



Промисы. Хранилище

Продвинутый JavaScript



Оглавление

Введение	3
Термины, используемые в лекции	3
Промисы (Promise)	3
Использование промисов: then()	5
Методы обработки ошибок	10
Метод catch()	10
Метод finally()	11
Методы для работы с массивом промисов	12
Promise.all()	12
Promise.race() и Promise.any()	13
Promise.allSettled()	14
Куки (Cookie)	15
LocalStorage и SessionStorage	16
Генераторы	20
Подведём итоги	23

Введение

На этом занятии мы подробнее разберём асинхронные запросы и работу с хранилищем браузера.

Ранее вы уже изучили, что в JavaScript есть специальные выражения, работа которых может быть связана с какими-то ожиданиями. И, дабы не задерживать работу нашего приложения, они выполняются асинхронно. Называются они промисами (promise — обещание с англ.). Чаще всего промисами пользуются для запроса данных с сервера, но есть и другие применения.

Хранилищем мы можем воспользоваться для сохранения каких-то данных на будущее или для передачи данных между модулями. В сети очень много сайтов, которые перед использованием спрашивают вас разрешение хранить данные у вас на компьютере. А как именно их использовать, мы узнаем в этой главе.

Термины, используемые в лекции

Асинхронная операция — операция, которая не блокирует текущее выполнение кода и после запуска этой операции продолжает выполнение других задач.

Промис — основной вид асинхронных операций в JavaScript. Мы даём «обещание», что скоро нам придут какие-то нужные данные и мы их выполним, а пока продолжим выполнение текущих, синхронных задач.

Генераторы — это специальный тип функции, который может приостанавливать своё выполнение, принимать промежуточные данные и продолжать своё выполнение с новыми вводными.

Промисы (Promise)

Представим, что вы заказываете что-то в интернет-магазине. Вы заказали, посылка идёт к вам какое-то время, после её получаете и пользуетесь. Во время ожидания доставки вы также можете заказать ещё товары новыми посылками или готовиться к получению тех, что уже заказали — купить или смастерить упаковку, если это подарок для кого-то, расчистить место для установки, если посылка объёмная и так далее. Ожидание посылки вас не блокирует, продавец также может отправить вам или кому-нибудь ещё другие посылки. Эта отправка называется асинхронной.

Ранее вы уже изучили, что такое асинхронные операции в JavaScript. Узнали пару функций запланированной асинхронности и научились запрашивать данные с сервера классическим методом. Но язык развивается, сетевые запросы стали удобнее, и мы рассмотрим их на заключительном уроке этого курса. А сейчас рассмотрим основу, на чём они работают — промисы.

Промис (англ. Promise) — это обещание, что мы сейчас запустим операцию и она выполнится когда-то в будущем. Как в приведённом выше примере — магазин отправил посылку, это как обещание, что мы когда-то получим нужную нам вещь. Когда получим, тогда и сможем ей воспользоваться, а пока можем заняться другими не менее важными делами.

Конечно, сама работа объекта Promise в JavaScript несколько сложнее, но приведённый пример нам позволяет понять общую задумку.

Формат создания экземпляра Promise:

```
1 let promise = new Promise(function(resolve, reject){
2     // функция — исполнитель
3 });
```

Функция, переданная в конструктор промиса — это функция-исполнитель. Она сама по себе является коллбеком и должна будет запуститься, когда промис создастся. В нашем примере это продавец, отправивший посылку. Аргументы коллбека внутри промиса — `resolve` и `reject` — это тоже, в свою очередь, коллбеки и должны будут вызваться при достижении какого-то результата:

- `Resolve` вызывается при успешном завершении, в нашем примере — это когда посылка дошла. При успешном завершении у нас будет результат `value`.
- `Reject` вызывается при возникновении ошибки, у нас — когда посылка потерялась или повредилась. Ошибка вернёт нам результат `error`.

У объекта `promise`, возвращаемого конструктором `new Promise`, есть внутренние свойства:

- `state` («состояние») — сначала «`pending`» («ожидание»), потом меняется на «`fulfilled`» при успешном выполнении (выполнился `resolve`) или на «`rejected`» при возникновении ошибок (выполнился `reject`).

- `result` («результат») — во время ожидания он равен `undefined`, после изменится на `value` при успешном выполнении, либо на `error` при возникновении ошибок.

