; }
```

Результат:

Общие положения (hr.margin)

Настоящие Правила дорожного движения устанавливают единый порядок дорожного движения на всей территории Российской Федерации. Другие нормативные акты, касающиеся дорожного движения, должны основываться на требованиях Правил и не противоречить им.

Общие обязанности водителей (hr.position)

Водитель механического транспортного средства обязан иметь при себе и по требованию сотрудников милиции передавать им для проверки:

- водительское удостоверение на право управления транспортным средством соответствующей категории;
- ...и так далее...

Обратите внимание на разницу между методами сдвига!

- `hr.margin` сначала сдвинулся, а потом нарисовался до конца блока.

- `hr.position` сначала нарисовался, а потом сдвинулся – в результате справа осталось пустое пространство.

Уже отсюда видно, что отрицательные `margin` – исключительно полезное средство позиционирования!

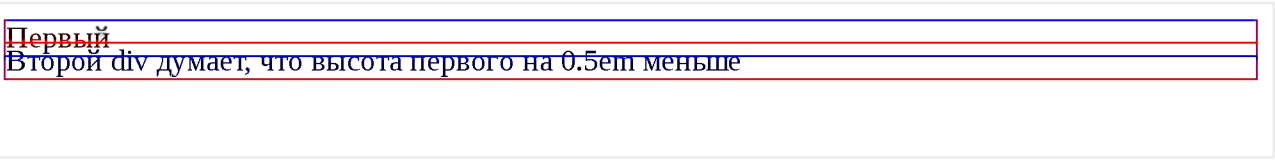
Отрицательные margin-right/bottom

Отрицательные `margin-right/bottom` ведут себя по-другому, чем `margin-left/top`. Они не сдвигают элемент, а «укорачивают» его.

То есть, хотя сам размер блока не уменьшается, но следующий элемент будет думать, что он меньше на указанное в `margin-right/bottom` значение.

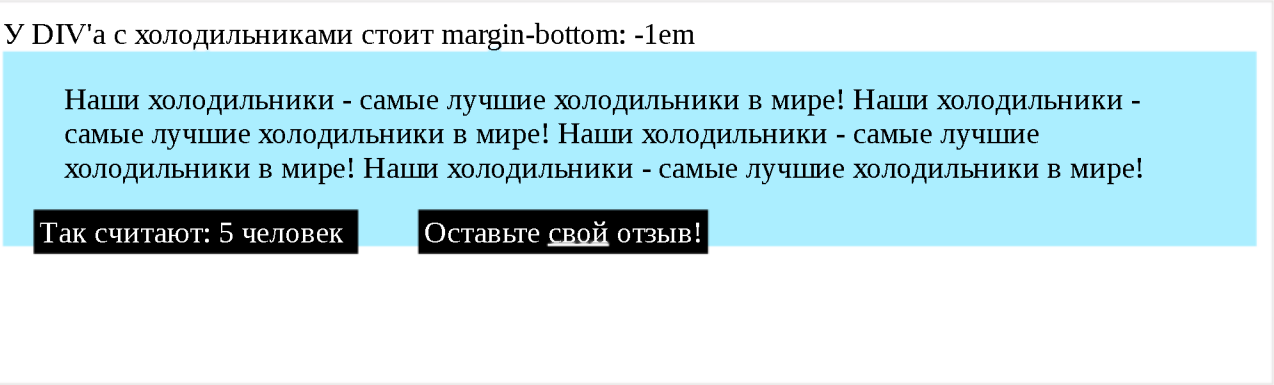
Например, в примере ниже вторая строка налезает на первую:

```
<div style="border: 1px solid blue; margin-bottom: -0.5em">
  Первый
</div>

<div style="border: 1px solid red">
  Второй div думает, что высота первого на 0.5em меньше
</div>
```



Это используют, в частности для красивых вносок, с приданием иллюзии глубины.

Например:



Итого

- Отрицательные `margin-left/top` сдвигают элемент влево-вверх. Остальные элементы это учитывают, в отличие от сдвига через `position`.
- Отрицательные `margin-right/bottom` заставляют другие элементы думать, что блок меньше по размеру справа-внизу, чем он на самом деле.

Отличная статья на тему отрицательных `margin`: The Definitive Guide to Using Negative Margins

Задачи

Нерабочие margin?

важность: 3

В примере ниже находится блок `.block` фиксированной высоты, а в нём – прямоугольник `.spacer`.

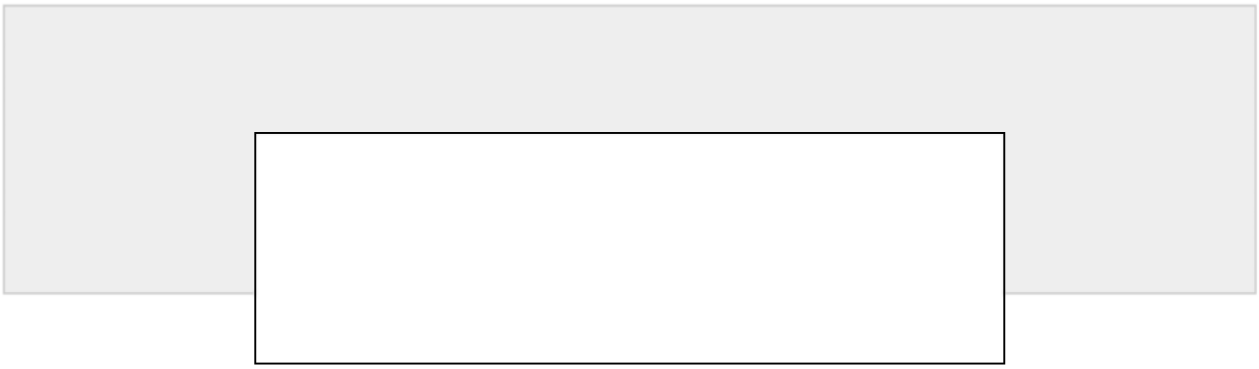
При помощи `margin-left: 20%` и `margin-right: 20%`, прямоугольник центрирован в родителе по горизонтали. Это работает.

Далее делается попытка при помощи свойств `height: 80%`, `margin-top: 10%` и `margin-bottom: 10%` расположить прямоугольник в центре по вертикали, чтобы сам элемент занимал `80%` высоты родителя, а сверху и снизу был одинаковый отступ.

Однако, как видите, это не получается. Почему? Как поправить?

```
<style>
  .block {
    height: 150px;

    border: 1px solid #CCC;
    background: #eee;
  }

  .spacer {
    margin-left: 20%;
    margin-right: 20%;

    height: 80%;
    margin-top: 10%;
    margin-bottom: 10%;

    border: 1px solid black;
    background: #FFF;
  }
</style>
```

```
<div class="block">
  <div class="spacer"></div>
</div>
```

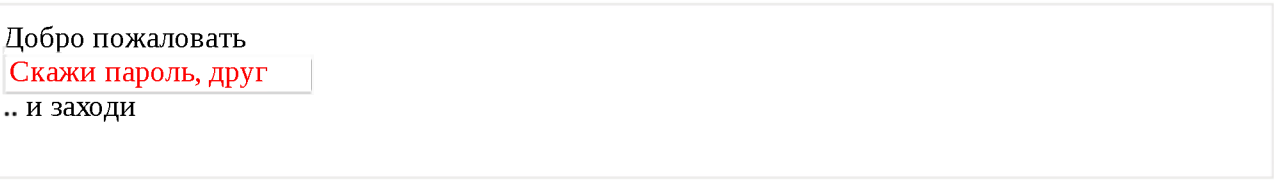


К решению

Расположить текст внутри INPUT

важность: 5

Создайте `<input type="password">` с цветной подсказкой внутри (должен правильно выглядеть, не обязан работать):



В дальнейшем мы сможем при помощи JavaScript сделать, чтобы текст при клике пропадал. Получится красивая подсказка.

P.S. Обратите внимание: `type="password"`! То есть, просто `value` использовать нельзя, будут звёздочки. Кроме того, подсказка, которую вы реализуете, может быть как угодно стилизована.

P.P.S. Вокруг `INPUT` с подсказкой не должно быть лишних отступов, блоки до и после продолжают идти в обычном потоке.

Открыть песочницу для задачи.

К решению

Лишнее место под IMG

Иногда под `IMG` «вдруг» появляется лишнее место. Посмотрим на эти грабли внимательнее, почему такое бывает и как этого избежать.

Демонстрация проблемы

Например:

```
<style>
  * {
    margin: 0;
    padding: 0;
  }
</style>
<table>
  <tr>
    <td style="border:1px red solid">
      <img src="square.png">
    </td>
  </tr>
</table>
```

Посмотрите внимательно! Вы видите расстояние между рамками снизу? Это потому, что **браузер резервирует дополнительное место под инлайновым элементом, чтобы туда выносить «хвосты» букв**.

Вот так выглядит та же картинка с выступающим вниз символом рядом:

В примере картинка обёрнута в красный `TD`. Таблица подстраивается под размер содержимого, так что проблема явно видна. Но она касается не только таблицы. Аналогичная ситуация возникнет, если вокруг `IMG` будет другой элемент с явно указанным размером, «облегающий» картинку по высоте. Браузер постарается зарезервировать место под `IMG`, хотя оно там не нужно.

Решение: сделать элемент блочным

Самый лучший способ избежать этого – поставить `display: block` таким картинкам:

```
<style>
  * {
    margin: 0;
    padding: 0;
  }

  img {
    display: block
  }
</style>
<table>
  <tr>
    <td style="border:1px red solid">
      <img src="square.png">
    </td>
  </tr>
</table>
```

Под блочным элементом ничего не резервируется. Проблема исчезла.

Решение: задать vertical-align

А что, если мы, по каким-то причинам, *не хотим* делать элемент блочным?

Существует ещё один способ избежать проблемы – использовать свойство vertical-align .

Если установить `vertical-align` в `top`, то инлайн-элемент будет отпозиционирован по верхней границе текущей строки.

При этом браузер не будет оставлять место под изображением, так как запланирует продолжение строки сверху, а не снизу:

```
<style>
  * {
    margin: 0;
    padding: 0;
  }

  img {
    vertical-align: top
  }
</style>
<table>
```

```
  <tr>
    <td style="border:1px red solid">
      <img src="square.png">
    </td>
  </tr>
</table>
```

А вот, как браузер отобразит соседние символы в этом случае: `p<img src="square.png">p`

С другой стороны, сама строка никуда не делась, изображение по-прежнему является её частью, а браузер планирует разместить другое текстовое содержимое рядом, хоть и сверху. Поэтому если изображение маленькое, то произойдёт дополнение пустым местом до высоты строки:

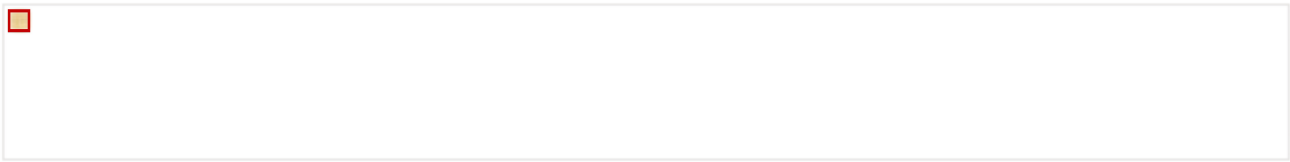
Например, для `<img src="square.png" style="height:10px">`:

Таким образом, требуется уменьшить ещё и `line-height` контейнера. Окончательное решение для маленьких изображений с `vertical-align`:

