Массив: перебирающие методы

- `forEach` – для *перебора* массива (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach);
- `filter` – для *фильтрации* массива (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter);
- `every/some` – для *проверки* массива (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/every);
- `map` – для *трансформации* массива в массив (https://developer.mozilla.org/en-US/docs/Web/ JavaScript/Reference/ Global_Objects/ Array/map);
- `reduce/reduceRight` – для *прохода по массиву с вычислением значения* (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce).

forEach

Метод «arr.forEach(callback[, thisArg])» используется для перебора массива.

Он для каждого элемента массива вызывает функцию `callback`.

Этой функции он передаёт три параметра `callback(item, i, arr)`:

- `item` – очередной элемент массива.
- `i` – его номер.
- `arr` – массив, который перебирается.

Например:

```
var arr = ["Яблоко", "Апельсин", "Груша"];

arr.forEach(function(item, i, arr) {
Console.log ( i + ": " + item + " (массив:" + arr + ")" );
}); // 0: Яблоко (массив:Яблоко,Апельсин,Груша)
```

Второй, необязательный аргумент `forEach` позволяет указать контекст `this` для `callback`. Мы обсудим его в деталях чуть позже, сейчас он нам не важен.

Метод `forEach` ничего не возвращает, его используют только для перебора, как более «элегантный» вариант, чем обычный цикл `for`.

filter

Метод «arr.filter(callback[, thisArg])» используется для *фильтрации* массива через функцию.

Он создаёт новый массив, в который войдут только те элементы `arr`, для которых вызов `callback(item, i, arr)` возвратит `true`.

Например:

```
var arr = [1, -1, 2, -2, 3];

var positiveArr = arr.filter(function(number) {
  return number > 0;
});
Console.log(positiveArr); // 1,2,3
```

map

Метод «arr.map(callback[, thisArg])» используется для *трансформации* массива.

Он создаёт новый массив, который будет состоять из результатов вызова `callback(item, i, arr)` для каждого элемента `arr`.

Например:

```
var names = ['HTML', 'CSS', 'JavaScript'];

var nameLengths = names.map(function(name) {
  return name.length;
});
// получили массив с длинами
Console.log ( nameLengths ); // 4,3,10
```

every/some

Эти методы используются для проверки массива.

- Метод «arr.every(callback[, thisArg])» возвращает `true`, если вызов `callback` вернёт `true` для *каждого* элемента `arr`.
- Метод «arr.some(callback[, thisArg])» возвращает `true`, если вызов `callback` вернёт `true` для *какого-нибудь* элемента `arr`.

```
var arr = [1, -1, 2, -2, 3];
function isPositive(number) {
  return number > 0;
}
Console.log ( arr.every(isPositive) ); // false, не все положительные
Console.log ( arr.some(isPositive) ); // true, есть хоть одно положительное
```

reduce/reduceRight

Метод «arr.reduce(callback[, initialValue])» используется для последовательной обработки каждого элемента массива с сохранением промежуточного результата.

Это один из самых сложных методов для работы с массивами. Но его стоит освоить, потому что временами с его помощью можно в несколько строк решить задачу, которая иначе потребовала бы в разы больше места и времени.

Метод `reduce` используется для вычисления на основе массива какого-либо единого значения, иначе говорят «для свёртки массива». Чуть далее мы разберём пример для вычисления суммы.

Он применяет функцию `callback` по очереди к каждому элементу массива слева направо, сохраняя при этом промежуточный результат.

Аргументы функции `callback(previousValue, currentItem, index, arr)`:

- `previousValue` – последний результат вызова функции, он же «промежуточный результат».
- `currentItem` – текущий элемент массива, элементы перебираются по очереди слева-направо.
- `index` – номер текущего элемента.
- `arr` – обрабатываемый массив.

Кроме `callback`, методу можно передать «начальное значение» – аргумент `initialValue`. Если он есть, то на первом вызове значение `previousValue` будет равно `initialValue`, а если у `reduce` нет второго аргумента, то оно равно первому элементу массива, а перебор начинается со второго.

Проще всего понять работу метода `reduce` на примере.

Например, в качестве «свёртки» мы хотим получить сумму всех элементов массива.