Попробуем создать промис с помощью функции таймаута:

```
1 let promise = new Promise(function(resolve, reject) {  
2   setTimeout(() =>  
3     Math.random()*2 < 1  
4     ? resolve('Выполнено')  
5     : reject(new Error("Ошибочка")), 500);  
6 });
```

Здесь у нас с помощью сгенерированного случайного числа решается, завершится наш промис ошибкой или успешным выполнением. Действие произойдёт асинхронно, с задержкой в полсекунды.



Промис может вернуть что-то одно либо успех, либо ошибку. После возврата какого-то решения он завершает работу и все остальные `resolve` и `reject` будут проигнорированы.



Колбеки `resolve` или `reject` могут взять не более одного аргумента (и это чаще всего объект). Всё, что будет после первого аргумента, будет проигнорировано.

Использование промисов: `then()`

Создавать новые промисы в своей практике вы будете нечасто. Гораздо важнее научиться пользоваться готовыми промисами. Например, с помощью существующего промиса `fetch()` вы, скорее всего, будете запрашивать данные с подключаемого бэкенд-сервера. Его мы изучим позже на этом курсе.

А сейчас нам нужно научиться пользоваться промисами в целом. И самым частым используемым методом в промисах является метод `then()`. Его синтаксис такой:

```
1 promise.then(onfulfilled, onrejected);
```

Где в качестве параметров `onfulfilled` и `onrejected` почти всегда выступают коллбеки-обработчики. Оба параметра не являются обязательными, но чаще всего `then` используется с одним параметром `onfulfilled`:

```
1 promise.then(onfulfilled);
```

🔥 При выполнении промиса вызовется только один соответствующий параметр. Второй параметр проигнорируется. Если был вызван только один параметр `onfulfilled`, то он выполнится только при успешном выполнении.

`Onfulfilled` выполнится тогда, когда выполнение промиса произошло без ошибок. `Onrejected` выполнится только при ошибке.

Почему в качестве параметров выступают коллбеки? Здесь всё просто. Нам нужно какое-то действие по результатам работы промиса. Синхронно оно нам не будет доступно, поэтому код:

```
1 console.log(promise.then());
```

Не вернёт нам внятного результата:

```
▼ Promise {<pending>} ⓘ  
  ► [[Prototype]]: Promise  
    [[PromiseState]]: "pending"  
    [[PromiseResult]]: undefined
```

Все обработчики должны быть колбеками и их можно помещать прямо в тело метода `then()`. Более того, чаще именно так и делают:

```
1 promise.then( result => {  
2     console.log(result);  
3 });
```

Если переменная в параметре одна, мы можем коллбек вызвать без круглых скобок вначале. Если параметров несколько или они сложные, то круглые скобки обязательны:

```
1 promise.then( (result1, result2) => {  
2     console.log(result1, result2);  
3 });
```

```
1 promise.then( ({result1, result2}) => {  
2     console.log(result1, result2);  
3 });
```

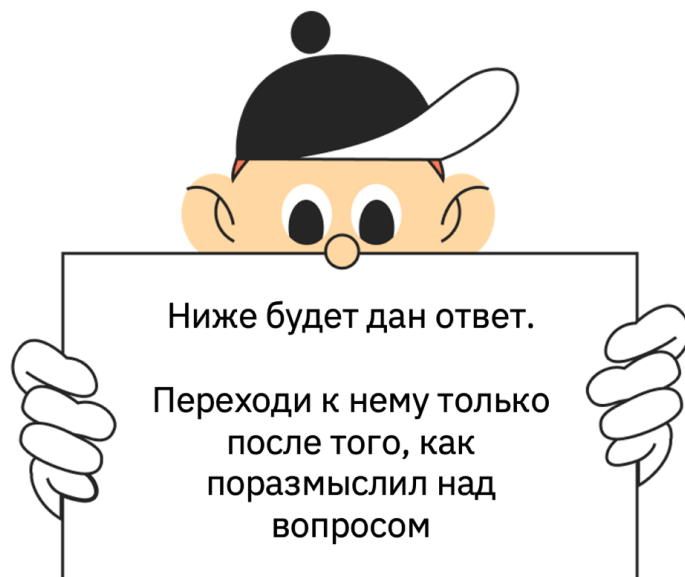
Фигурные скобки во втором примере являются деструктуризацией объекта и чаще всего применяются, когда нам не нужен объект целиком, а только одно или несколько его свойств. Мы также можем деструктурировать и массивы:

```
1 promise.then( ([result1, result2]) => {  
2     console.log(result1, result2);  
3 });
```

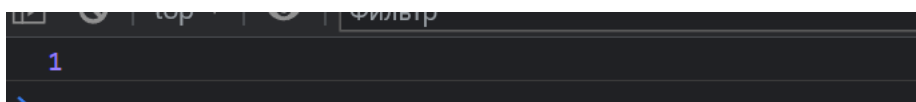
Эти приёмы вы изучали ранее.