```
<style>
  * {
    margin: 0;
    padding: 0;
  }

  td {
    line-height: 1px
  }

  img {
    vertical-align: top
  }
</style>
```

```
<table>
  <tr>
    <td style="border:1px red solid">
      <img src="square.png" style="height:10px">
    </td>
  </tr>
</table>
```

Результат:



Итого

- Пробелы под картинками появляются, чтобы оставить место под «хвосты» букв в строке. Строка «подразумевается», т.к. `display:inline`.
- Можно избежать пробела, если изменить `display`, например, на `block`.
- Альтернатива: `vertical-align:top` (или `bottom`), но для маленьких изображений может понадобиться уменьшить `line-height`, чтобы контейнер не оставлял место под строку.

Свойство overflow

Свойство `overflow` управляет тем, как ведёт себя содержимое блочного элемента, если его размер превышает допустимую длину/ширину.

Обычно блок увеличивается в размерах при добавлении в него элементов, заключая в себе всех потомков.

Но что, если высота/ширина указаны явно? Тогда блок не может увеличиться, и содержимое «переполняет» блок. Его отображение в этом случае задаётся свойством `overflow`.

Возможные значения

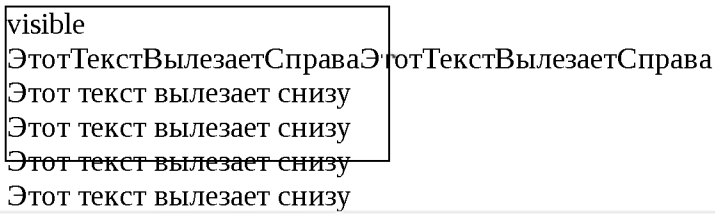
- `visible` (по умолчанию)
- `hidden`
- `scroll`
- `auto`

visible

Если не ставить `overflow` явно или поставить `visible`, то содержимое отображается за границами блока.

Например:

```
<style>
  .overflow {
    /* overflow не задан */
    width: 200px;
    height: 80px;
    border: 1px solid black;
  }
</style>

<div class="overflow">
  visible ЭтотТекстВылезаетСправаЭтотТекстВылезаетСправа
  Этот текст вылезает снизу Этот текст вылезает снизу
  Этот текст вылезает снизу Этот текст вылезает снизу
</div>
```

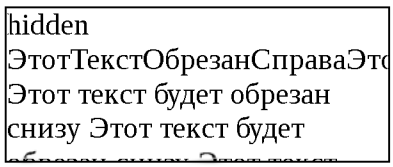


Как правило, такое переполнение указывает на ошибку в вёрстке. Если содержимое может быть больше контейнера – используют другие значения.

hidden

Переполняющее содержимое не отображается.

```
<style>
  .overflow {
    overflow: hidden;
    width: 200px;
    height: 80px;
    border: 1px solid black;
  }
</style>

<div class="overflow">
  hidden ЭтотТекстОбрезанСправаЭтотТекстОбрезанСправа
  Этот текст будет обрезан снизу Этот текст будет обрезан снизу
  Этот текст будет обрезан снизу Этот текст будет обрезан снизу
</div>
```



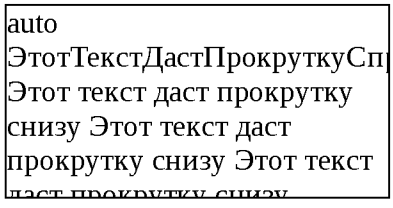
Вылезающее за границу содержимое становится недоступно.

Это свойство иногда используют от лени, когда какой-то элемент дизайна немного вылезает за границу, и вместо исправления вёрстки его просто скрывают. Как правило, долго оно не живёт, вёрстку всё равно приходится исправлять.

auto

При переполнении отображается полоса прокрутки.

```
<style>
  .overflow {
    overflow: auto;
    width: 200px;
    height: 100px;
    border: 1px solid black;
  }
</style>

<div class="overflow">
  auto ЭтотТекстДастПрокруткуСправаЭтотТекстДастПрокруткуСправа
  Этот текст даст прокрутку снизу Этот текст даст прокрутку снизу
  Этот текст даст прокрутку снизу
</div>
```

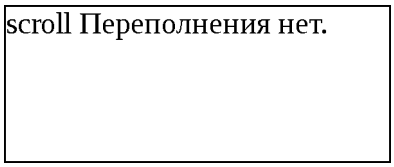


scroll

Полоса прокрутки отображается всегда.

```
<style>
  .overflow {
    overflow: scroll;
    width: 200px;
    height: 80px;
    border: 1px solid black;
```

```
  }
</style>

<div class="overflow">
  scroll
  Переполнения нет.
</div>
```

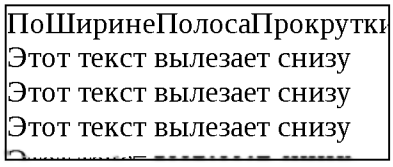


То же самое, что `auto`, но полоса прокрутки видна всегда, даже если переполнения нет.

overflow-x, overflow-y

Можно указать поведение блока при переполнении по ширине в `overflow-x` и высоте – в `overflow-y`:

```
<style>
  .overflow {
    overflow-x: auto;
    overflow-y: hidden;
    width: 200px;
    height: 80px;
    border: 1px solid black;
  }
</style>

<div class="overflow">
  ПоШиринеПолосаПрокруткиAutoПоШиринеПолосаПрокруткиAuto
  Этот текст вылезает снизу Этот текст вылезает снизу
  Этот текст вылезает снизу Этот текст вылезает снизу
</div>
```



Итого

Свойства `overflow-x/overflow-y` (или оба одновременно: `overflow`) задают поведение контейнера при переполнении:

`visible`

По умолчанию, содержимое вылезает за границы блока.

`hidden`

Переполняющее содержимое невидимо.

`auto`

Полоса прокрутки при переполнении.

`scroll`

Полоса прокрутки всегда.

Кроме того, значение `overflow: auto | hidden` изменяет поведение контейнера, содержащего `float`'ы. Так как элемент с `float` находится вне потока, то обычно контейнер не выделяет под него место. Но если стоит такой `overflow`, то место выделяется, т.е. контейнер растягивается. Более подробно этот вопрос рассмотрен в статье Свойство float.

Особенности свойства height в %

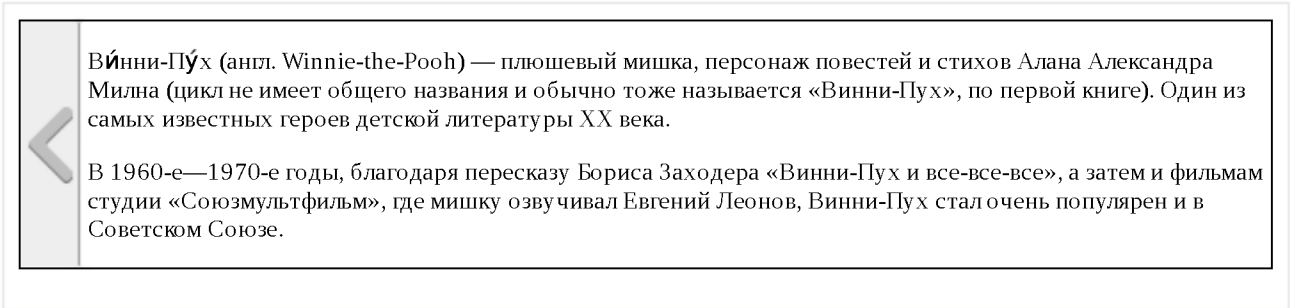
Обычно свойство `height`, указанное в процентах, означает высоту относительно внешнего блока.

Однако, всё не так просто. Интересно, что для произвольного блочного элемента `height` в процентах работать не будет!

Чтобы лучше понимать ситуацию, рассмотрим пример.

Пример

Наша цель – получить вёрстку такого вида:



При этом блок с левой стрелкой должен быть отдельным элементом внутри контейнера.

Это удобно для интеграции с JavaScript, чтобы отлавливать на нём клики мыши.

То есть, HTML-код требуется такой:

```
<div class="container">
  <div class="toggler">
    <!-- стрелка влево при помощи CSS, ширина фиксирована -->
  </div>
  <div class="content">
    ...Текст...
  </div>
</div>
```

Как это реализовать? Подумайте перед тем, как читать дальше…

Придумали?.. Если да – продолжаем.

Есть разные варианты, но, возможно, вы решили сдвинуть `.toggler` влево, при помощи `float:left` (тем более что он фиксированной ширины) и увеличить до `height: 100%`, чтобы он занял всё пространство по вертикали.

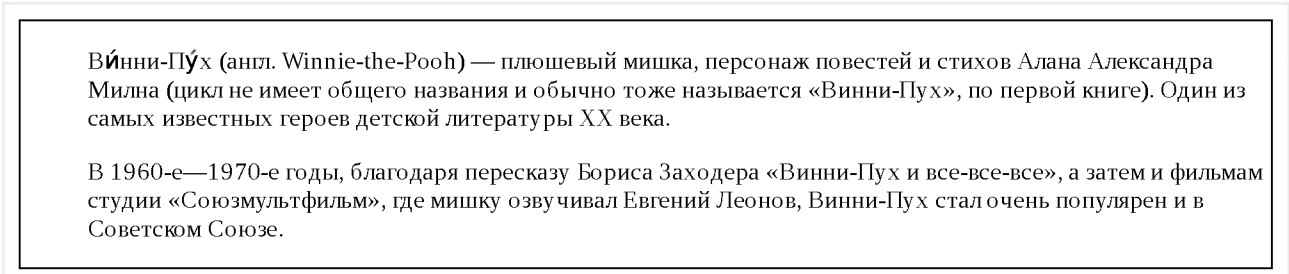
Вы ещё не видите подвох? Смотрим внимательно, что будет происходить с `height: 100%`...

Демо height:100% + float:left

CSS:

```
.container {
  border: 1px solid black;
}

.content {
  /* margin-left нужен, так как слева от содержимого будет стрелка */
  margin-left: 35px;
}

.toggler {
  /* Зададим размеры блока со стрелкой */
  height: 100%;
  width: 30px;
  float: left;

  background: #EEE url("arrow_left.png") center center no-repeat;
  border-right: #AAA 1px solid;
  cursor: pointer;
}
```

А теперь – посмотрим этот вариант в действии:



Как видно, блок со стрелкой вообще исчез! Куда же он подевался?

Ответ нам даст спецификация CSS 2.1 пункт 10.5 .

"Если высота внешнего блока вычисляется по содержимому, то высота в % не работает, и заменяется на `height:auto`. Кроме случая, когда у элемента стоит `position:absolute`."

В нашем случае высота `.container` как раз определяется по содержимому, поэтому для `.toggler` проценты не действуют, а размер вычисляется как при `height:auto`.

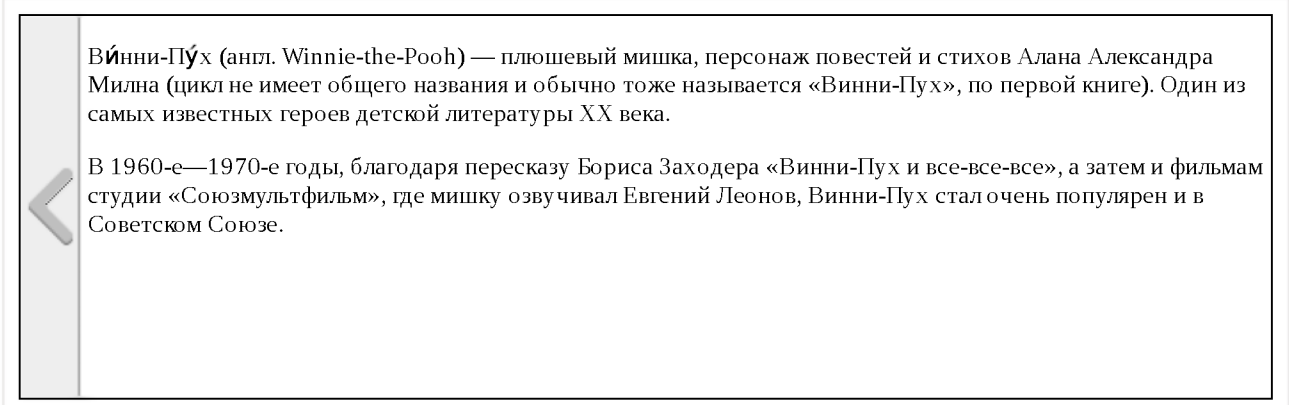
Какая же она – эта автоматическая высота? Вспоминаем, что обычно размеры `float` определяются по содержимому (10.3.5). А содержимого-то в `.toggler` нет, так что высота нулевая. Поэтому этот блок и не виден.

Если бы мы точно знали высоту внешнего блока и добавили её в CSS – это решило бы проблему.

Например:

