Современные возможности ES-2015

Современный стандарт ES-2015 и его расширения для JavaScript.

ES-2015 сейчас

- Стандарт ES-2015 был принят в июне 2015. Пока что большинство браузеров реализуют его частично, текущее состояние реализации различных возможностей можно посмотреть здесь: https://kangax.github.io/compat-table/es6/.
- Когда стандарт будет более-менее поддерживаться во всех браузерах, то весь учебник будет обновлён в соответствии с ним. Пока же, как центральное место для «сбора» современных фич JavaScript, создан этот раздел.
- Чтобы писать код на ES-2015 прямо сейчас, есть следующие варианты.

Переменные: let и const

У объявлений переменной через let есть три основных отличия от var:
Область видимости переменной let – блок {...}.
Как мы помним, переменная, объявленная через var, видна везде в функции.
Переменная, объявленная через let, видна только в рамках блока {...}, в котором объявлена.
Это, в частности, влияет на объявления внутри if, while или for.

Например, переменная через var:

```
 var apples = 5;

if (true) {
  var apples = 10;

  alert(apples); // 10 (внутри блока)
}

alert(apples); // 10 (снаружи блока то же самое)
```

В примере выше apples – одна переменная на весь код, которая модифицируется в if.

То же самое с let будет работать по-другому:

```
let apples = 5; // (*)

if (true) {
  let apples = 10;

  alert(apples); // 10 (внутри блока)
}

alert(apples); // 5 (снаружи блока значение не изменилось)
```

Здесь, фактически, две независимые переменные apples, одна – глобальная, вторая – в блоке if.

Заметим, что если объявление let apples в первой строке (*) удалить, то в последнем alert будет ошибка: переменная не определена:

```
if (true) {
  let apples = 10;

  alert(apples); // 10 (внутри блока)
}
```

Это потому что переменная let всегда видна именно в том блоке, где объявлена, и не более.

Переменная let видна только после объявления.

Как мы помним, переменные var существуют и до объявления. Они равны undefined:

```
alert(a); // undefined
var a = 5
```

С переменными let всё проще. До объявления их вообще нет.
Такой доступ приведёт к ошибке:

```
alert(a); // ошибка, нет такой переменной
let a = 5;
```

Заметим также, что переменные let нельзя повторно объявлять. То есть, такой код выведет ошибку:

```
let x;
let x; // ошибка: переменная x уже объявлена
```

Это – хоть и выглядит ограничением по сравнению с var, но на самом деле проблем не создаёт. Например, два таких цикла совсем не конфликтуют:

```
// каждый цикл имеет свою переменную i
for(let i = 0; i<10; i++) { /* … */ }
for(let i = 0; i<10; i++) { /* … */ }

alert( i ); // ошибка: глобальной i нет
```

При объявлении внутри цикла переменная i будет видна только в блоке цикла. Она не видна снаружи, поэтому будет ошибка в последнем alert.

При использовании в цикле, для каждой итерации создаётся своя переменная.

Переменная var – одна на все итерации цикла и видна даже после цикла:

```
 for(var i=0; i<10; i++) { /* … */ }

alert(i); // 10
```

С переменной let – всё по-другому.

Каждому повторению цикла соответствует своя независимая переменная let. Если внутри цикла есть вложенные объявления функций, то в замыкании каждой будет та переменная, которая была при соответствующей итерации.

```
function makeArmy() {

  let shooters = [];

  for (let i = 0; i < 10; i++) {
    shooters.push(function() {
      alert( i ); // выводит свой номер
    });
  }

  return shooters;
}

var army = makeArmy();

army[0](); // 0
army[5](); // 5
```

Если бы объявление было var i, то была бы одна переменная i на всю функцию, и вызовы в последних строках выводили бы 10 (подробнее – см. задачу Армия функций).

А выше объявление let i создаёт для каждого повторения блока в цикле свою переменную, которую функция и получает из замыкания в последних строках.

const

Объявление const задаёт константу, то есть переменную, которую нельзя менять:

```
const apple = 5;
apple = 10; // ошибка
```

В остальном объявление const полностью аналогично let.