А теперь вопрос для размышлений. Что произойдёт в следующем коде?

```
1 new Promise(resolve => resolve(1)).then(console.log);
```



Здесь в качестве параметра метода `then()` передан `console.log`. Он будет обработан как функция-коллбек метода `then()` и ему передадутся данные для обработки при успешном выполнении промиса. А что лучше всего делает метод `log()` класса `console`? Всё правильно, он выведет успешный результат промиса в консоль:



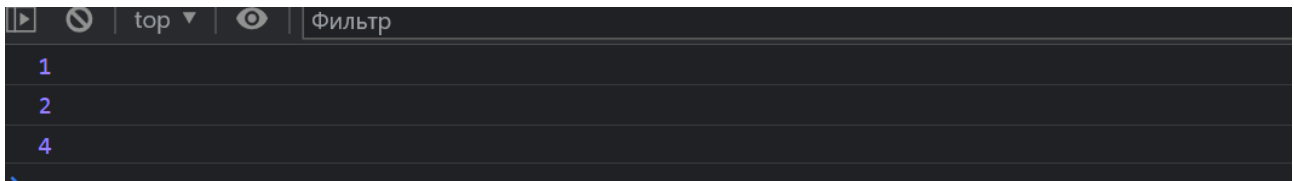
Если мы в коллбеке возвращаем новый промис, например, как в `fetch()`, то мы можем использовать цепочки промисов:

```
1 new Promise(function(resolve) {
2   setTimeout(() => resolve(1), 1000);
3 }).then(function(result) {
4   console.log(result);
5   return new Promise((resolve) => {
6     setTimeout(() => resolve(result * 2), 1000);
7   });
8 }).then(function(result) {
9   console.log(result);
10  return new Promise((resolve) => {
11    setTimeout(() => resolve(result * 2), 1000);
12  });
13 }).then(function(result) {
```



```
14     console.log(result);
15 });
```

Здесь мы три раза с интервалом в секунду получим три ответа. Первый промис через таймаут в 1 секунду выдаст результат 1 с помощью функции `resolve(1)`, второй промис умножит его на 2, третий промис полученный результат из второго промиса умножит на 2. Смотрим в консоль:



The screenshot shows a browser console with three log entries. The first entry is the number 1, the second is 2, and the third is 4. The console interface includes a filter input and a 'top' dropdown menu.

Нам вывелось три числа, как мы и ожидали. На самом деле, нам не нужно каждый раз создавать новый промис, а возвращать объект так называемого класса `Thenable`, в котором мы определим метод `then()`:

```
1 class Thenable {
2     constructor(num) {
3         this.num = num;
4     }
5     then(resolve) {
6         console.log(resolve); // function() { native code }
7         // будет успешно выполнено с аргументом this.num*2 через 1 секунду
8         setTimeout(() => resolve(this.num * 2), 1000);
9     }
10 }
11 new Promise(resolve => resolve(1))
12     .then(result => {
13         return new Thenable(result);
14     })
15     .then(console.log); // показывает 2 через 1 с
```

Методы обработки ошибок

Метод catch()

Для обработки ошибок в промисах используются методы `catch()` и `finally()`. И... где же мы их уже встречали? Правильно. В предыдущем уроке, в конструкции `try...catch...finally`. Более того, работают они схожим образом.

🔥 Напоминаем, что конструкция `try...catch...finally` работает только в синхронном коде и её нет никакого смысла использовать при работе с промисами.

Как вы видели, в предыдущих примерах я почти не использовал второй параметр метода `then()`. Всё потому, что его работа аналогична методу `catch()`. Так для чего же у нас используются две конструкции аналогичного принципа действия? Всё дело в том, что вначале в промисах был только метод `then()` и обрабатывать ошибки можно было только одним способом. Потом решили ввести в новых стандартах обработку ошибок схожую по синтаксису и функционалу с уже имеющейся конструкцией `try...catch...finally`, а старое использование оставили для совместимости. Но никто не запрещает использовать метод `then()` по-старому.