Вот решение в одну строку:

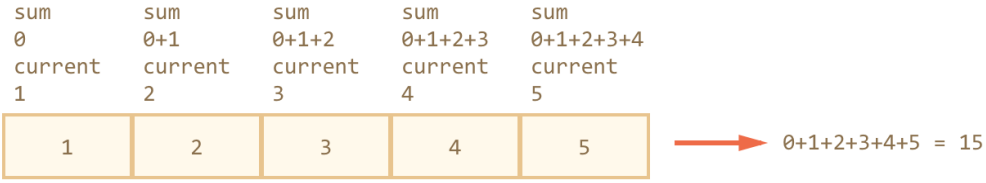
```
var arr = [1, 2, 3, 4, 5]

// для каждого элемента массива запустить функцию,
// промежуточный результат передавать первым аргументом далее
var result = arr.reduce(function(sum, current) {
  return sum + current;
}, 0);
Console.log ( result ); // 15
```

Разберём, что в нём происходит.

При первом запуске `sum` – исходное значение, с которого начинаются вычисления, равно нулю (второй аргумент `reduce`).

Сначала анонимная функция вызывается с этим начальным значением и первым элементом массива, результат запоминается и передаётся в следующий вызов, уже со вторым аргументом массива, затем новое значение участвует в вычислениях с третьим аргументом и так далее.

Поток вычислений получается такой



В виде таблицы где каждая строка – вызов функции на очередном элементе массива:

	`sum`	`current`	**результат**
первый вызов	`0`	`1`	`1`
второй вызов	`1`	`2`	`3`
третий вызов	`3`	`3`	`6`
четвёртый вызов	`6`	`4`	`10`
пятый вызов	`10`	`5`	`15`

Как видно, результат предыдущего вызова передаётся в первый аргумент следующего.

Кстати, полный набор аргументов функции для `reduce` включает в себя `function(sum, current, i, array)`, то есть номер текущего вызова `i` и весь массив `arr`, но здесь в них нет нужды.

Посмотрим, что будет, если не указать `initialValue` в вызове `arr.reduce`:

```
var arr = [1, 2, 3, 4, 5]

// убрали 0 в конце
var result = arr.reduce(function(sum, current) {
  return sum + current
});
Console.log ( result ); // 15
```

Результат – точно такой же! Это потому, что при отсутствии `initialValue` в качестве первого значения берётся первый элемент массива, а перебор стартует со второго.

Таблица вычислений будет такая же, за вычетом первой строки.

Метод arr.reduceRight работает аналогично, но идёт по массиву справа-налево.

Массивы: методы

(https://learn.javascript.ru/array-methods)

MDN Web docs (moz://a): https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Методы массивов:

- `push/pop`, `shift/unshift`, `splice` – для добавления и удаления элементов.
- `join/split` – для преобразования строки в массив и обратно.
- `slice` – копирует участок массива.
- `sort` – для сортировки массива. Если не передать функцию сравнения – сортирует элементы как строки.
- `reverse` – меняет порядок элементов на обратный.
- `concat` – объединяет массивы.
- `indexOf/lastIndexOf` – возвращают позицию элемента в массиве (не поддерживается в IE8-).

Дополнительно:

- `Object.keys(obj)` возвращает массив свойств объекта.

Метод split

Ситуация из реальной жизни. Мы пишем сервис отсылки сообщений и посетитель вводит имена тех, кому его отправить: `Маша, Петя, Марина, Василий...`. Но нам-то гораздо удобнее работать с массивом имен, чем с одной строкой.

К счастью, есть метод `split(s)`, который позволяет превратить строку в массив, разбив ее по разделителю `s`. В примере ниже таким разделителем является строка из запятой и пробела.

```
var names = 'Маша, Петя, Марина, Василий';

var arr = names.split(', ');

for (var i = 0; i < arr.length; i++) {
  alert( 'Вам сообщение ' + arr[i] );
}
```

Второй аргумент `split`

У метода `split` есть необязательный второй аргумент – ограничение на количество элементов в массиве. Если их больше, чем указано – остаток массива будет отброшен:

```
alert( "a,b,c,d".split(',', 2) ); // a,b
```

Разбивка по буквам

Вызов `split` с пустой строкой разобьёт по буквам:

```
var str = "тест";
alert( str.split('') ); // т,е,с,т
```

Метод join

Вызов `arr.join(str)` делает в точности противоположное `split`. Он берет массив и склеивает его в строку, используя `str` как разделитель.