Заметим, что если в константу присвоен объект, то от изменения защищена сама константа, но не свойства внутри неё:

```
const user = {
  name: "Вася"
};

user.name = "Петя"; // допустимо
user = 5; // нельзя, будет ошибка
```

Константы, которые жёстко заданы всегда, во время всей программы, обычно пишутся в верхнем регистре. Например: const ORANGE = "#ffa500".

Большинство переменных – константы в другом смысле: они не меняются после присвоения. Но при разных запусках функции это значение может быть разным. Для таких переменных можно использовать const и обычные строчные буквы в имени.

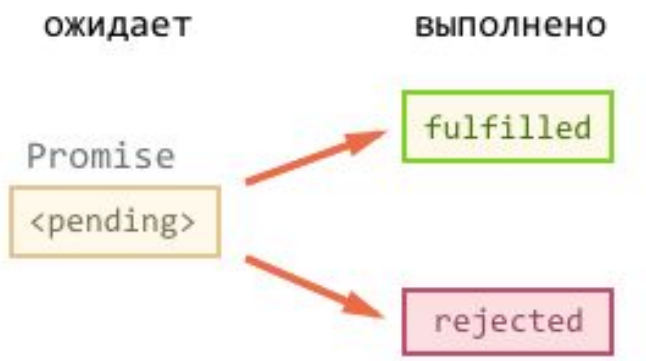
Promise

Promise (обычно их так и называют «промисы») – предоставляют удобный способ организации асинхронного кода.

В современном JavaScript промисы часто используются в том числе и неявно, при помощи генераторов, но об этом чуть позже.

Что такое Promise?

Promise – это специальный объект, который содержит своё состояние. Вначале pending («ожидание»), затем – одно из: fulfilled («выполнено успешно») или rejected («выполнено с ошибкой»).



На promise можно навешивать коллбэки двух типов:

onFulfilled – срабатывают, когда promise в состоянии «выполнен успешно».
onRejected – срабатывают, когда promise в состоянии «выполнен с ошибкой».
Способ использования, в общих чертах, такой:

Код, которому надо сделать что-то асинхронно, создаёт объект promise и возвращает его.
Внешний код, получив promise, навешивает на него обработчики.
По завершении процесса асинхронный код переводит promise в состояние fulfilled (с результатом) или rejected (с ошибкой). При этом автоматически вызываются соответствующие обработчики во внешнем коде.
Синтаксис создания Promise:

```
var promise = new Promise(function(resolve, reject) {
  // Эта функция будет вызвана автоматически

  // В ней можно делать любые асинхронные операции,
  // А когда они завершатся — нужно вызвать одно из:
  // resolve(результат) при успешном выполнении
  // reject(ошибка) при ошибке
})
```

Универсальный метод для навешивания обработчиков:

promise.then(onFulfilled, onRejected)
onFulfilled – функция, которая будет вызвана с результатом при resolve.
onRejected – функция, которая будет вызвана с ошибкой при reject.
С его помощью можно назначить как оба обработчика сразу, так и только один:

```
// onFulfilled сработает при успешном выполнении
promise.then(onFulfilled)
// onRejected сработает при ошибке
promise.then(null, onRejected)
```

Для того, чтобы поставить обработчик только на ошибку, вместо .then(null, onRejected) можно написать .catch(onRejected) – это то же самое.

Если в функции промиса происходит синхронный throw (или иная ошибка), то вызывается reject:

```
'use strict';
let p = new Promise((resolve, reject) => {
  // то же что reject(new Error("o_O"))
  throw new Error("o_O");
});
p.catch(alert); // Error: o_O
```

Возьмём setTimeout в качестве асинхронной операции, которая должна через некоторое время успешно завершиться с результатом «result»:

```
'use strict';

// Создаётся объект promise
let promise = new Promise((resolve, reject) => {

  setTimeout(() => {
    // переведёт промис в состояние fulfilled с результатом
"result"
    resolve("result");
  }, 1000);

});

// promise.then навешивает обработчики на успешный
результат или ошибку
promise
  .then(
    result => {
      // первая функция-обработчик - запустится при вызове
resolve
      alert("Fulfilled: " + result); // result - аргумент resolve
    },
    error => {
      // вторая функция - запустится при вызове reject
      alert("Rejected: " + error); // error - аргумент reject
    }
  );
```

В результате запуска кода выше – через 1 секунду выведется «Fulfilled: result».