```
/*+ no-beautify */
.container {
  height: 200px; /* теперь height в % внутри будет работать */
}
```

Результат:



…Но в данном случае этот путь неприемлем! Ведь мы не знаем, сколько будет текста и какой будет итоговая высота.

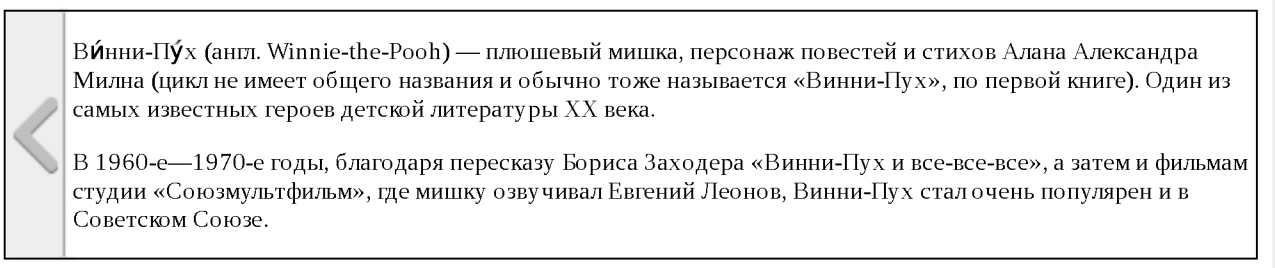
Поэтому решим задачу по-другому.

Правильно: height:100% + position:absolute

Проценты будут работать, если поставить `.toggler` свойство `position: absolute` и спозиционировать его в левом-верхнем углу `.container` (у которого сделать `position:relative`):

```
.container {
  position: relative;
  border: 1px solid black;
}

.content {
  margin-left: 35px;
}

.toggler {
  position: absolute;
  left: 0;
  top: 0;

  height: 100%;
  width: 30px;
  cursor: pointer;

  border-right: #AAA 1px solid;
  background: #EEE url("arrow_left.png") center center no-repeat;
}
```

Результат:



Проблема с height: 100%, проявляющаяся, когда у родительского элемента не установлен height, но указан min-height

Вам необходимо установить height: 1px для родителя, чтобы дочерний элемент смог занять всю высоту указанную в min-height.

```
.parent {
    min-height: 300px;
    height: 1px; /* Требуется, чтобы дочерний блок взял высоту 100% */
}

.child {
    height: 100%;
}
```

Итого

- Свойство `height`, указанное в %, работает только если для внешнего блока указана высота.
- Стандарт CSS 2.1 предоставляет обход этой проблемы, отдельно указывая, что проценты работают при `position:absolute`. На практике это часто выручает.
- Если у родительского элемента не установлено height, а указано min-height, то, чтобы дочерний блок занял 100% высоты, нужно родителю поставить свойство height: 1px;

Знаете ли вы селекторы?

CSS3-селекторы – фундаментально полезная вещь.

Даже если вы почему-то (старый IE?) не пользуетесь ими в CSS, есть много фреймворков для их кросс-браузерного использования CSS3 из JavaScript.

Поэтому эти селекторы необходимо знать.

Основные виды селекторов

Основных видов селекторов всего несколько:

- `*` – любые элементы.
- `div` – элементы с таким тегом.
- `#id` – элемент с данным `id`.
- `.class` – элементы с таким классом.
- `[name="value"]` – селекторы на атрибут (см. далее).
- `:visited` – «псевдоклассы», остальные разные условия на элемент (см. далее).

Селекторы можно комбинировать, записывая последовательно, без пробела:

- `.c1.c2` – элементы одновременно с двумя классами `c1` и `c2`
- `a#id.c1.c2:visited` – элемент `a` с данным `id`, классами `c1` и `c2`, и псевдоклассом `visited`

Отношения

В CSS3 предусмотрено четыре вида отношений между элементами.

Самые известные вы наверняка знаете:

- `div p` – элементы `p`, являющиеся потомками `div`.
- `div > p` – только непосредственные потомки

Есть и два более редких:

- `div ~ p` – правые соседи: все `p` на том же уровне вложенности, которые идут после `div`.
- `div + p` – первый правый сосед: `p` на том же уровне вложенности, который идёт сразу после `div` (если есть).

Посмотрим их на примере HTML:

```
<h3>Балтославянские языки</h3>

<ol id="languages">
  ...Вложенный OL/LI список языков...
</ol>
```

CSS-селекторы:

```
/*+ no-beautify */
#languages li {
  color: brown;   /* потомки #languages, подходящие под селектор  LI */
}

#languages > li {
  color: black;   /* первый уровень детей #languages подходящих под LI */
}

#e-slavic { font-style: italic; }

#e-slavic ~ li {  /* правые соседи #e-slavic с селектором LI */
  color: red;
```

```
}

#latvian {
  font-style: italic;
}

#latvian * {      /* потомки #latvian, подходяще под * (т.е. любые) */
  font-style: normal;
}

#latvian + li {  /* первый правый сосед #latvian с селектором LI */
 color: green;
}
```

Результат:

Балтославянские языки

1. Славянские языки
    1. Славянские микроязыки
    2. Праславянский язык
    3. *Восточнославянские языки* `#e-slavic`
    4. Западнославянские языки `#e-slavic ~ li`
    5. Южнославянские языки `#e-slavic ~ li`
    6. ... `#e-slavic ~ li`
2. Балтийские языки
    1. *Литовский язык*
    2. *Латышский язык* `#latvian`
        1. Латгальский язык `#latvian *`
    3. Прусский язык `#latvian + li`
    4. ... (следующий элемент уже не `#latvian + li`)

Фильтр по месту среди соседей

При выборе элемента можно указать его место среди соседей.

Список псевдоклассов для этого:

- `:first-child` – первый потомок своего родителя.
- `:last-child` – последний потомок своего родителя.
- `:only-child` – единственный потомок своего родителя, соседних элементов нет.
- `:nth-child(a)` – потомок номер `a` своего родителя, например `:nth-child(2)` – второй потомок. Нумерация начинается с `1`.
- `:nth-child(an+b)` – расширение предыдущего селектора через указание номера потомка формулой, где `a, b` – константы, а под `n` подразумевается любое целое число.

Этот псевдокласс будет фильтровать все элементы, которые попадают под формулу при каком-либо `n`. Например: - `:nth-child(2n)` даст элементы номер `2`, `4`, `6`..., то есть чётные.

- `:nth-child(2n+1)` даст элементы номер `1`, `3`..., то есть нечётные.
- `:nth-child(3n+2)` даст элементы номер `2`, `5`, `8` и так далее.

Пример использования для выделения в списке:

- Древнерусский язык
- Древненовгородский диалект `li:nth-child(2n)`
- Западнорусский письменный язык `li:nth-child(3)`
- Украинский язык `li:nth-child(2n)`
- Белорусский язык
- Другие языки `li:nth-child(2n)`

```
/*+ hide="CSS к примеру выше"  no-beautify */
li:nth-child(2n) { /* чётные */
  background: #eee;
}

li:nth-child(3) {  /* 3-ий потомок */
  color: red;
}
```

- `:nth-last-child(a)`, `:nth-last-child(an+b)` – то же самое, но отсчёт начинается с конца, например `:nth-last-child(2)` – второй элемент с конца.

Фильтр по месту среди соседей с тем же тегом

Есть аналогичные псевдоклассы, которые учитывают не всех соседей, а только с тем же тегом:

- `:first-of-type`
- `:last-of-type`
- `:only-of-type`

- `:nth-of-type`
- `:nth-last-of-type`

Они имеют в точности тот же смысл, что и обычные `:first-child`, `:last-child` и так далее, но во время подсчёта игнорируют элементы с другими тегами, чем тот, к которому применяется фильтр.

Пример использования для раскраски списка `DT` «через один» и предпоследнего `DD`:

Первый dt
Описание dd
Второй dt `dt:nth-of-type(2n)`
Описание dd
Третий dt
Описание dd `dd:nth-last-of-type(2)`
Четвёртый dt `dt:nth-of-type(2n)`
Описание dd

```
/*+ hide="CSS к примеру выше"  no-beautify */
dt:nth-of-type(2n) {
  /* чётные dt (соседи с другими тегами игнорируются) */
  background: #eee;
}

dd:nth-last-of-type(2) {
  /* второй dd снизу */
  color: red;
}
```

Как видим, селектор `dt:nth-of-type(2n)` выбрал каждый второй элемент `dt`, причём другие элементы (`dd`) в подсчётах не участвовали.

Селекторы атрибутов

На атрибут целиком:

- `[attr]` – атрибут установлен,
- `[attr="val"]` – атрибут равен `val`.

На начало атрибута:

- `[attr^="val"]` – атрибут начинается с `val`, например `"value"`.
- `[attr|="val"]` – атрибут равен `val` *или* начинается с `val-`, например равен `"val-1"`.

На содержание:

- `[attr*="val"]` – атрибут содержит подстроку `val`, например равен `"myvalue"`.
- `[attr~="val"]` – атрибут содержит `val` как одно из значений через пробел.

Например: `[attr~="delete"]` верно для `"edit delete"` и неверно для `"undelete"` или `"no-delete"`.

На конец атрибута:

- `[attr$="val"]` – атрибут заканчивается на `val`, например равен `"myval"`.

Другие псевдоклассы

- `:not(селектор)` – все, кроме подходящих под селектор.
- `:focus` – в фокусе.
- `:hover` – под мышью.
- `:empty` – без детей (даже без текстовых).
- `:checked`, `:disabled`, `:enabled` – состояния `INPUT`.
- `:target` – этот фильтр сработает для элемента, `ID` которого совпадает с анкором `#...` текущего URL.

  Например, если на странице есть элемент с `id="intro"`, то правило `:target { color: red }` подсветит его в том случае, если текущий URL имеет вид `http://...#intro`.

Псевдоэлементы ::before, ::after

«Псевдоэлементы» – различные вспомогательные элементы, которые браузер записывает или может записать в документ.

При помощи *псевдоэлементов* `::before` и `::after` можно добавлять содержимое в начало и конец элемента:

```
<style>
  li::before {
    content: " [[ ";
  }

  li::after {
    content: " ]] ";
  }
</style>

Обратите внимание: содержимое добавляется <b>внутрь</b> LI.