Например:

```
var arr = ['Маша', 'Петя', 'Марина', 'Василий'];
var str = arr.join(';');
alert( str ); // Маша;Петя;Марина;Василий
```

new Array + join = Повторение строки

Код для повторения строки 3 раза:

```
alert( new Array(4).join("ля") ); // ляляля
```

Как видно, `new Array(4)` делает массив без элементов длины 4, который `join` объединяет в строку, вставляя *между его элементами* строку `"ля"`.

В результате, так как элементы пусты, получается повторение строки. Такой вот небольшой трюк.

Удаление из массива

Так как массивы являются объектами, то для удаления ключа можно воспользоваться обычным `delete`:

```
var arr = ["Я", "иду", "домой"];

delete arr[1]; // значение с индексом 1 удалено

// теперь arr = ["Я", undefined, "домой"];
alert( arr[1] ); // undefined
```

Да, элемент удален из массива, но не так, как нам этого хочется. Образовалась «дырка».

Это потому, что оператор `delete` удаляет пару «ключ-значение». Это – все, что он делает. Обычно же при удалении из массива мы хотим, чтобы оставшиеся элементы сдвинулись и заполнили образовавшийся промежуток.

Поэтому для удаления используются специальные методы: из начала – `shift`, с конца – `pop`, а из середины – `splice`, с которым мы сейчас познакомимся.

Метод splice

Метод `splice` – это универсальный раскладной нож для работы с массивами. Умеет все: удалять элементы, вставлять элементы, заменять элементы – по очереди и одновременно.

Его синтаксис:

```
arr.splice(index[, deleteCount, elem1, ..., elemN])
```

Удалить `deleteCount` элементов, начиная с номера `index`, а затем вставить `elem1, ..., elemN` на их место. Возвращает массив из удалённых элементов.

Этот метод проще всего понять, рассмотрев примеры.

Начнём с удаления:

```
var arr = ["Я", "изучаю", "JavaScript"];
arr.splice(1, 1); // начиная с позиции 1, удалить 1 элемент
alert( arr ); //  осталось ["Я", "JavaScript"]
```

В следующем примере мы удалим 3 элемента и вставим другие на их место:

```
var arr = ["Я", "сейчас", "изучаю", "JavaScript"];

// удалить 3 первых элемента и добавить другие вместо них
```

```
arr.splice(0, 3, "Мы", "изучаем")

alert( arr ) // теперь ["Мы", "изучаем", "JavaScript"]
```

Здесь видно, что `splice` возвращает массив из удаленных элементов:

```
var arr = ["Я", "сейчас", "изучаю", "JavaScript"];

// удалить 2 первых элемента
var removed = arr.splice(0, 2);

alert( removed ); // "Я", "сейчас" <-- array of removed elements
```

Метод `splice` также может вставлять элементы без удаления, для этого достаточно установить `deleteCount` в `0`:

```
var arr = ["Я", "изучаю", "JavaScript"];

// с позиции 2
// удалить 0
// вставить "сложный", "язык"
arr.splice(2, 0, "сложный", "язык");

alert( arr ); // "Я", "изучаю", "сложный", "язык", "JavaScript"
```

Допускается использование отрицательного номера позиции, которая в этом случае отсчитывается с конца:

```
var arr = [1, 2, 5]

// начиная с позиции индексом -1 (перед последним элементом)
// удалить 0 элементов,
// затем вставить числа 3 и 4
arr.splice(-1, 0, 3, 4);

alert( arr ); // результат: 1,2,3,4,5
```

Метод slice

Метод `slice(begin, end)` копирует участок массива от `begin` до `end`, не включая `end`. Исходный массив при этом не меняется.

Например:

```
var arr = ["Почему", "надо", "учить", "JavaScript"];

var arr2 = arr.slice(1, 3); // элементы 1, 2 (не включая 3)
```

```
alert( arr2 ); // надо, учить
```

Аргументы ведут себя так же, как и в строковом `slice`:

- Если не указать `end` – копирование будет до конца массива:

```
var arr = ["Почему", "надо", "учить", "JavaScript"];

alert( arr.slice(1) ); // взять все элементы, начиная с номера 1
```

- Можно использовать отрицательные индексы, они отсчитываются с конца:

```
var arr2 = arr.slice(-2); // копировать от 2-го элемента с конца и дальше
```

- Если вообще не указать аргументов – скопируется весь массив:

```
var fullCopy = arr.slice();
```

Совсем как в строках

Синтаксис метода `slice` одинаков для строк и для массивов. Тем проще его запомнить.