А если бы вместо resolve("result") был вызов reject("error"), то вывелось бы «Rejected: error». Впрочем, как правило, если при выполнении возникла проблема, то reject вызывают не со строкой, а с объектом ошибки типа new Error:

```
// Этот promise завершится с ошибкой через 1 секунду
var promise = new Promise((resolve, reject) => {

  setTimeout(() => {
    reject(new Error("время вышло!"));
  }, 1000);

});

promise
  .then(
    result => alert("Fulfilled: " + result),
    error => alert("Rejected: " + error.message) // Rejected: время вышло!
  );
```

Конечно, вместо setTimeout внутри функции промиса может быть и запрос к серверу и ожидание ввода пользователя, или другой асинхронный процесс. Главное, чтобы по своему завершению он вызвал resolve или reject, которые передадут результат обработчикам.

Функции resolve/reject принимают ровно один аргумент – результат/ошибку.

Именно он передаётся обработчикам в .then, как можно видеть в примерах выше.

Promise после reject/resolve – неизменны

Заметим, что после вызова resolve/reject промис уже не может «передумать».

Когда промис переходит в состояние «выполнен» – с результатом (resolve) или ошибкой (reject) – это навсегда.

В результате вызова этого кода сработает только первый обработчик then, так как после вызова resolve промис уже получил состояние (с результатом), и в дальнейшем его уже ничто не изменит.

Последующие вызовы resolve/reject будут просто проигнорированы.

```
use strict';

let promise = new Promise((resolve, reject) => {

  // через 1 секунду готов результат: result
  setTimeout(() => resolve("result"), 1000);

  // через 2 секунды — reject с ошибкой, он будет проигнорирован
  setTimeout(() => reject(new Error("ignored")), 2000);

});

promise
  .then(
    result => alert("Fulfilled: " + result), // сработает
    error => alert("Rejected: " + error) // не сработает
  );
```

Промисификация

Промисификация – это когда берут асинхронный функционал и делают для него обёртку, возвращающую промис.

После промисификации использование функционала зачастую становится гораздо удобнее.

В качестве примера сделаем такую обёртку для запросов при помощи XMLHttpRequest.

Функция httpGet(url) будет возвращать промис, который при успешной загрузке данных с url будет переходить в fulfilled с этими данными, а при ошибке – в rejected с информацией об ошибке:

```
function httpGet(url) {

  return new Promise(function(resolve, reject) {

    var xhr = new XMLHttpRequest();
    xhr.open('GET', url, true);

    xhr.onload = function() {
      if (this.status == 200) {
        resolve(this.response);
      } else {
        var error = new Error(this.statusText);
        error.code = this.status;
        reject(error);
      }
    };

    xhr.onerror = function() {
      reject(new Error("Network Error"));
    };
    xhr.send();
  });

}
```

Как видно, внутри функции объект XMLHttpRequest создаётся и отсылается как обычно, при onload/onerror вызываются, соответственно, resolve (при статусе 200) или reject.

Использование:

```
httpGet("/article/promise/user.json")
  .then(
    response => alert(`Fulfilled: ${response}`),
    error => alert(`Rejected: ${error}`)
  );
```

Заметим, что ряд современных браузеров уже поддерживает fetch – новый встроенный метод для AJAX-запросов, призванный заменить XMLHttpRequest. Он гораздо мощнее, чем httpGet. И – да, этот метод использует промисы.

Цепочки промисов

Чейнинг» (chaining), то есть возможность строить асинхронные цепочки из промисов – пожалуй, основная причина, из-за которой существуют и активно используются промисы.