<ul>
  <li>Первый элемент</li>
  <li>Второй элемент</li>
</ul>
```

Обратите внимание: содержимое добавляется **внутрь** LI.

- [[Первый элемент]]
- [[Второй элемент]]

Псевдоэлементы `::before`/`::after` добавили содержимое в начало и конец каждого `LI`.

`:before` или `::before`?

Когда-то все браузеры реализовали эти псевдоэлементы с одним двоеточием: `:after`/`:before`.

Стандарт с тех пор изменился и сейчас все, кроме IE8, понимают также современную запись с двумя двоеточиями. А для IE8 нужно по-прежнему одно.

Поэтому если вам важна поддержка IE8, то имеет смысл использовать одно двоеточие.

Практика

Вы можете использовать информацию выше как справочную для решения задач ниже, которые уже реально покажут, владеете вы CSS-селекторами или нет.

Задачи

Выберите элементы селектором

важность: 5

HTML-документ:

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

<h3 id="widget-title">Сообщения:</h3>
<ul id="messages">
  <li id="message-1">Сообщение 1</li>
  <li id="message-2">Сообщение 2</li>
  <li id="message-3" data-action="delete">Сообщение 3</li>
  <li id="message-4" data-action="edit do-not-delete">Сообщение 4</li>
  <li id="message-5" data-action="edit delete">Сообщение 5</li>
  <li><a href="#">...</a></li>
</ul>

<a href="http://site.com/list.zip">Ссылка на архив</a>
<a href="http://site.com/list.pdf">..И на PDF</a>
```

Задания:

1. Выбрать `input` типа `checkbox`.
2. Выбрать `input` типа `checkbox`, НЕ отмеченный.
3. Найти все элементы с `id=message` или `message-*`.
4. Найти все элементы с `id=message-*`.
5. Найти все ссылки с расширением `href="...zip"`.
6. Найти все элементы с атрибутом `data-action`, содержащим `delete` в списке (через пробел).
7. Найти все элементы, у которых ЕСТЬ атрибут `data-action`, но он НЕ содержит `delete` в списке (через пробел).
8. Выбрать все чётные элементы списка `#messages`.
9. Выбрать один элемент сразу за заголовком `h3#widget-title` на том же уровне вложенности.
10. Выбрать все ссылки, следующие за заголовком `h3#widget-title` на том же уровне вложенности.
11. Выбрать ссылку внутри последнего элемента списка `#messages`.

Открыть песочницу для задачи.

К решению

Отступ между элементами, размер одна строка

важность: 4

Есть список `UL/LI`.

```
Текст вверху без отступа от списка.
<ul>
  <li>Маша</li>
  <li>Паша</li>
  <li>Даша</li>
  <li>Женя</li>
  <li>Саша</li>
  <li>Гоша</li>
</ul>
Текст внизу без отступа от списка.
```

Размеры шрифта и строки заданы стилем:

```
body {
  font: 14px/1.5 serif;
}
```

Сделайте, чтобы между элементами был вертикальный отступ.

- Размер отступа: ровно 1 строка.
- Нужно добавить только одно правило CSS с одним псевдоселектором, можно использовать CSS3.
- Не должно быть лишних отступов сверху и снизу списка.

Результат:

Текст вверху без отступа от списка.

- Маша
- Паша
- Даша
- Женя
- Саша
- Гоша

Текст внизу без отступа от списка.

Открыть песочницу для задачи.

К решению

Отступ между парами, размером со строку

важность: 4

Есть список `UL/LI`.

```
Текст вверху без отступа от списка.
<ul>
  <li>Маша</li>
  <li>Паша</li>
  <li>Даша</li>
  <li>Женя</li>
  <li>Саша</li>
  <li>Гоша</li>
</ul>
Текст внизу без отступа от списка.
```

Размеры шрифта и строки заданы стилем:

```
body {
  font: 14px/1.5 serif;
}
```

Сделайте, чтобы между каждой парой элементов был вертикальный отступ.

- Размер отступа: ровно 1 строка.
- Нужно добавить только одно правило CSS, можно использовать CSS3.
- Не должно быть лишних отступов сверху и снизу списка.

Результат:

Текст вверху без отступа от списка.

- Маша
- Паша

- Даша
- Женя

- Саша
- Гоша

Текст внизу без отступа от списка.

Открыть песочницу для задачи.

К решению

CSS-спрайты

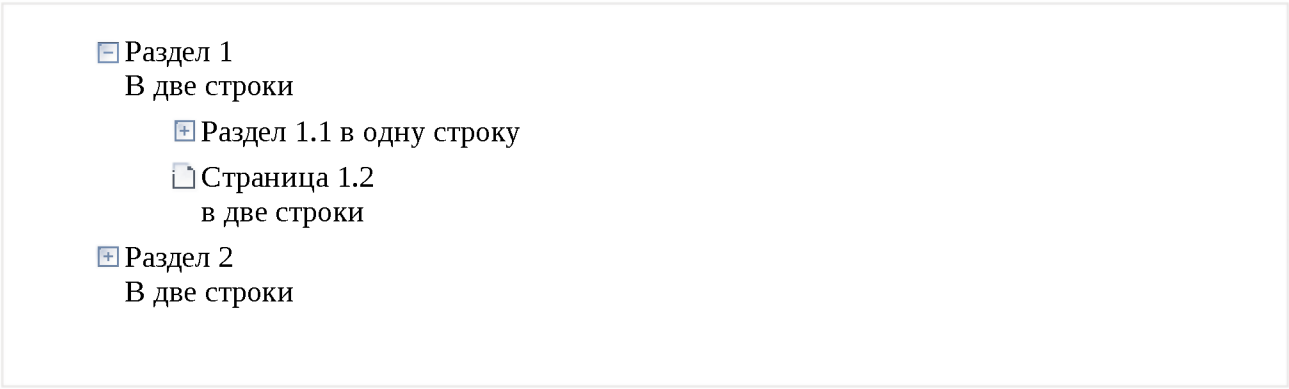
CSS-спрайт – способ объединить много изображений в одно, чтобы:

1. Сократить количество обращений к серверу.
2. Загрузить несколько изображений сразу, включая те, которые понадобятся в будущем.
3. Если у изображений сходная палитра, то объединённое изображение будет меньше по размеру, чем совокупность исходных картинок.

Рассмотрим, как это работает, на примере дерева:

```
<ul>
  <li class="open">
    <div class="icon"></div>
    <div class="text">Раздел 1
      <br>В две строки</div>
    <ul>
      <li class="closed">
        <div class="icon"></div>
        <div class="text">Раздел 1.1 в одну строку</div>
      </li>
      <li class="leaf">
        <div class="icon"></div>
        <div class="text">Страница 1.2
          <br> в две строки</div>
      </li>
    </ul>
```

```
  </li>
  <li class="closed">
    <div class="icon"></div>
    <div class="text">Раздел 2
      <br>В две строки</div>
  </li>
</ul>
```



Сейчас «плюс», «минус» и «статья» – три отдельных изображения. Объединим их в спрайт.

Шаг 1. Использовать background

Первый шаг к объединению изображений в «спрайт» – показывать их через `background`., а не через тег `IMG`.

В данном случае он уже сделан. Стиль для дерева:

```
.icon {
  width: 16px;
  height: 16px;
  float: left;
}

.open .icon {
  cursor: pointer;
  background: url(minus.gif);
}

.closed .icon {
  cursor: pointer;
  background: url(plus.gif);
}

.leaf .icon {
  cursor: text;
  background: url(article.gif);
}
```

Шаг 2. Объединить изображения

Составим из нескольких изображений одно `icons.gif`, расположив их, например, по вертикали.

Из , и получится одна картинка:

Шаг 3. Показать часть спрайта в «окошке»

А теперь самое забавное. Размер `DIV'а` для иконки – жёстко фиксирован:

```
/*+ no-beautify */
.icon {
  width: 16px;
  height: 16px;
  float: left;
}
```

Это значит, что если поставить `background'ом` объединённую картинку, то вся она не поместится, будет видна только верхняя часть:

Пример раздела

Если бы высота иконки была больше, например, `16x48`, как в примере ниже, то было бы видно и остальное:

Пример раздела

…Но так как там всего `16px`, то помещается только одно изображение.

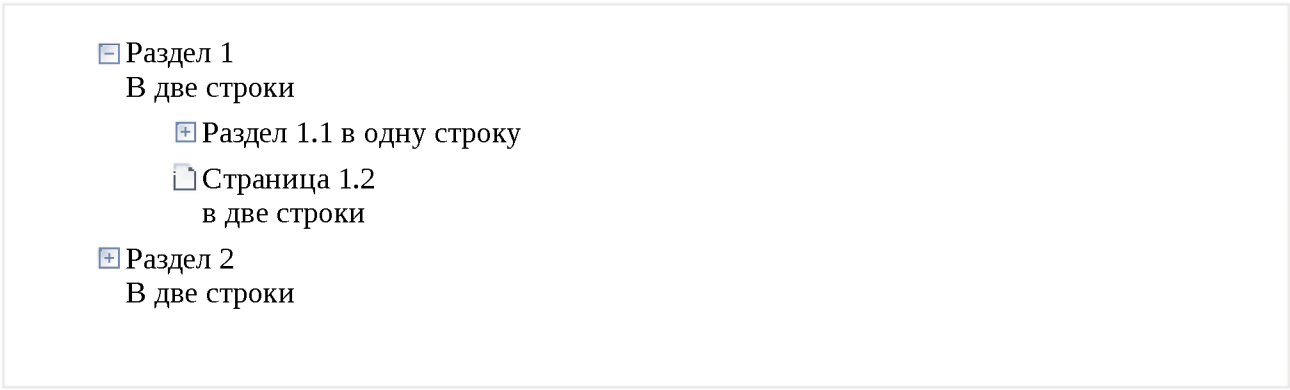
Шаг 4. Сдвинуть спрайт

Сдвиг фона `background-position` позволяет выбирать, какую именно часть спрайта видно.

В спрайте `icons.gif` изображения объединены так, что сдвиг на `16px` покажет следующую иконку:

```
/*+ no-beautify */
.icon {
  width: 16px;
  height: 16px;
  float: left;
  background: url(icons.gif) no-repeat;
}

.open .icon {
  background-position: 0 -16px; /* вверх на 16px */
  cursor: pointer;
}

.closed .icon {
  background-position: 0 0px; /* по умолчанию */
  cursor: pointer;
}

.leaf .icon {
  background-position: 0 -32px; /* вверх на 32px */
  cursor: text;
}
```

Результат:



- В спрайт могут объединяться изображения разных размеров, т.е. сдвиг может быть любым.
- Сдвигать можно и по горизонтали и по вертикали.

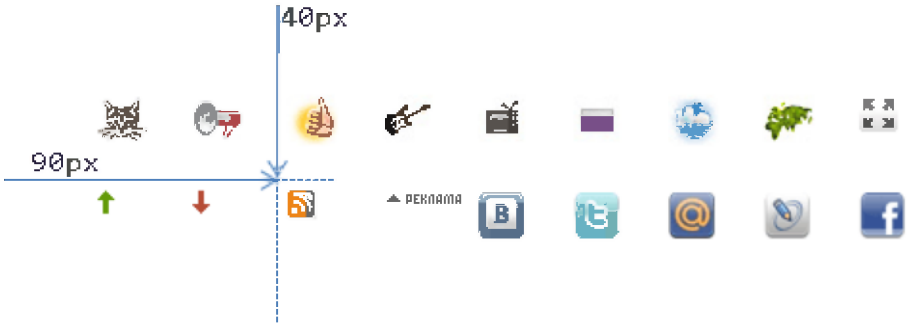
Отступы

Обычно отступы делаются `margin/padding`, но иногда их бывает удобно предусмотреть в спрайте.

Тогда если элемент немного больше, чем размер изображения, то в «окошке» не появится лишнего.

Пример спрайта с отступами:

Иконка RSS находится в нём на координатах `(90px, 40px)`:



Это значит, что чтобы показать эту иконку, нужно сместить фон:

```
background-position: -90px -40px;
```

При этом в левом-верхнем углу фона как раз и будет эта иконка:

Очень интересная новость Очень интересная новость Очень интересная новость Очень интересная новость Очень интересная новость Очень интересная новость Очень интересная новость Очень интересная новость Очень интересная новость

Элемент, в котором находится иконка (в рамке), больше по размеру, чем картинка.