Сортировка, метод sort(fn)

Метод `sort()` сортирует массив *на месте*. Например:

```
var arr = [ 1, 2, 15 ];
arr.sort();
alert( arr );  // 1, 15, 2
```

Не заметили ничего странного в этом примере?

Порядок стал `1, 15, 2`, это точно не сортировка чисел. Почему?

Это произошло потому, что по умолчанию `sort` сортирует, преобразуя элементы к строке.

Поэтому и порядок у них строковый, ведь `"2" > "15"`.

Свой порядок сортировки

Для указания своего порядка сортировки в метод `arr.sort(fn)` нужно передать функцию `fn` от двух элементов, которая умеет сравнивать их.

Внутренний алгоритм функции сортировки умеет сортировать любые массивы – апельсинов, яблок, пользователей, и тех и других и третьих – чего угодно. Но для этого ему нужно знать, как их сравнивать. Эту роль и выполняет `fn`.

Если эту функцию не указать, то элементы сортируются как строки.

Например, укажем эту функцию явно, отсортируем элементы массива как числа:

```
function compareNumeric(a, b) {
```

```
  if (a > b) return 1;
  if (a < b) return -1;
}

var arr = [ 1, 2, 15 ];

arr.sort(compareNumeric);

alert(arr);  // 1, 2, 15
```

Обратите внимание, мы передаём в `sort()` именно саму функцию `compareNumeric`, без вызова через скобки. Был бы ошибкой следующий код:

```
arr.sort( compareNumeric() );  // не сработает
```

Как видно из примера выше, функция, передаваемая `sort`, должна иметь два аргумента.

Алгоритм сортировки, встроенный в JavaScript, будет передавать ей для сравнения элементы массива. Она должна возвращать:

- Положительное значение, если `a > b`,
- Отрицательное значение, если `a < b`,
- Если равны – можно `0`, но вообще – не важно, что возвращать, если их взаимный порядок не имеет значения.

Алгоритм сортировки

В методе `sort`, внутри самого интерпретатора JavaScript, реализован универсальный алгоритм сортировки. Как правило, это ««быстрая сортировка»», дополнительно оптимизированная для небольших массивов.

Он решает, какие пары элементов и когда сравнивать, чтобы отсортировать побыстрее. Мы даём ему функцию – способ сравнения, дальше он вызывает её сам.

Кстати, те значения, с которыми `sort` вызывает функцию сравнения, можно увидеть, если вставить в неё `alert`:

```
[1, -2, 15, 2, 0, 8].sort(function(a, b) {
  alert( a + " <> " + b );
});
```

Сравнение `compareNumeric` в одну строку

Функцию `compareNumeric` для сравнения элементов-чисел можно упростить до одной строчки.

```
function compareNumeric(a, b) {
```

```
  return a - b;
}
```

Эта функция вполне подходит для `sort`, так как возвращает положительное число, если `a > b`, отрицательное, если наоборот, и `0`, если числа равны.

reverse

Метод arr.reverse() меняет порядок элементов в массиве на обратный.

```
var arr = [1, 2, 3];
arr.reverse();

alert( arr ); // 3,2,1
```

concat

Метод arr.concat(value1, value2, ... valueN) создаёт новый массив, в который копируются элементы из `arr`, а также `value1, value2, ... valueN`.

Например:

```
var arr = [1, 2];
var newArr = arr.concat(3, 4);

alert( newArr ); // 1,2,3,4
```

У `concat` есть одна забавная особенность.

Если аргумент `concat` – массив, то `concat` добавляет элементы из него.

Например:

```
var arr = [1, 2];

var newArr = arr.concat([3, 4], 5); // то же самое, что arr.concat(3,4,5)

alert( newArr ); // 1,2,3,4,5
```

indexOf/lastIndexOf

Эти методы не поддерживаются в IE8-. Для их поддержки подключите библиотеку ES5-shim.

Метод «arr.indexOf(searchElement[, fromIndex])» возвращает номер элемента `searchElement` в массиве `arr` или `-1`, если его нет.

Поиск начинается с номера `fromIndex`, если он указан. Если нет – с начала массива.

Для поиска используется строгое сравнение `===`.