Например, мы хотим по очереди:

Загрузить данные посетителя с сервера (асинхронно).
Затем отправить запрос о нём на github (асинхронно).
Когда это будет готово, вывести его github-аватар на экран (асинхронно).
…И сделать код расширяемым, чтобы цепочку можно было легко продолжить.

Вот код для этого, использующий функцию httpGet, описанную выше:

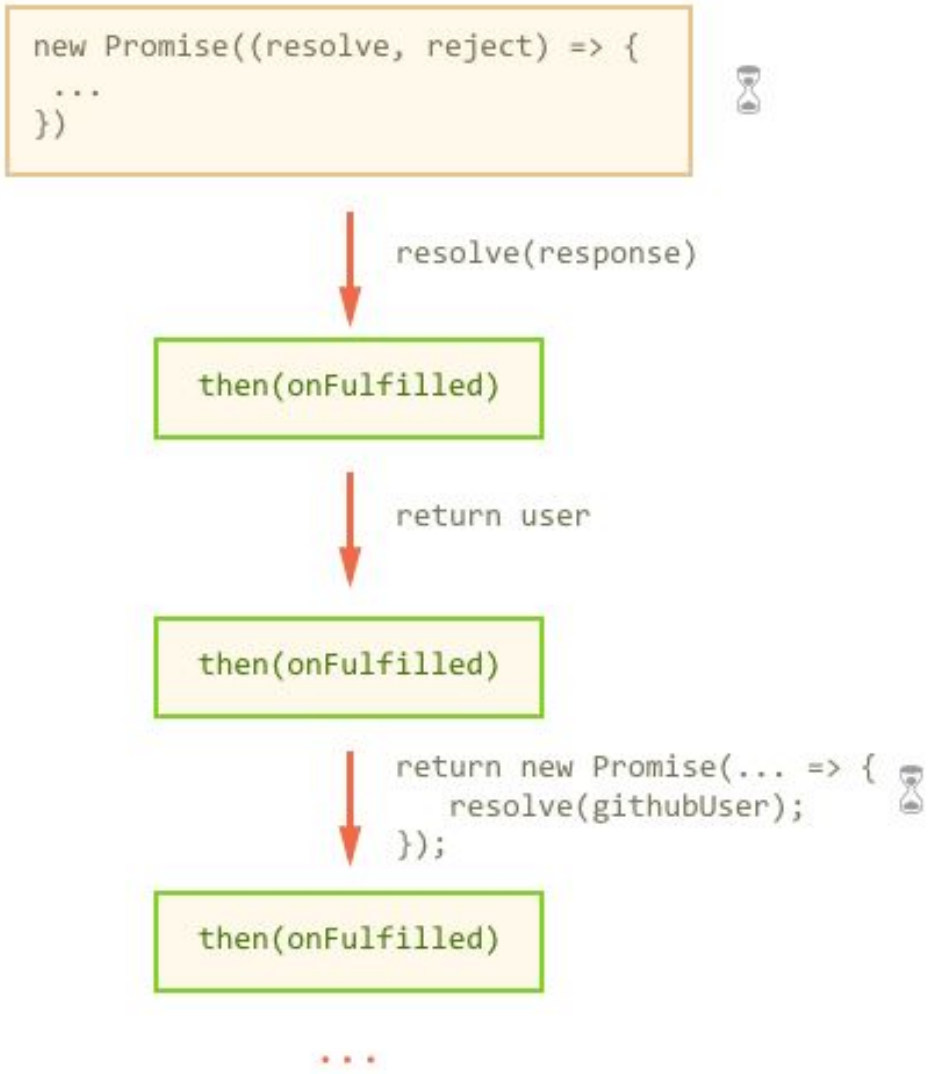
```
'use strict';

// сделать запрос
httpGet('/article/promise/user.json')
  // 1. Получить данные о пользователе в JSON и передать дальше
  .then(response => {
    console.log(response);
    let user = JSON.parse(response);
    return user;
  })
  // 2. Получить информацию с github
  .then(user => {
    console.log(user);
    return httpGet(`https://api.github.com/users/${user.name}`);
  })
  // 3. Вывести аватар на 3 секунды (можно с анимацией)
  .then(githubUser => {
    console.log(githubUser);
    githubUser = JSON.parse(githubUser);

    let img = new Image();
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.appendChild(img);

    setTimeout(() => img.remove(), 3000); // (*)
  });
```

При чейнинге, то есть последовательных вызовах .then…then…then, в каждый следующий then переходит результат от предыдущего. Вызовы console.log оставлены, чтобы при запуске можно было посмотреть конкретные значения, хотя они здесь и не очень важны.