Его стиль:

```
/*+ no-beautify */
.rss {
  width: 35px; /* ширина/высота больше чем размер иконки */
  height: 35px;
  border: 1px solid black;
  float: left;
```

```
  background-image: url(sprite.png);
  background-position: -90px -40px;
}
```

Если бы в спрайте не было отступов, то в такое большое «окошко» наверняка влезли бы другие иконки.

Итого

Когда использовать для изображений `IMG`, а когда – `CSS background`?

Решение лучше всего принимать, исходя из принципов семантической вёрстки.

Задайте вопрос – что здесь делает изображение? Является ли оно самостоятельным элементом страницы (фотография, аватар посетителя), или же оформляет что-либо (иконка узла дерева)?

Элемент `IMG` следует использовать в первом случае, а для оформления у нас есть CSS.

Спрайты позволяют:

1. Сократить количество обращений к серверу,
2. Загрузить несколько изображений сразу, включая те, которые понадобятся в будущем.
3. Если у изображений сходная палитра, то объединённое изображение будет меньше по размеру, чем совокупность исходных картинок.

Если фоновое изображение нужно повторять по горизонтали или вертикали, то спрайты тоже подойдут – изображения в них нужно располагать в этом случае так, чтобы при повторении не были видны соседи, т.е., соответственно, вертикально или горизонтально, но не «решёткой».

Далее мы встретимся со спрайтами при создании интерфейсов, чтобы кнопка при наведении меняла своё изображение. Один спрайт будет содержать все состояния кнопки, а переключение внешнего вида – осуществляться при помощи сдвига `background-position`.

Для автоматизированной сборки спрайтов используются специальные инструменты, например SmartSprites .

Правила форматирования CSS

Для того, чтобы CSS легко читался, полезно соблюдать пять правил форматирования.

Каждое свойство – на отдельной строке

Так – неверно:

```
/*+ no-beautify */
#snapshot-box h2 { padding: 0 0 6px 0; font-weight: bold; position: absolute; left:
```

Так – правильно:

```
/*+ no-beautify */
#snapshot-box h2 {
  position: absolute;
  left: 0;
  top: 0;
  padding: 0 0 6px 0;
  font-weight: bold;
}
```

Цель – лучшая читаемость, проще найти и поправить свойство.

Каждый селектор – на отдельной строке

Неправильно:

```
/*+ no-beautify */
#snapshot-box h2, #profile-box h2, #order-box h2 {
  padding: 0 0 6px 0;
  font-weight: bold;
}
```

Правильно:

```
/*+ no-beautify */
#snapshot-box h2,
#profile-box h2,
#order-box h2 {
  padding: 0 0 6px 0;
  font-weight: bold;
}
```

Цель – лучшая читаемость, проще найти селектор.

Свойства, сильнее влияющие на документ, идут первыми

Рекомендуется располагать свойства в следующем порядке:

1. Сначала положение элемента относительно других: `position`, `left/right/top/bottom`, `float`, `clear`, `z-index`.
2. Затем размеры и отступы: `width`, `height`, `margin`, `padding`...
3. Рамка `border`, она частично относится к размерам.
4. Общее оформление содержимого: `list-style-type`, `overflow`...
5. Цветовое и стилевое оформление: `background`, `color`, `font`...

Общая логика сортировки: «от общего – к локальному и менее важному».

При таком порядке свойства, определяющие структуру документа, будут видны наиболее отчётливо, в начале, а самые незначительно влияющие (например цвет) – в конце.

Например:

```
/*+ no-beautify */
#snapshot-box h2 {
  position: absolute; /* позиционирование */
  left: 0;
  top: 0;

  padding: 0 0 6px 0; /* размеры и отступы */

  color: red;         /* стилевое оформление */
  font-weight: bold;
}
```

Свойство без префикса пишется последним.

Например:

```
-webkit-box-shadow:0 0 100px 20px #000;
box-shadow:0 0 100px 20px #000;
```

Это нужно, чтобы стандартная (окончательная) реализация всегда была важнее, чем временные браузерные.

Организация CSS-файлов проекта

Стили можно разделить на две основные группы:

1. **Блоки-компоненты имеют свой CSS.** Например, CSS для диалогового окна, CSS для профиля посетителя, CSS для меню.

   Такой CSS идёт «в комплекте» с модулем, его удобно выделять в отдельные файлы и подключать через `@import`.

   Конечно, при разработке будет много CSS-файлов, но при выкладке проекта система сборки и сжатия CSS заменит директивы `@import` на их содержимое, объединяя все CSS в один файл.

2. **Страничный и интегрирующий CSS**.

   Этот CSS описывает общий вид страницы, расположение компонент и их дополнительную стилизацию, зависящую от места на странице и т.п.

```
/*+ no-beautify */
.tab .profile { /* профиль внутри вкладки */
  float: left;
  width: 300px;
  height: 200px;
}
```

   Важные страничные блоки можно выделять особыми комментариями:

```
/** ============================
 *  Профиль посетителя
 *  ============================
*/

.profile {
  border: 1px solid gray;
}

.profile h2 {
  color: blue;
  font-size: 1.8em;
}
```

CSS-препроцессоры, такие как SASS , LESS , Stylus , Autoprefixer делают написание CSS сильно удобнее…

Выберите один из них и используйте. Единственно, они добавляют дополнительную предобработку CSS, которую нужно учесть, и желательно, на сервере.

Решения

ArrayBuffer, бинарные массивы

Соедините типизированные массивы

Открыть решение с тестами в песочнице.

К условию

Fetch

Получите данные о пользователях GitHub

Чтобы получить сведения о пользователе, нам нужно вызвать `fetch('https://api.github.com/users/USERNAME')`.

Если ответ приходит со статусом `200`, то вызываем метод `.json()`, чтобы прочитать JS-объект.

А если запрос завершается ошибкой или код статуса в ответе отличен от 200, то мы просто возвращаем `null` в массиве результатов.

Вот код:

```
async function getUsers(names) {
  let jobs = [];

  for(let name of names) {
    let job = fetch(`https://api.github.com/users/${name}`).then(
      successResponse => {
        if (successResponse.status != 200) {
          return null;
        } else {
          return successResponse.json();
        }
      },
      failResponse => {
        return null;
      }
    );
    jobs.push(job);
  }

  let results = await Promise.all(jobs);

  return results;
}
```

Пожалуйста, обратите внимание: вызов `.then` прикреплён к `fetch`, чтобы, когда ответ получен, сразу начинать считывание данных с помощью `.json()`, не дожидаясь завершения других запросов.

Если бы мы использовали `await Promise.all(names.map(name => fetch(...)))` и вызывали бы `.json()` на результатах запросов, то пришлось бы ждать, пока завершатся все из них. Вызывая `.json()` сразу после каждого `fetch`, мы добились того, что считывание присланных по каждому запросу данных происходит независимо от других запросов.

Это пример того, как относительно низкоуровневое Promise API может быть полезным, даже если мы в основном используем `async/await` в коде.

Открыть решение с тестами в песочнице.

К условию

Fetch: запросы на другие сайты

Почему нам нужен Origin?

Нам нужен `Origin`, потому что иногда `Referer` отсутствует. Например, когда мы запрашиваем через `fetch` HTTP-страницу с HTTPS (менее безопасный доступ с более безопасного), то `Referer` нет.

Content Security Policy (политика безопасности содержимого) может запретить отправление `Referer`.

Как мы увидим позже, у `fetch` есть опции, которые предотвращают отправку `Referer` и даже позволяют изменять его (в пределах того же сайта).

Согласно спецификации, `Referer` является необязательным HTTP-заголовком.

Именно потому что `Referer` ненадёжен, был изобретён `Origin`. Браузер гарантирует наличие правильного `Origin` при запросах на другой источник.

К условию

LocalStorage, sessionStorage

Автосохранение поля формы

Открыть решение в песочнице.

К условию

CSS-анимации

Анимировать самолёт (CSS)

CSS для анимации двух свойств `width` и `height`:

```
/* original class */

#flyjet {
  transition: all 3s;
}

/* JS adds .growing */
#flyjet.growing {
  width: 400px;
  height: 240px;
}
```

При разработке следует учитывать, что событие `transitionend` сработает два раза – для каждого свойства (высота и ширина). Таким образом, если не предусмотреть дополнительную проверку, тогда сообщение появится два раза.

Открыть решение в песочнице.

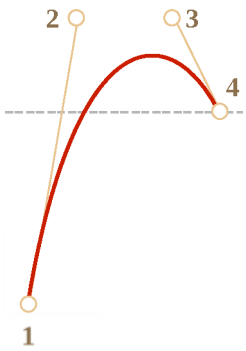
К условию

Анимировать самолёт с перелётом (CSS)

Для такой анимации необходимо подобрать правильную кривую Безье. Для того чтобы самолёт «выпрыгнул», она должна иметь `y>1` на одном из участков.

Например, мы можем указать `y>1` для обеих контрольных точек: `cubic-bezier(0.25, 1.5, 0.75, 1.5)`.

График кривой Безье:



Открыть решение в песочнице.

К условию

Анимированный круг

Открыть решение в песочнице.

К условию

JavaScript-анимации

Анимируйте прыгающий мячик

Чтобы заставить мячик прыгать, можно использовать CSS-свойство `top` и задать мячику `position:absolute` внутри поля с `position:relative`.

Нижняя координата поля – `field.clientHeight`. CSS-свойство `top` относится к верхней границе мяча, которая должна идти от 0 до `field.clientHeight - ball.clientHeight`.

А чтобы получить эффект «скачущего» мяча, мы можем использовать функцию расчёта времени `bounce` в режиме `easeOut`.

Вот конечный код для анимации:

```
let to = field.clientHeight - ball.clientHeight;

animate({
  duration: 2000,
  timing: makeEaseOut(bounce),
  draw(progress) {
    ball.style.top = to * progress + 'px'
  }
});
```

Открыть решение в песочнице.

К условию

Анимируйте мячик, прыгающий вправо

В задаче Анимируйте прыгающий мячик нам надо было анимировать только одно свойство. Теперь необходимо добавить ещё одно: `elem.style.left`.

Горизонтальная координата меняется по другому закону: она не «подпрыгивает», а постепенно увеличивается, сдвигая шар вправо.

Для этого мы можем написать ещё одну функцию `animate`.

В качестве временной функции можно использовать `linear`, но `makeEaseOut(quad)` будет выглядеть гораздо лучше.

Код:

```
let height = field.clientHeight - ball.clientHeight;
let width = 100;

// анимация top (прыжки)
animate({
  duration: 2000,
  timing: makeEaseOut(bounce),
  draw: function(progress) {
    ball.style.top = height * progress + 'px'
  }
});

// анимация left (движение вправо)
animate({
  duration: 2000,
```

```
  timing: makeEaseOut(quad),
  draw: function(progress) {
    ball.style.left = width * progress + "px"
  }
});
```

Открыть решение в песочнице.

К условию

Пользовательские элементы (Custom Elements)

Элемент "живой таймер"

Пожалуйста, обратите внимание:

1. Мы останавливаем таймер `setInterval`, когда элемент удаляется из документа. Это важно, иначе он продолжит тикать, даже если больше не нужен. И браузер не сможет очистить память от этого элемента.
2. Мы можем получить доступ к текущей дате через свойство `elem.date`. Все методы и свойства класса, естественно, являются методами и свойствами элемента.

Открыть решение в песочнице.

К условию

Якоря: начало строки ^ и конец $

Регулярное выражение ^$

Единственной подходящей строкой будет пустая: она начинается и в тот же момент заканчивается.

Это задание ещё раз показывает, что якоря являются не символами, а проверками.