Вызов `catch(onrejected)` аналогичен, вызову `then(null, onrejected)`. Если нам не нужна обработка данных, которые приходят при успешном выполнении промиса, метод `then()` мы вообще можем не вызывать.

В методах `catch` мы можем пробрасывать ошибки для правильной дальнейшей обработки и обрабатывать таким образом несколько типов ошибок. Делается это с помощью уже известного нам оператора `throw`. А если мы обработаем все ошибки в `catch()`, то можем продолжить работу дальше и управление перейдёт следующему методу `then()`:

```
1 new Promise((resolve, reject) => {
2   throw new Error("Ошибка!");
3 }).catch((error) => {
4   console.log(`Ошибка ${error} обработана, продолжаем работу`);
5 }).then(() => console.log("Управление перейдёт в следующий then"));
```

Приведём пример проброса ошибок:

```

1 new Promise((resolve, reject) => {
2     throw new Error("Ошибка!");
3 }).catch((error) => {
4     if (error instanceof URIError) {
5         // обрабатываем ошибку
6     } else {
7         console.log("Не могу обработать ошибку");
8         throw error; // пробрасывает эту или другую ошибку в следующий catch
9     }
10 }).then(() => {
11     /* не выполнится */
12 }).catch(error => {
13     console.log(`Неизвестная ошибка: ${error}`);
14     // ничего не возвращаем => выполнение продолжается в нормальном режиме
15 });

```

Здесь в теле промиса генерируется ошибка, мы её получаем в первом `catch()` и пробуем обработать. Если обработка успешна, то управление передаём последующему `then()`. Если обработка неудачна, то у нас срабатывает оператор `throw` и тогда работает следующий `catch()`, а `then()` перед ним пропускается. В текущем примере заключительный `catch()` обработал ошибку и если после него был ещё `then()`, управление передалось ему как в предыдущем примере.

Метод `finally()`

По аналогии с конструкцией `try...catch...finally`, метод промисов `finally()` выполнится в любом случае, независимо от того, произошла ошибка или нет. Главным отличием метода `finally()` состоит в том, что он не принимает никаких аргументов и не возвращает никаких значений. Он также не знает ни о чём, что происходит в промисе. Если там произошла ошибка или сформировался какой-то результат, `finally()` пропустит его для последующих обработчиков.

Для чего он может быть нужен? `finally()` выполнится тогда, когда завершится ожидание (статус `pending`) у промиса. И нам нужно будет выполнить какие-то действия, не зависящие от его результата. Например, на время получения данных, мы можем навесить лодер (loader) — пиктограмму, просящую пользователя

подождать. Допустим, у нас есть объект лодера `start()`, который его запускает и `stop()`, который его скрывает. Перед выполнением промиса мы его запустим, а потом, когда будет доступен какой-то результат выполнения промиса, мы его скроем с помощью метода `finally()`:

```
1 Loader.start();  
2 new Promise((resolve, reject) => {  
3 })  
4 .finally(() => Loader.stop)  
5 .then(result => {})  
6 .catch(error => {});
```

Методы `then()` и `catch()` примут свои результаты вне зависимости от работы метода `finally()`.

Метод `finally()` был добавлен в стандарте ECMAScript 2018.

Методы для работы с массивом промисов

В классе `Promise` есть ещё методы, которые дают нам новые возможности. Многие из них появились в недавних стандартах, поэтому в старых браузерах могут не работать.

`Promise.all()`

Принимает массив (или любой другой итерируемый объект) промисов и возвращает новый промис. Переданные промисы будут выполнены по порядку последовательно, результатом выполнения будет массив результатов из каждого промиса. Если в каком-то промисе произойдёт ошибка, то результатом выполнения будет эта ошибка, остальные результаты проигнорируются. Выполнение промисов не прервётся, так как сейчас в самом классе промисов нет возможности для отмены запроса и вернут какой-то результат. Но этот результат будет проигнорирован, а управление передаётся в метод `catch()` для всего `Promise.all()`.

```
1 Promise.all([  
2   new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
```