Если очередной then вернул промис, то далее по цепочке будет передан не сам этот промис, а его результат.

В коде выше:

Функция в первом then возвращает «обычное» значение user. Это значит, что then возвратит промис в состоянии «выполнен» с user в качестве результата. Он станет аргументом в следующем then.
Функция во втором then возвращает промис (результат нового вызова httpGet). Когда он будет завершён (может пройти какое-то время), то будет вызван следующий then с его результатом.
Третий then ничего не возвращает.
Схематично его работу можно изобразить так:



Значком «песочные часы» помечены периоды ожидания, которых всего два: в исходном httpGet и в подвызове далее по цепочке.

Если then возвращает промис, то до его выполнения может пройти некоторое время, оставшаяся часть цепочки будет ждать.

То есть, логика довольно проста:

В каждом then мы получаем текущий результат работы.
Можно его обработать синхронно и вернуть результат (например, применить JSON.parse). Или же, если нужна асинхронная обработка – инициировать её и вернуть промис.
Обратим внимание, что последний then в нашем примере ничего не возвращает. Если мы хотим, чтобы после setTimeout (*) асинхронная цепочка могла быть продолжена, то последний then тоже должен вернуть промис. Это общее правило: если внутри then стартует новый асинхронный процесс, то для того, чтобы оставшаяся часть цепочки выполнилась после его окончания, мы должны вернуть промис.

В данном случае промис должен перейти в состояние «выполнен» после срабатывания setTimeout.

```
.then(githubUser => {
  ...

  // вместо setTimeout(() => img.remove(), 3000); (*)

  return new Promise((resolve, reject) => {
    setTimeout(() => {
      img.remove();
      // после таймаута — вызов resolve,
      // можно без результата, чтобы управление перешло в следующий then
      // (или можно передать данные пользователя дальше по цепочке)
      resolve();
    }, 3000);
  });
})
```

Теперь, если к цепочке добавить ещё then, то он будет вызван после окончания setTimeout.

Генераторы

Генераторы – новый вид функций в современном JavaScript. Они отличаются от обычных тем, что могут приостанавливать своё выполнение, возвращать промежуточный результат и далее возобновлять его позже, в произвольный момент времени.

Создание генератора

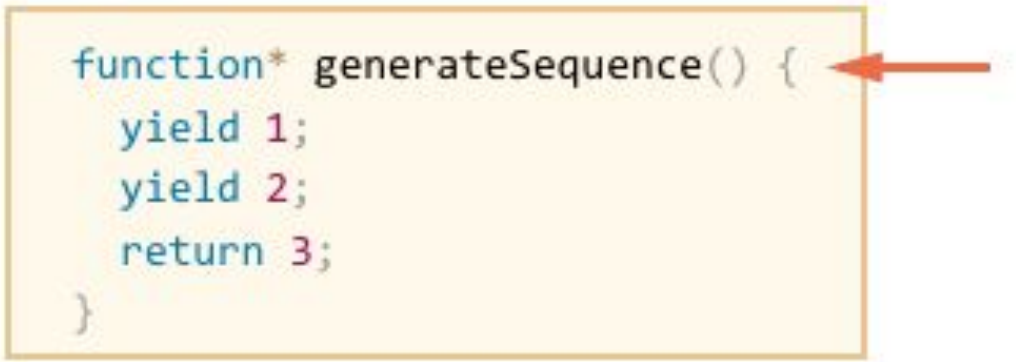
Для объявления генератора используется новая синтаксическая конструкция: function* (функция со звёздочкой).