Строка `""` – пустая. Движок пытается найти совпадение с `^` (начало ввода) – да, оно на месте, и далее ищет совпадение с `$` – оно тоже на

месте. То есть сразу найдено полное совпадение.

К условию

Граница слова: \b

Найдите время

Ответ: `\b\d\d:\d\d\b`.

```
alert( "Завтрак в 09:00 в комнате 123:456.".match( /\b\d\d:\d\d\b/ ) ); // 0
```

К условию

Наборы и диапазоны [...]

Java[^script]

Ответы: **нет**, **да**.

- Нет, т.к. в строке `Java` нет каких-либо совпадений, потому что `[^script]` означает «любой символ, кроме заданных». Таким образом, регулярное выражение ищет `"Java"`, за которым следует один такой символ, но после конца строки нет символов.

  ```
  alert( "Java".match(/Java[^script]/) ); // null
  ```

- Да, потому что регулярное выражение регистрозависимое – `[^script]` совпадает с символом `"S"`.

  ```
  alert( "JavaScript".match(/Java[^script]/) ); // "JavaS"
  ```

К условию

Найдите время как hh:mm или hh-mm

Ответ: `\d\d[-:]\d\d`.

```
let regexp = /\d\d[-:]\d\d/g;
alert( "Завтрак в 09:00. Ужин в 21-30".match(regexp) ); // 09:00, 21-30
```

Обратите внимание, что дефис `'-'` имеет специальное значение в квадратных скобках, но только между другими символами, а не в начале или в конце, поэтому нам не нужно экранировать его.

К условию

Квантификаторы +, *, ? и {n}

Как найти многоточие "..." ?

Решение:

```
let regexp = /\.{3,}/g;
alert( "Привет!... Как дела?.....".match(regexp) ); // ..., .....
```

Обратите внимание, что точка – это специальный символ. Мы должны экранировать её, то есть вставлять как `\.`.

К условию

Регулярное выражение для HTML-цветов

Нам нужно найти символ `#`, за которым следуют 6 шестнадцатеричных символов.

Шестнадцатеричный символ может быть описан с помощью регулярного выражения как `[0-9a-fA-F]`. Или же как `[0-9a-f]`, если мы используем модификатор `i`.

Затем мы можем добавить квантификатор `{6}`, так как нам нужно 6 таких символов.

В результате наше регулярное выражение получилось таким: `/#[a-f0-9]{6}/gi`.

```
let regexp = /#[a-f0-9]{6}/gi;

let str = "color:#121212; background-color:#AA00ef bad-colors:f#fddee #fd2"

alert( str.match(regexp) );  // #121212,#AA00ef
```

Проблема в том, что находятся также совпадения, принадлежащие более длинным последовательностям символов:

```
alert( "#12345678".match( /#[a-f0-9]{6}/gi ) ) // #12345678
```

Чтобы исправить это, мы можем добавить в конец нашего регулярного выражения `\b`:

```
// цвет
alert( "#123456".match( /#[a-f0-9]{6}\b/gi ) ); // #123456

// не цвет
alert( "#12345678".match( /#[a-f0-9]{6}\b/gi ) ); // null
```

К условию

Жадные и ленивые квантификаторы

Совпадение для /d+? d+?/

Результат будет: `123 4`.

Первый, ленивый шаблон, `\d+?` попытается получить как можно меньше цифр до первого пробела, поэтому совпадением будет `123`.

Тогда второй `\d+?` возьмёт только одну цифру, потому что этого будет достаточно.

К условию

Поиск HTML-комментариев

Нам нужно найти начало комментария `<!--`. После этого, весь текст до конца комментария `-->`.

Подходящий вариант: `<!--.*?-->` – ленивый квантификатор остановит точку прямо перед `-->`. Но нужно не забыть поставить флаг `s`, чтобы точка включала в себя перевод строки.

Иначе многострочные комментарии не будут найдены:

```
let regexp = /<!--.*?-->/gs;

let str = `... <!-- My -- comment
 test --> ..  <!----> ..
`;

alert( str.match(regexp) ); // '<!-- My -- comment \n test -->', '<!---->'
```

К условию

Поиск HTML-тегов

Решением будет `<[^<>]+>`.

```
let regexp = /<[^<>]+>/g;

let str = '<> <a href="/"> <input type="radio" checked> <b>';

alert( str.match(regexp) ); // '<a href="/">', '<input type="radio" checked>
```

К условию

Скобочные группы

Найти цвет в формате #abc или #abcdef

Регулярное выражение для поиска номера цвета из трёх символов `#abc`: `/#[a-f0-9]{3}/i`.

Нам нужно найти ещё ровно 3 дополнительных шестнадцатеричных цифры. Ни больше ни меньше – в цвете либо 3, либо 6 цифр.

Используем для этого квантификатор `{1,2}`, получится `/#([a-f0-9]{3}){1,2}/i`.

Здесь шаблон `[a-f0-9]{3}` заключён в скобки для корректного применения к нему квантификатора `{1,2}`.

В действии:

```
let regexp = /#([a-f0-9]{3}){1,2}/gi;

let str = "color: #3f3; background-color: #AA00ef; and: #abcd";

alert( str.match(regexp) ); // #3f3 #AA00ef #abc
```

Здесь есть небольшая проблема: шаблон находит `#abc` в `#abcd`. Чтобы предотвратить это, мы можем добавить `\b` в конец:

```
let regexp = /#([a-f0-9]{3}){1,2}\b/gi;

let str = "color: #3f3; background-color: #AA00ef; and: #abcd";

alert( str.match(regexp) ); // #3f3 #AA00ef
```

К условию

Найти все числа

Положительное число с необязательным присутствием десятичной части (из прошлой задачи): `\d+(\.\d+)?`.

Добавим необязательный минус `-?` в начало:

```
let regexp = /-?\d+(\.\d+)?/g;

let str = "-1.5 0 2 -123.4.";

alert( str.match(regexp) );   // -1.5, 0, 2, -123.4
```

К условию

Разобрать выражение

Регулярное выражение для числа: `-?\d+(\.\d+)?`. Мы создали его в предыдущих задачах.

Регулярное выражение для оператора `[-+*/]`. Дефис `-` стоит в начале квадратных скобок, потому что в середине этот символ будет означать диапазон, а нам нужен просто символ `-`.

Отметим, что косая черта должна быть экранирована внутри регулярного выражения JavaScript `/.../`.

Нам необходимо число, оператор и, затем, другие числа. Между ними могут быть необязательные пробелы.

Полное выражение: `-?\d+(\.\d+)?\s*[-+*/]\s*-?\d+(\.\d+)?`.

Оно состоит из трёх частей, между которыми стоит `\s*`:

1. `-?\d+(\.\d+)?` – первое число,
2. `[-+*/]` – оператор,
3. `-?\d+(\.\d+)?` – второе число.

Для получения этих частей в виде отдельных элементов массива-результата давайте вставим скобки вокруг каждой из них, получится `(-?\d+(\.\d+)?)\s*([-+*/])\s*(-?\d+(\.\d+)?)`.

В действии:

```
let regexp = /(-?\d+(\.\d+)?)\s*([-+*\/])\s*(-?\d+(\.\d+)?)/;

alert( "1.2 + 12".match(regexp) );
```

Результат `result` включает в себя:

- `result[0] == "1.2 + 12"` (полное совпадение)
- `result[1] == "1.2"` (первая группа `(-?\d+(\.\d+)?)` – первое число, включая десятичную часть)
- `result[2] == ".2"` (вторая группа `(\.\d+)?` – первая десятичная часть)
- `result[3] == "+"` (третья группа `([-+*\/])` – оператор)
- `result[4] == "12"` (чертвертая группа `(-?\d+(\.\d+)?)` – второе число)
- `result[5] == undefined` (пятая группа `(\.\d+)?` – вторая десятичная часть отсутствует, поэтому значение `undefined`)

Нам необходимы только числа и оператор без полного совпадения или десятичной части, поэтому давайте «почистим» этот результат.

Первый элемент массива (полное совпадение) можно удалить при помощи сдвига массива `result.shift()`.

Группы, которые содержат десятичную часть (номер 2 и 4) `(.\d+)` можно убрать из массива, добавив `?:` в начало: `(?:\.\d+)?`.

Итоговое решение:

```
function parse(expr) {
  let regexp = /(-?\d+(?:\.\d+)?)\s*([-+*\/])\s*(-?\d+(?:\.\d+)?)/;

  let result = expr.match(regexp);

  if (!result) return [];
  result.shift();

  return result;
}

alert( parse("-1.23 * 3.45") );  // -1.23, *, 3.45
```

К условию

Проверьте MAC-адрес

Двузначное шестнадцатеричное число – это `[0-9a-f]{2}` (предполагается, что флаг `i` стоит).

Нам нужно число `NN`, после которого `:NN` повторяется ещё 5 раз.

Регулярное выражение: `[0-9a-f]{2}(:[0-9a-f]{2}){5}`

Теперь давайте покажем, что шаблон должен захватить весь текст (всю строку): от начала и до конца. Для этого обернём шаблон в `^...$`.

Итог:

```
let regexp = /^[0-9a-fA-F]{2}(:[0-9a-fA-F]{2}){5}$/i;

alert( regexp.test('01:32:54:67:89:AB') ); // true

alert( regexp.test('0132546789AB') ); // false (нет двоеточий)

alert( regexp.test('01:32:54:67:89') ); // false (5 чисел, должно быть 6)

alert( regexp.test('01:32:54:67:89:ZZ') ) // false (ZZ в конце строки)
```

К условию

Альтернация (или) |

Найдите языки программирования

Первая идея, которая может прийти в голову – перечислить языки, разделив их `|`.

Но это не сработает, как надо:

```
let regexp = /Java|JavaScript|PHP|C|C\+\+/g;

let str = "Java, JavaScript, PHP, C, C++";

alert( str.match(regexp) ); // Java,Java,PHP,C,C
```

Движок регулярных выражений ищет альтернации в порядке их перечисления. То есть, он сначала смотрит, есть ли `Java`, а если нет – ищет `JavaScript` и так далее.

В результате `JavaScript` не будет найден никогда, только потому что `Java` проверяется первым.

То же самое – с языками `C` и `C++`.

Есть два решения проблемы:

1. Поменять порядок, чтобы более длинное совпадение проверялось первым: `JavaScript|Java|C\+\+|C|PHP`.
2. Соединить одинаково начинающиеся варианты: `Java(Script)?|C(\+\+)?|PHP`.

В действии:

```
let regexp = /Java(Script)?|C(\+\+)?|PHP/g;

let str = "Java, JavaScript, PHP, C, C++";

alert( str.match(regexp) ); // Java,JavaScript,PHP,C,C++
```

К условию

Найдите пары BB-кодов

Открывающий тег – это `\[(b|url|quote)\]`.

Затем, чтобы найти всё до закрывающего тега – используем выражение `.*?` с флагом `s`: оно найдёт любые символы, включая новую строку, и затем добавим обратную ссылку на открывающий тег.

Полное выражение: `\[(b|url|quote)\].*?\[/\1\]`.

В действии:

```
let regexp = /[(b|url|quote)].*?[/\1]/gs;

let str = `
  [b]привет![/b]
  [quote]
    [url]http://ya.ru[/url]
  [/quote]
`;

alert( str.match(regexp) ); // [b]привет![/b],[quote][url]http://ya.ru[/url]
```

Обратите внимание, что необходимо экранировать слеш `[/\1]`, потому что обычно слеш завершает паттерн.

К условию

Найдите строки в кавычках

Решение: `/"(\\.|[^"\\])*"/g`.