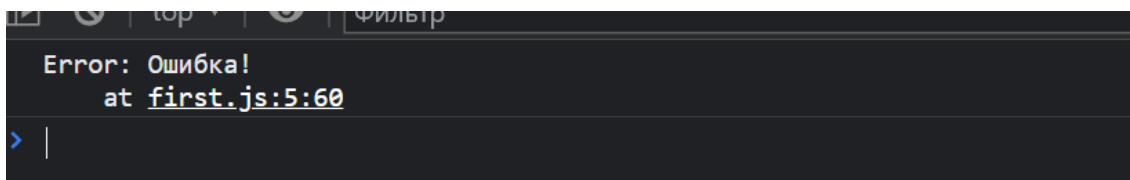
```

3  new Promise((resolve, reject) => setTimeout(() => reject(new
    Error("Ошибка!")), 2000)),
4  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
5 ])

6  .then(console.log)
7  .catch(console.log);

```

Здесь передали массив промисов в `Promise.all()`. Во втором промисе у нас возникла ошибка, поэтому управление передалось в метод `catch()`, а метод `then()` проигнорировался.



```

Error: Ошибка!
    at first.js:5:60
> |

```

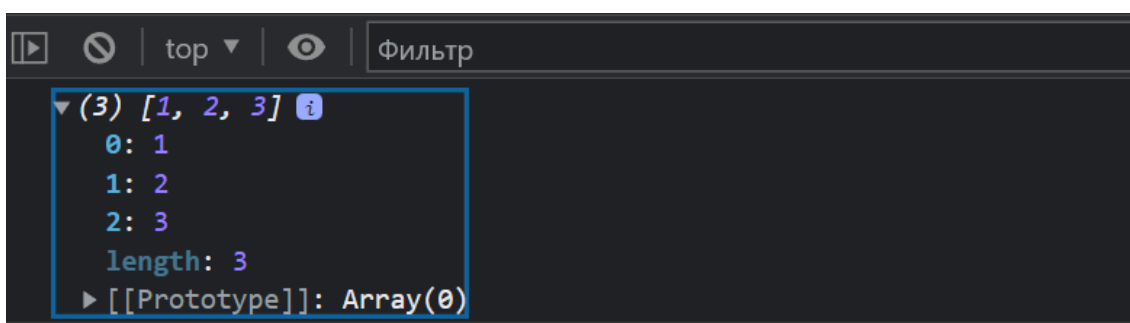
Если у нас будет всё успешно:

```

Promise.all([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(2), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
])
  .then(console.log)
  .catch(console.log);

```

Здесь через 6 секунд вернётся массив из трёх элементов:



```

(3) [1, 2, 3]
  0: 1
  1: 2
  2: 3
  length: 3
  [[Prototype]]: Array(0)

```

Promise.race() и Promise.any()

Два похожих метода, которые, в свою очередь, похожи на `Promise.all()`. В качестве аргумента берут массив промисов, как в `Promise.all()`.

Метод `Promise.race()` берёт из массива промисов первый выполнившийся (по скорости) промис и возвращает его результат. Остальные промисы будут проигнорированы.

В нашем примере:

```
1 Promise.race([
2   new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
3   new Promise((resolve, reject) => setTimeout(() => reject(new
    Error("Ошибка!")), 2000)),
4   new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
5 ])
6 .then(console.log)
7 .catch(console.log);
```

Вернётся результат только первого промиса, так как он быстрее.

`Promise.any` ждёт результат первого **успешно** выполнившегося промиса, остальные игнорируются. Если все промисы будут с ошибками, то вернётся ошибка объекта `AggregateError`. Это специальный объект ошибок промисов, которые хранятся в его свойстве `errors`.

Promise.allSettled()

Этот метод аналогичен `Promise.all`, но исправляет его недостатки. Результатом его выполнения будет массив объектов с итоговым статусом и результатом выполнения.

```
1 Promise.allSettled([
2   new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
3   new Promise((resolve, reject) => setTimeout(() => reject(new
    Error("Ошибка!")), 2000)),
4   new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
5 ])
6 .then(console.log)
7 .catch(console.log);
```

В консоли будет такой результат:

```
▼ (3) [{...}, {...}, {...}] ⓘ
  ▶ 0: {status: 'fulfilled', value: 1}
  ▶ 1: {status: 'rejected', reason: Error: Ошибка! at file:///C:/repo/learn-js/first.js:5:60}
  ▶ 2: {status: 'fulfilled', value: 3}
    length: 3
  ▶ [[Prototype]]: Array(0)
```

Этот метод был введен в стандарте ECMAScript 2020.