Её называют «функция-генератор» (generator function).

Выглядит это так:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

При запуске generateSequence() код такой функции не выполняется. Вместо этого она возвращает специальный объект, который как раз и называют «генератором».



// generator function создаёт generator
let generator = generateSequence();
Правильнее всего будет воспринимать генератор как «замороженный вызов функции»:

При создании генератора код находится в начале своего выполнения.

Основным методом генератора является next(). При вызове он возобновляет выполнение кода до ближайшего ключевого слова yield. По достижении yield выполнение приостанавливается, а значение – возвращается во внешний код:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```


```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();

let one = generator.next();

alert(JSON.stringify(one)); // {value: 1, done: false}
```

Повторный вызов generator.next() возобновит выполнение и вернёт результат следующего yield:

```
let two = generator.next();

alert(JSON.stringify(two)); // {value: 2, done: false}
```


И, наконец, последний вызов завершит выполнение функции и вернёт результат return:

```
let three = generator.next();

alert(JSON.stringify(three)); // {value: 3, done: true}
```

Генератор – итератор

Как вы, наверно, уже догадались по наличию метода next(), генератор связан с итераторами. В частности, он является итерируемым объектом.

Его можно перебирать и через for..of:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1, затем 2
}
```

Заметим, однако, существенную особенность такого перебора!

При запуске примера выше будет выведено значение 1, затем 2. Значение 3 выведено не будет. Это потому что стандартный перебор итератора игнорирует value на последнем значении, при done: true. Так что результат return в цикле for..of не выводится.

Соответственно, если мы хотим, чтобы все значения возвращались при переборе через for..of, то надо возвращать их через yield:

```
function* generateSequence() {
  yield 1;
  yield 2;
  yield 3;
}

let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1, затем 2, затем 3
}
```

…А зачем вообще return при таком раскладе, если его результат игнорируется? Он тоже нужен, но в других ситуациях. Перебор через for..of – в некотором смысле «исключение». Как мы увидим дальше, в других контекстах return очень даже востребован.

Композиция генераторов

Один генератор может включать в себя другие. Это называется композицией.

Разберём композицию на примере.

Пусть у нас есть функция generateSequence, которая генерирует последовательность чисел:

```
function* generateSequence(start, end) {

  for (let i = start; i <= end; i++) {
    yield i;
  }

}

// Используем оператор … для преобразования
итерируемого объекта в массив
let sequence = [...generateSequence(2,5)];

alert(sequence); // 2, 3, 4, 5
```

Мы хотим на её основе сделать другую функцию generateAlphaNumCodes(), которая будет генерировать коды для буквенно-цифровых символов латинского алфавита:

48..57 – для 0..9
65..90 – для A..Z
97..122 – для a..z
Далее этот набор кодов можно превратить в строку и использовать, к примеру, для выбора из него случайного пароля. Только символы пунктуации ещё хорошо бы добавить для надёжности, но в этом примере мы будем без них.

Естественно, раз в нашем распоряжении есть готовый генератор generateSequence, то хорошо бы его использовать.

Конечно, можно внутри generateAlphaNum запустить несколько раз generateSequence, объединить результаты и вернуть. Так мы бы сделали с обычными функциями. Но композиция – это кое-что получше.

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}

function* generateAlphaNum() {

  // 0..9
  yield* generateSequence(48, 57);

  // A..Z
  yield* generateSequence(65, 90);

  // a..z
  yield* generateSequence(97, 122);

}

let str = '';

for(let code of generateAlphaNum()) {
  str += String.fromCharCode(code);
}

alert(str); // 0..9A..Za..z
```

Основы XMLHttpRequest

- Объект XMLHttpRequest (или, как его кратко называют, «XHR») дает возможность из JavaScript делать HTTP-запросы к серверу без перезагрузки страницы.

- Несмотря на слово «XML» в названии, XMLHttpRequest может работать с любыми данными, а не только с XML.

- Использовать его очень просто.

Как правило, XMLHttpRequest используют для загрузки данных.