Шаг за шагом:

- Сначала ищем открывающую кавычку `"`
- Затем, если есть обратный слеш `\\` (удвоение обратного слеша – техническое, потому что это спец.символ, на самом деле там один обратный слеш), то после него также подойдёт любой символ (точка).
- Иначе берём любой символ, кроме кавычек (которые будут означать конец строки) и обратного слеша (чтобы предотвратить одинокие обратные слеши, сам по себе единственный обратный слеш не нужен, он должен экранировать какой-то символ) `[^"\\]`
- …И так далее, до закрывающей кавычки.

В действии:

```
let regexp = /"(\\.|[^"\\])*"/g;
let str = ' .. "test me" .. "Скажи \\"Привет\\"!" .. "\\\\ \\"" .. ';

alert( str.match(regexp) ); // "test me","Скажи \"Привет\"!","\\ \""
```

К условию

Найдите весь тег

Начало шаблона очевидно: `<style`.

…А вот дальше… Мы не можем написать просто `<style.*?>`, потому что `<styler>` удовлетворяет этому выражению.

После `<style` должен быть либо пробел, после которого может быть что-то ещё, либо закрытие тега `>`.

На языке регулярных выражений: `<style(>|\s.*?>)`.

В действии:

```
let regexp = /<style(>|\s.*?>)/g;

alert( '<style> <styler> <style test="...">'.match(regexp) ); // <style>, <s
```

К условию

Опережающие и ретроспективные проверки

Найдите неотрицательные целые

Регэксп для целого числа: `\d+`.

Мы можем исключить отрицательные добавлением негативной ретроспективной проверки: `(?<!-)\d+`.

Однако, если попробуем применить такой регэксп, то увидим лишний результат:

```
let regexp = /(?<!-)\d+/g;

let str = "0 12 -5 123 -18";

console.log( str.match(regexp) ); // 0, 12, 123, 8
```

Как видите, оно находит `8` из `-18`. То есть, берёт только цифру из числа `-18`, так как это формально подходит под регулярное выражение.

Чтобы исключить такой вариант, надо убедиться, что регэксп не будет искать число с середины другого (неподходящего) числа.

Мы можем сделать это добавлением ещё одной проверки: `(?<!-)(?<!\d)\d+`. Теперь `(?<!\d)` гарантирует, что поиск не начнётся после цифры.

Можем объединить проверки в одну:

```
let regexp = /(?<![-\d])\d+/g;

let str = "0 12 -5 123 -18";

alert( str.match(regexp) ); // 0, 12, 123
```

К условию

Вставьте после фрагмента

Для того, чтобы вставить после тега `<body>`, нужно вначале его найти. Будем использовать регулярное выражение `<body.*>`.

Далее, нам нужно оставить сам тег `<body>` на месте и добавить текст после него.

Это можно сделать вот так:

```
let str = '...<body style="...">...';
str = str.replace(/<body.*>/, '$&<h1>Hello</h1>');

alert(str); // ...<body style="..."><h1>Hello</h1>...
```

В строке замены `$&` означает само совпадение, то есть мы заменяем `<body.*>` заменяется на самого себя плюс `<h1>Hello</h1>`.

Альтернативный вариант – использовать ретроспективную проверку:

```
let str = '...<body style="...">...';
str = str.replace(/(?<=<body.*>)/, `<h1>Hello</h1>`);

alert(str); // ...<body style="..."><h1>Hello</h1>...
```

Такое регулярное выражение на каждой позиции будет проверять, не идёт ли прямо перед ней `<body.*>`. Если да – совпадение найдено. Но сам тег `<body.*>` в совпадение не входит, он только участвует в проверке. А других символов после проверки в нём нет, так что текст совпадения будет пустым.

Происходит замена «пустой строки», перед которой идёт `<body.*>` на `<h1>Hello</h1>`. Что, как раз, и есть вставка этой строки после `<body>`.

P.S. Этому регулярному выражению не помешают флаги: `/<body.*>/si`, чтобы в «точку» входил перевод строки (тег может занимать несколько строк), а также чтобы теги в другом регистре типа `<BODY>` тоже находились.

К условию

Свойство float

Разница inline-block и float

Разница колоссальная.

В первую очередь она в том, что `inline-block` продолжают участвовать в потоке, а `float` – нет.

Чтобы её ощутить, достаточно задать себе следующие вопросы:

1. Что произойдёт, если контейнеру `UL` поставить рамку `border` – в первом и во втором случае?
2. Что будет, если элементы `LI` различаются по размеру? Будут ли они корректно перенесены на новую строку в обоих случаях?
3. Как будут вести себя блоки, находящиеся под галереей?

Попробуйте сами на них ответить.

Затем читайте дальше.

Что будет, если контейнеру `UL` поставить рамку `border`?

Контейнер не выделяет пространство под `float`. А больше там ничего нет. В результате он просто сожмётся в одну линию сверху.

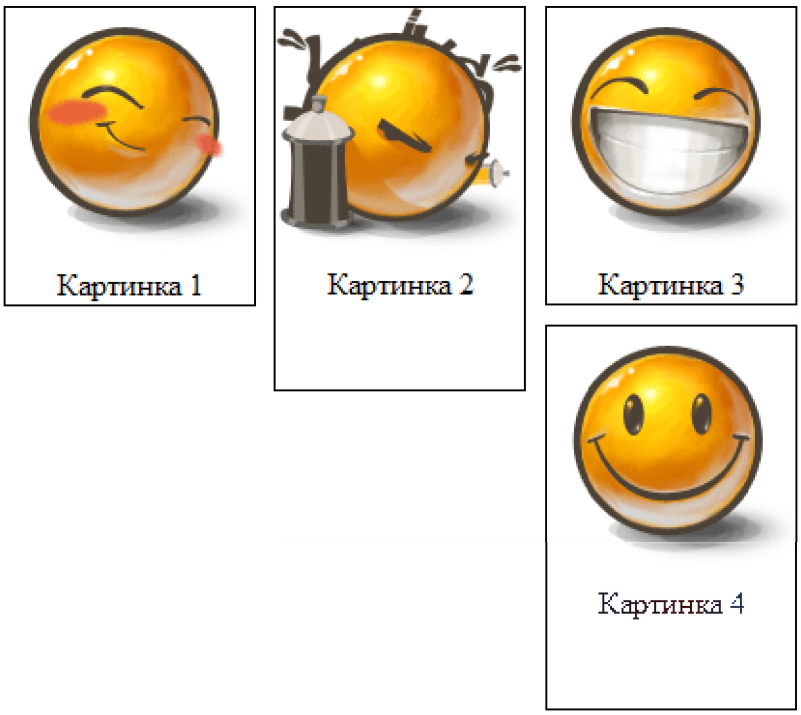
Попробуйте сами, добавьте рамку в песочнице .

А в случае с `inline-block` всё будет хорошо, т.к. элементы остаются в потоке.

Что будет, если элементы `LI` различаются по размеру? Будут ли они корректно перенесены на новую строку в обоих случаях?

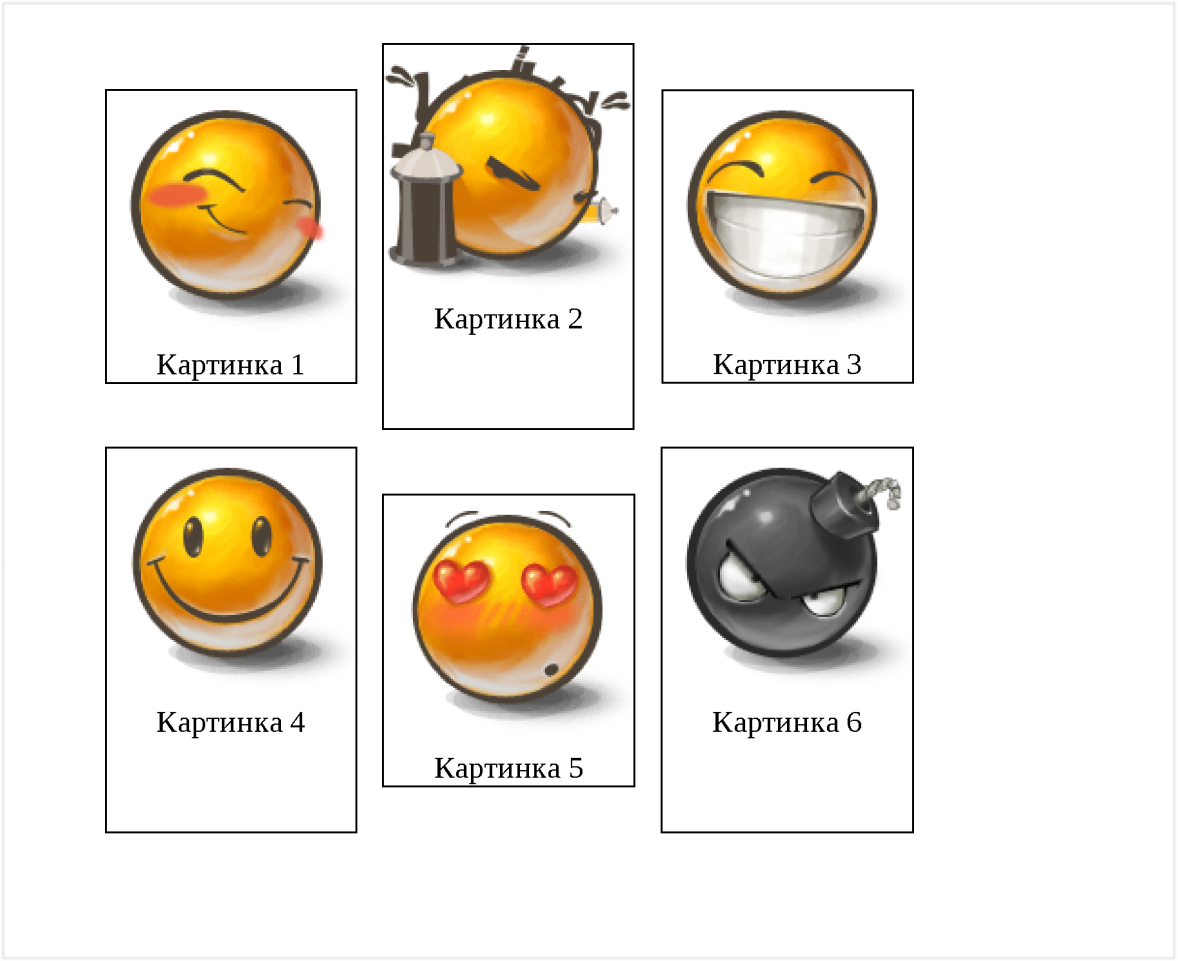
При `float:left` элементы двигаются направо до тех пор, пока не наткнутся на границу внешнего блока (с учётом `padding`) или на другой `float`-элемент.

Может получиться вот так:



Вы можете увидеть это, открыв демо-галерею в отдельном окне и изменяя его размер:

При использовании `inline-block` таких странностей не будет, блоки перенесутся корректно на новую строку. И, кроме того, можно выровнять элементы по высоте при помощи `li { vertical-align:middle }`:



Как будут вести себя блоки, находящиеся под галереей?

В случае с `float` нужно добавить дополнительную очистку с `clear`, чтобы поведение было идентично обычному блоку.

Иначе блоки, находящиеся под галереей, вполне могут «заехать» по вертикали на территорию галереи.

Открыть решение в песочнице.

К условию

Дерево с многострочными узлами

Для решения можно применить принцип двухколоночной вёрстки `float` + `margin`. Иконка будет левой колонкой, а содержимое – правой.

Открыть решение в песочнице.