Куки (Cookie)

Куки — это строки с данными, которые хранятся в браузере. Куки не входят в стандарт ECMAScript, они являются частью стандарта протокола HTTP по спецификации RFC 6265.


Куки устанавливаются веб-сервером. Устанавливаются они с помощью HTTP-заголовка Set-Cookie, затем браузер их будет вставлять во все запросы с помощью заголовка Cookie.

Наиболее частое применение — это авторизация. Сервер при авторизации устанавливает в куки токен авторизации с помощью заголовка Set-Cookie, затем при следующих запросах, браузер отправляет токен авторизации в заголовке Cookie и сервер верифицирует пользователя по токenu.

Мы можем изменять установленные куки при помощи свойства cookie объекта document, который, в свою очередь, является частью глобального объекта window. Это свойство доступно как для чтения, так и для записи. Имеет формат «ключ=значение;», причём при записи не перетирается полностью, а добавляются или редактируются только указанные ключи. Приведём пример:

```
1 console.log(document.cookie);
2 // → "expires=Tue, 14 Oct 2014 20:23:32 GMT; path=/"
3 document.cookie = 'TEST=1';
4 console.log(document.cookie);
5 // → "TEST=1; expires=Tue, 14 Oct 2014 20:23:32 GMT; path=/"
```


Куки имеют максимальный объём данных 4Кб для одной пары ключ-значение, поэтому много информации записать в них не получится. Значений на один домен максимальное количество около 20 в зависимости от браузера.

 При работе локально без сервера все изменения куки будут проигнорированы. Поэтому, если вы работаете локально, у вас, скорее всего, будет выдаваться пустая строка.

Куки имеют несколько особенностей и приёмов, но для текущей работы нам достаточно полученного материала.

LocalStorage и SessionStorage

LocalStorage и SessionStorage представляют собой непосредственно хранилище браузера, в которое мы по своему усмотрению можем записать и считать данные из любого места скрипта. В отличие от cookie, они никак не зависят от наличия сервера и хранить можно гораздо больше данных.

 Хранить в LocalStorage и SessionStorage можно от 5 Мб данных в зависимости от настроек браузера. При желании, эту цифру можно изменить в настройках.

Ещё отличие от куки — сервер никак не знает о наличие хранилища и не может им управлять. Само хранилище зависит от источника (например, от сайта) и является разным для каждого источника.

Хранятся данные в хранилище в паре ключ-значение, причём и то и другое имеет тип «строка».

Данные localStorage не удаляются при закрытии браузера и хранятся там, пока какое-то событие их не изменит или очистит. Очистить данные можно в настройках браузера или специальной командой `clear()`, это метод объекта LocalStorage и SessionStorage.

SessionStorage, в отличие от LocalStorage, может хранить данные только в пределах одной вкладки браузера. При обновлении страницы данные сохраняются, но при закрытии браузера или вкладки удаляются. С этим и связано весьма ограниченное использование SessionStorage.

Оба объекта являются дочерними глобального объекта window, поэтому могут использоваться в любом месте скрипта.

Методы у LocalStorage и SessionStorage одинаковы.

- `setItem(ключ, значение)` — сохранить элемент с ключом «ключ» и данными «значение».
- `getItem(ключ)` — получить значение по ключу.
- `removeItem(ключ)` — удалить пару «ключ» и «значение» по заданному ключу.
- `key(номер позиции)` — получить ключ на заданной позиции.
- `length` — количество элементов в хранилище.
- `clear()` — очистка хранилища.