Для начала посмотрим на пример использования, который загружает файл phones.json из текущей директории и выдаёт его содержимое:

```
// 1. Создаём новый объект XMLHttpRequest
var xhr = new XMLHttpRequest();

// 2. Конфигурируем его: GET-запрос на URL 'phones.json'
xhr.open('GET', 'phones.json', false);

// 3. Отсылаем запрос
xhr.send();

// 4. Если код ответа сервера не 200, то это ошибка
if (xhr.status != 200) {
  // обработать ошибку
  alert( xhr.status + ': ' + xhr.statusText ); // пример вывода: 404: Not Found
} else {
  // вывести результат
  alert( xhr.responseText ); // responseText -- текст ответа.
}
```

Настроить: open

xhr.open(method, URL, async, user, password)

Этот метод – как правило, вызывается первым после создания объекта XMLHttpRequest.

Задаёт основные параметры запроса:
method – HTTP-метод. Как правило, используется GET либо POST, хотя доступны и более экзотические, вроде TRACE/DELETE/PUT и т.п.
URL – адрес запроса. Можно использовать не только http/https, но и другие протоколы, например ftp:// и file://.

При этом есть ограничения безопасности, называемые «Same Origin Policy»: запрос со страницы можно отправлять только на тот же протокол://домен:порт, с которого она пришла. В следующих главах мы рассмотрим, как их можно обойти.

async – если установлено в false, то запрос производится синхронно, если true – асинхронно.

«Синхронный запрос» означает, что после вызова xhr.send() и до ответа сервера главный поток будет «заморожен»: посетитель не сможет взаимодействовать со страницей – прокручивать, нажимать на кнопки и т.п. После получения ответа выполнение продолжится со следующей строки.

«Асинхронный запрос» означает, что браузер отправит запрос, а далее результат нужно будет получить через обработчики событий, которые мы рассмотрим далее.

user, password – логин и пароль для HTTP-авторизации, если нужны.

Отослать данные: send

xhr.send([body])

Именно этод метод открывает соединение и отправляет запрос на сервер.

В body находится тело запроса. Не у всякого запроса есть тело, например у GET-запросов тела нет, а у POST – основные данные как раз передаются через body.

Синхронные и асинхронные запросы

Если в методе open установить параметр async равным false, то запрос будет синхронным.

Синхронные вызовы используются чрезвычайно редко, так как блокируют взаимодействие со страницей до окончания загрузки. Посетитель не может даже прокручивать её. Никакой JavaScript не может быть выполнен, пока синхронный вызов не завершён – в общем, в точности те же ограничения как alert.

Если синхронный вызов занял слишком много времени, то браузер предложит закрыть «зависшую» страницу.

Из-за такой блокировки получается, что нельзя отослать два запроса одновременно. Кроме того, забегая вперёд, заметим, что ряд продвинутых возможностей, таких как возможность делать запросы на другой домен и указывать таймаут, в синхронном режиме не работают.

Из всего вышесказанного уже должно быть понятно, что синхронные запросы используются чрезвычайно редко, а асинхронные – почти всегда.

Для того, чтобы запрос стал асинхронным, укажем

```
// Синхронный запрос
xhr.open('GET', 'phones.json', false);

// Отсылаем его
xhr.send();
// ...весь JavaScript "подвиснет", пока запрос не завершится
```

```
var xhr = new XMLHttpRequest();

xhr.open('GET', 'phones.json', true);

xhr.send(); // (1)

xhr.onreadystatechange = function() { // (3)
  if (xhr.readyState != 4) return;

  button.innerHTML = 'Готово!';

  if (xhr.status != 200) {
    alert(xhr.status + ': ' + xhr.statusText);
  } else {
    alert(xhr.responseText);
  }

}

button.innerHTML = 'Загружаю...'; // (2)
button.disabled = true;
```

Если в open указан третий аргумент true (или если третьего аргумента нет), то запрос выполняется асинхронно. Это означает, что после вызова xhr.send() в строке (1) код не «зависает», а преспокойно продолжает выполняться, выполняется строка (2), а результат приходит через событие (3), мы изучим его чуть позже.