К условию

Постраничная навигация (CSS)

HTML-структура:

```
<div class="nav">
  <img src="arrow-left.jpg" class="left" width="40" height="40">
  <img src="arrow-right.jpg" class="right" width="40" height="40">
  <ul class="pages">
    <li>...</li>
  </ul>
</div>
```

Стили:

```
.nav {
  height: 40px;
  width: 80%;
  margin: auto;
}

.nav .left {
  float: left;
  cursor: pointer;
}

.nav .right {
  float: right;
  cursor: pointer;
}

.nav .pages {
  list-style: none;
  text-align: center;
  margin: 0;
  padding: 0;
}

.nav .pages li {
  display: inline;
  margin: 0 3px;
  line-height: 40px;
  cursor: pointer;
}
```

Основные моменты:

- **Сначала идёт левая кнопка, затем правая, а лишь затем – текст.** Почему так, а не лево – центр – право?

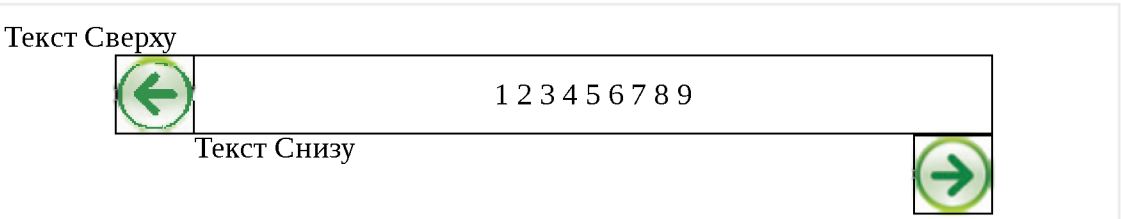
Дело в том, что `float` смещает элемент вправо относительно обычного места. А какое обычное место будет у правого `IMG` без `float`?

Оно будет под списком, так как список – блочный элемент, а `IMG` – инлайн-элемент. При добавлении `float:right` элемент `IMG` сдвинется вправо, оставшись под списком.

Код в порядке лево-центр-право (неправильный):

```
<div...>
  <img src="arrow-left.jpg" class="left" width="40" height="40">
  <ul class="pages"> (li) 1 2 3 4 5 6 7 8 9</ul>
  <img src="arrow-right.jpg" class="right" width="40" height="40">
</div>
```

Его демо:



Правильный порядок: лево-право-центр, тогда `float` останется на верхней строке.

Код, который даёт правильное отображение:

```
<div ...>
  <img src="arrow-left.jpg" class="left" width="40" height="40">
  <img src="arrow-right.jpg" class="right" width="40" height="40">
  <ul class="pages"> .. список .. </ul>
</div>
```

Также можно расположить стрелки при помощи `position: absolute`. Тогда, чтобы текст при уменьшении размеров окна не налез на стрелки – нужно добавить в контейнер левый и правый `padding`:

Выглядеть будет примерно так:

```
<div style="position:relative; padding: 0 40px;">
  <img style="position:absolute;left:0" src="..left.." width="40" height=
```

```
  <ul> (li) 1 2 3 4 5 6 7 8 9 </ul>
  <img style="position:absolute;right:0" srr="..right.." width="40" heigh
</div>
```

- **Центрирование одной строки по вертикали осуществляется указанием `line-height`, равной высоте.**

  Это красиво лишь для одной строки: если окно становится слишком узким, и строка вдруг разбивается на две – получается некрасиво, хотя и читаемо.

  Если хочется сделать красивее для двух строк, то можно использовать другой способ центрирования.

Открыть решение в песочнице.

К условию

Добавить рамку, сохранив ширину

Подсказка

Используйте свойство `box-sizing`.

Решение

Да, можно – указываем `box-sizing: border-box` и добавляем свойства:

```
<style>
  .left {
    float:left;
    width:30%;
    background: #aef;
  }

  .right {
    float:right;
    width:70%;

    box-sizing: border-box;
    -moz-box-sizing: border-box;

    border-left: 2px solid green;
    padding-left: 10px;

    background: tan;
```

```
  }
</style>

<div class="left">
 Левая<br>Колонка
</div>
<div class="right">
 Правая<br>Колонка<br>...
</div>
```

К условию

Свойство position

Модальное окно

Если использовать `position: absolute`, то `DIV` не растянется на всю высоту документа, т.к. координаты вычисляются *относительно окна*.

Можно, конечно, узнать эту высоту при помощи JavaScript, но CSS даёт более удобный способ. Будем использовать `position:fixed`:

Стиль:

```
#box {
  position: fixed;
  left: 0;
  top: 0;
  width: 100%;
  height: 100%;
  z-index: 999;
}
```

Свойство `z-index` должно превосходить все другие элементы управления, чтобы они перекрывались.

Открыть решение в песочнице.

К условию

Центрирование горизонтальное и вертикальное

Поместите мяч в центр поля (CSS)

Сместим мяч в центр при помощи `left/top=50%`, а затем приподымем его указанием `margin`:

```
#ball {
  position: absolute;
  left: 50%;
  top: 50%;
  margin-left: -20px;
  margin-top: -20px;
}
```

Открыть решение в песочнице.

К условию

Форма + модальное окно

Структура решения

Шаги решения:

1. Для того, чтобы элементы окна не работали, их нужно перекрыть `DIV'ом` с большим `z-index`.
2. Внутри него будет лежать «экран» с полупрозрачностью. Чтобы он растягивался, можно дать ему `position: absolute` и указать все координаты `top/left/right/bottom`. Это работает в IE8+.
3. Форму можно отцентрировать при помощи `margin` или `display: table-cell` + `vertical-align` на внешнем `DIV`.

Решение

Открыть решение в песочнице.

К условию

vertical-align + table-cell + position = ?

Подсказка

Центрирование не работает из-за `position: absolute`.

Решение

Центрирование не работает потому, что `position: absolute` автоматически меняет элементу `display` на `block`.

В однострочном случае можно сделать центрирование при помощи `line-height`:

```
<style>
  .arrow {
    position: absolute;
    height: 60px;
    border: 1px solid black;
    font-size: 28px;

    line-height: 60px;
  }
</style>

<div class="arrow">«</div>
```

Если же центрировать нужно несколько строк или блок, то есть и другие техники центрирования, которые сработают без `display:table-cell`.

К условию

Свойство margin

Нерабочие margin?

Ошибка заключается в том, что `margin` при задании в процентах высчитывается *относительно ширины*. Так написано в стандарте.

При этом не важно, какой отступ: левый, правый. верхний или нижний. Все они в процентах отсчитываются от ширины. Из-за этого и ошибка.

Ситуацию можно исправить, например, заданием `margin-top/margin-bottom` в пикселях, если это возможно или, в качестве альтернативы, использовать другие средства, в частности, `position` или `padding-top/padding-bottom` на родителе.

К условию

Расположить текст внутри INPUT

Подсказка

Надвиньте элемент с текстом на `INPUT` при помощи отрицательного `margin`.

Решение

Надвинем текст на `INPUT` при помощи отрицательного `margin-top`. Поднять следует на одну строку, т.е. на `1.25em`, можно для красоты чуть больше – `1.3em`:

Также нам понадобится обнулить «родной» `margin` у `INPUT`, чтобы не сбивал вычисления.

```
<style>
  input {
    margin: 0;
    width: 12em;
  }

  #placeholder {
    color: red;
    margin: -1.3em 0 0 0.2em;
  }
</style>

<input type="password" id="input">
<div id="placeholder">Скажи пароль, друг</div>
```

Открыть решение в песочнице.

К условию

Знаете ли вы селекторы?

Выберите элементы селектором

```
<!DOCTYPE HTML>
<html>

<head>
```

```
  <meta charset="utf-8">
</head>

<body>

  <input type="checkbox">
  <input type="checkbox" checked>
  <input type="text" id="message">

  <h3 id="widget-title">Сообщения:</h3>
  <ul id="messages">
    <li id="message-1">Сообщение 1</li>
    <li id="message-2">Сообщение 2</li>
    <li id="message-3" data-action="delete">Сообщение 3</li>
    <li id="message-4" data-action="edit do-not-delete">Сообщение 4</li>
    <li id="message-5" data-action="edit delete">Сообщение 5</li>
    <li><a href="#">...</a></li>
  </ul>


  <a href="http://site.com/list.zip">Ссылка на архив</a>
  <a href="http://site.com/list.pdf">..И на PDF</a>


  <script>
    // тестовая функция для селекторов
    // проверяет, чтобы элементов по селектору selector было ровно count
    function test(selector, count) {
      var elems = document.querySelectorAll(selector);
      var ok = (elems.length == count);

      if (!ok) alert(selector + ": " + elems.length + " != " + count);
    }

    // ------------- селекторы --------------

    // Выбрать input типа checkbox
    test('input[type="checkbox"]', 2);

    // Выбрать input типа checkbox, НЕ отмеченный
    test('input[type="checkbox"]:not(:checked)', 1);

    // Найти все элементы с id=message или message-*
    test('[id|="message"]', 6);

    // Найти все элементы с id=message-*
    test('[id^="message-"]', 5);

    // Найти все ссылки с расширением href="...zip"
    test('a[href$=".zip"]', 1);

    // Найти все элементы с data-action, содержащим delete в списке (через п
    test('[data-action~="delete"]', 2);
```

```
    // Найти все элементы, у которых ЕСТЬ атрибут data-action,
    // но он НЕ содержащит delete в списке (через пробел)
    test('[data-action]:not([data-action~="delete"])', 1);

    // Выбрать все чётные элементы списка #messages
    test('#messages li:nth-child(2n)', 3);

    // Выбрать один элемент сразу за заголовком h3#widget-title
    // на том же уровне вложенности
    test('h3#widget-title + *', 1);

    // Выбрать все ссылки, следующие за заголовком h3#widget-title
    // на том же уровне вложенности
    test('h3#widget-title ~ a', 2);

    // Выбрать ссылку внутри последнего элемента списка #messages
    test('#messages li:last-child a', 1);
  </script>
</body>

</html>
```

К условию

Отступ между элементами, размер одна строка

Выбор элементов

Для выбора элементов, начиная с первого, можно использовать селектор nth-child .

Его вид: `li:nth-child(n+2)`, т.к. `n` идёт от нуля, соответственно первым будет второй элемент (`n=0`), что нам и нужно.

Решение

Отступ, размером в одну строку, при `line-height: 1.5` – это `1.5em`.

Правило:

```
li:nth-child(n+2) {
  margin-top: 1.5em;
}
```

Ещё решение

Ещё один вариант селектора: `li + li`

```
li + li {
  margin-top: 1.5em;
}
```

Открыть решение в песочнице.

К условию

Отступ между парами, размером со строку

Селектор

Для отступа между парами, то есть перед каждым нечётным элементом, можно использовать селектор nth-child .

Селектор будет `li:nth-child(odd)`, к нему нужно ещё добавить отсечение первого элемента: `li:nth-child(odd):not(:first-child)`.

Можно поступить и по-другому: `li:nth-child(2n+3)` выберет все элементы для `n=0,1,2...`, то есть 3-й, 5-й и далее, те же, что и предыдущий селектор. Немного менее очевидно, зато короче.

Правило

Отступ, размером в одну строку, при `line-height: 1.5` – это `1.5em`.

Поставим отступ перед каждым *нечётным* элементом, кроме первого:

```
li:nth-child(odd):not(:first-child) {
  margin-top: 1.5em;
}
```

Получится так:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <style>
    body {
      font: 14px/1.5 serif;
    }
```

```
    ul {
      margin: 0;
    }

    li:nth-child(odd):not(:first-child) {
      margin-top: 1.5em;
    }
  </style>
</head>

<body>

  Текст вверху без отступа от списка.
  <ul>
    <li>Маша</li>
    <li>Паша</li>
    <li>Даша</li>
    <li>Женя</li>
    <li>Саша</li>
    <li>Гоша</li>
  </ul>
  Текст внизу без отступа от списка.

</body>

</html>
```

Открыть решение в песочнице.

К условию