Попробуем поработать с данными хранилища.

```
1 console.log(localStorage);
```

```
▼ Storage ⓘ  
  length: 0  
  ► [[Prototype]]: Storage
```

Так выглядит пустое хранилище. Это у нас класс типа Storage. Запишем в него данные:

```
1 localStorage.setItem('Собачка', 'Жучка');  
2 localStorage.setItem('Кошечка', 'Мурка');  
3 localStorage.setItem(true, false);  
4  
5 console.log(localStorage);
```

```
▼ Storage {Кошечка: 'Мурка', true: 'false', Собачка: 'Жучка', length: 3}  
  true: "false"  
  Кошечка: "Мурка"  
  Собачка: "Жучка"  
  length: 3  
  ► [[Prototype]]: Storage
```

Здесь мы видим, что как ключом, так и значением могут быть строки не обязательно из латинского алфавита. Данные ключей и значений могут быть любыми, так как хранятся в строках. Обратимся по ключу:

```
1 localStorage.setItem('Собачка', 'Жучка');
2 localStorage.setItem('Кошечка', 'Мурка');
3
4 console.log(localStorage.getItem('Собачка'));
```

Нет проблем

Жучка



Так как у нас объект хранилища однородный, мы можем попробовать его поитерировать, но...

```
1 localStorage.setItem('Собачка', 'Жучка');
2 localStorage.setItem('Кошечка', 'Мурка');
3
4 for(let key in localStorage){
5     console.log(localStorage[key]);
6 }
```

Мурка	first.js:8
Жучка	first.js:8
2	first.js:8
<i>f clear() { [native code] }</i>	first.js:8
<i>f getItem() { [native code] }</i>	first.js:8
<i>f key() { [native code] }</i>	first.js:8
<i>f removeItem() { [native code] }</i>	first.js:8
<i>f setItem() { [native code] }</i>	first.js:8

...нам, кроме значений, будут выданы и все внутренние методы и свойство length. Поэтому можно использовать более подходящие для этой цели `Object.keys` и `Object.values`:

```

1 localStorage.setItem('Собачка', 'Жучка');
2 localStorage.setItem('Кошечка', 'Мурка');
3
4 const keys = Object.keys(localStorage);
5 const values = Object.values(localStorage);
6
7 for(let i=0; i<localStorage.length; i++){
8     console.log(keys[i], values[i]);
9 }

```

Нет проблем

Кошечка Мурка

firs

Собачка Жучка

firs

Либо в использованном ранее цикле for...in... отфильтровать лишние элементы:

```

1 localStorage.setItem('Собачка', 'Жучка');
2 localStorage.setItem('Кошечка', 'Мурка');
3
4 for(let key in localStorage){
5     if (localStorage.hasOwnProperty(key)) console.log(key, localStorage[key]);
6 }

```

Кошечка Мурка

fi

Собачка Жучка

fi

Генераторы

Генераторы — это специальный тип функции, который может приостанавливать своё выполнение, принимать промежуточные данные и продолжать своё выполнение с новыми вводными.

При использовании итераторов ранее мы много внимания уделяли поддержке внутреннего состояния итератора. Генератор же представляет альтернативу итератору позволяя определить алгоритм перебора с помощью единственной функции, умеющей поддерживать собственное состояние.

Приведём пример использования генератора:

```
1 function* generator(){
2   let ret = 0;
3   yield ++ret;
4   yield ++ret;
5   return ++ret;
6 }
7
8 let it = generator();
9
10 console.log(it); // объект генератора
11
12 console.log(it.next().value); // {done: false, value: 1}
13 console.log(it.next().value); // 2
14 console.log(it.next().value); // 3
```

Сам генератор записывается как функция — выражение со звёздочкой после ключевого слова `function`. Промежуточные значения могут возвращаться с помощью оператора `yield`. Само выражение с объявлением генератора не вызывает его, а возвращает специальный объект генератора, с помощью которого мы можем управлять его выполнением.

Запустить выполнение генератора мы можем с помощью метода `next()`. Генератор выполняется до первого оператора `yield` и возвращает объект типа:

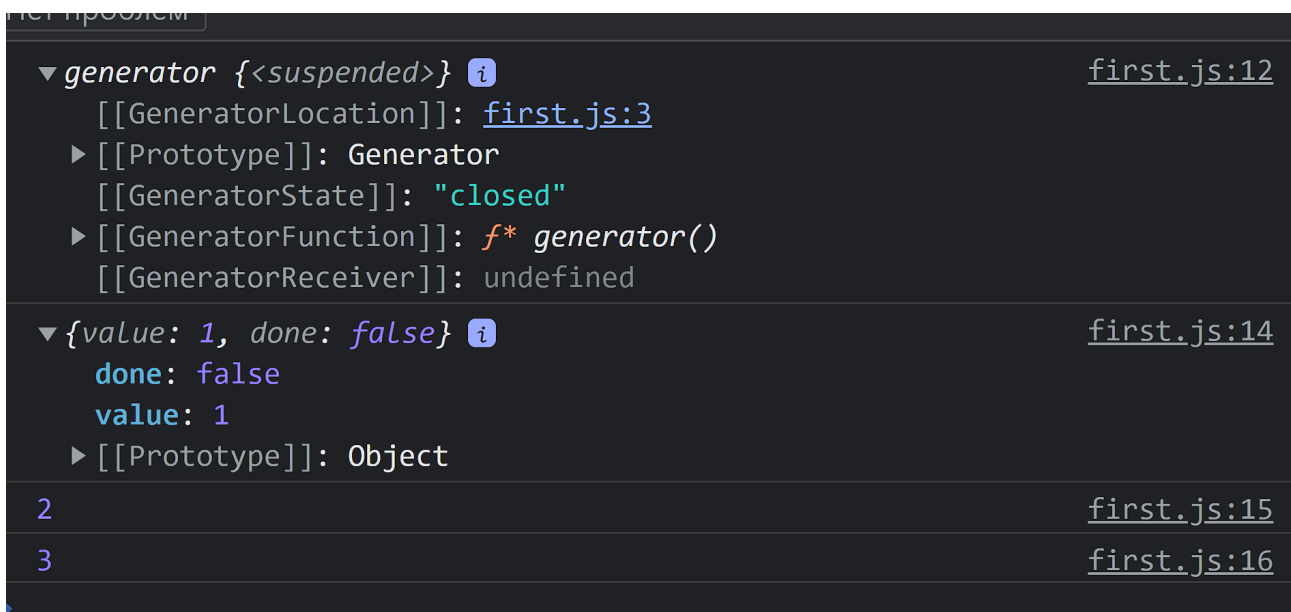
```

1 {
2   done: false,
3   value: 1
4 }
5 // Где value — это значение, переданное оператором yield

```

Повторные вызовы метода `next()` продолжат выполнение генератора с точки остановки. Внутреннее состояние генератора при этом сохраняется и мы можем использовать его контекст по нашему усмотрению.

Вот что нам выведется в консоль:



В уроке про итераторы мы собирали итератор именно таким же образом — с помощью метода `next()` и использования промежуточного объекта `{done, value}`. Это не случайно сделано одинаково. Генераторы тоже можно итерировать с помощью цикла `for(.. of ..)`.

```

1 цикла for(.. of ..).
2 function* generator(){
3   let ret = 0;
4   yield ++ret;
5   yield ++ret;
6   return ++ret;

```

```

7 }
8
9 let it = generator();
10
11 for(let value of it) {
12   console.log(value);
13 }

```

1	first.js:14
2	first.js:14
>	

Мы видим, что последнее значение проигнорировано. Это всё из-за того, что цикл `for(.. of ..)` игнорирует значение, когда `done` у нас становится равным `true`. Мы можем вместо `return` использовать `yield`, тогда значение у нас выведется.

```

1 function* generator(){
2   let ret = 0;
3   yield ++ret;
4   yield ++ret;
5   yield ++ret;
6 }
7
8 let it = generator();
9
10 for(let value of it) {
11   console.log(value);
12 }

```

Нет проблем

1	first.js:13
2	first.js:13
3	first.js:13
>	

Мы можем передавать данные в генератор также через оператор `yield`. Только нужно помнить, что переданные данные будут доступны только при следующем вызове генератора.

```
1 function* generator(){
2   let ret = 17;
3   let another_ret = yield;
4   console.log(ret, another_ret);
5   another_ret = yield ret;
6   console.log(ret, another_ret);
7 }
8
9 let it = generator();
10
11 it.next().value;
12 console.log(it.next(100).value);
13 it.next(10).value;
```

17 100	first.js:6
17	first.js:14
17 10	first.js:8

Подведём итоги

На этом уроке мы завершаем изучение продвинутых возможностей JavaScript.

Сегодня мы разобрали работу промисов. Промисы — мощный элемент работы с асинхронностью в JavaScript. Разобрали работу с массивами промисов и обработку ошибок, возникающих в них.

Сегодня изучили ещё работу с хранилищем браузера — с куки и со Storage. Выяснили, каким образом мы можем хранить данные и отправлять их на сервер.

В завершении урока мы разобрали работу особых функций — генераторов, которые могут продолжать работу после своей остановки.