Например:

```
var arr = [1, 0, false];

alert( arr.indexOf(0) ); // 1
alert( arr.indexOf(false) ); // 2
alert( arr.indexOf(null) ); // -1
```

Как вы могли заметить, по синтаксису он полностью аналогичен методу indexOf для строк.

Метод «arr.lastIndexOf(searchElement[, fromIndex])» ищет справа-налево: с конца массива или с номера `fromIndex`, если он указан.

Методы `indexOf/lastIndexOf` осуществляют поиск перебором

Если нужно проверить, существует ли значение в массиве – его нужно перебрать. Только так. Внутренняя реализация `indexOf/lastIndexOf` осуществляет полный перебор, аналогичный циклу `for` по массиву. Чем длиннее массив, тем дольше он будет работать.

Коллекция уникальных элементов

Рассмотрим задачу – есть коллекция строк, и нужно быстро проверять: есть ли в ней какой-то элемент. Массив для этого не подходит из-за медленного `indexOf`. Но подходит объект! Доступ к свойству объекта осуществляется очень быстро, так что можно сделать все элементы ключами объекта и проверять, есть ли уже такой ключ.

Например, организуем такую проверку для коллекции строк `"div"`, `"a"` и `"form"`:

```
var store = {}; // объект для коллекции
var items = ["div", "a", "form"];
for (var i = 0; i < items.length; i++) {
  var key = items[i]; // для каждого элемента создаём свойство
  store[key] = true; // значение здесь не важно
}
```

Теперь для проверки, есть ли ключ `key`, достаточно выполнить `if (store[key])`. Если есть – можно использовать значение, если нет – добавить.

Такое решение работает только со строками, но применимо к любым элементам, для которых можно вычислить строковый «уникальный ключ».

Object.keys(obj)

Ранее мы говорили о том, что свойства объекта можно перебрать в цикле `for..in`.

Если мы хотим работать с ними в виде массива, то к нашим услугам – замечательный метод Object.keys(obj). Он поддерживается везде, кроме IE8-:

```
var user = {
  name: "Петя",
  age: 30
}
var keys = Object.keys(user);
alert( keys ); // name, age
```

Логические операторы

Для операций над логическими значениями в JavaScript есть `||` (ИЛИ), `&&` (И) и `!` (НЕ).

Хоть они и называются *«логическими»*, но в JavaScript могут применяться к значениям любого типа и возвращают также значения любого типа.

|| (ИЛИ)

Оператор ИЛИ выглядит как двойной символ вертикальной черты.

Логическое ИЛИ в классическом программировании работает следующим образом: "если *хотя бы один*из аргументов `true`, то возвращает `true`, иначе – `false`". В JavaScript, как мы увидим далее, это не совсем так, но для начала рассмотрим только логические значения.

Итак, оператор `||` вычисляет операнды слева направо до первого «истинного» и возвращает его, а если все ложные – то последнее значение.

Иначе можно сказать, что "`||` запинается на правде".

&& (И)

Оператор И пишется как два амперсанда `&&`:

В классическом программировании И возвращает `true`, если оба аргумента истинны, а иначе – `false.`
К И применим тот же принцип «короткого цикла вычислений», но немного по-другому, чем к ИЛИ.

Если левый аргумент – `false`, оператор И возвращает его и заканчивает вычисления. Иначе – вычисляет и возвращает правый аргумент.

Итак, оператор `&&` вычисляет операнды слева направо до первого «ложного» и возвращает его, а если все истинные – то последнее значение.

Иначе можно сказать, что "`&&` запинается на лжи".

Приоритет у `&&` больше, чем у `||`
Приоритет оператора И `&&` больше, чем ИЛИ `||`, так что он выполняется раньше.

Не используйте `&&` вместо `if`
Оператор `&&` в простых случаях можно использовать вместо `if`

! (НЕ)

Оператор НЕ – самый простой. Он получает один аргумент

Действия `!`:

1. Сначала приводит аргумент к логическому типу `true/false`.
2. Затем возвращает противоположное значение.

Циклы

Цикл while

Цикл `while` имеет вид:

```
while (условие) {
  // код, тело цикла
}
```

Пока `условие` верно – выполняется код из тела цикла.

Например, цикл ниже выводит `i` пока `i < 3`:

```
var i = 0;
while (i < 3) {
  alert( i );
  i++;
}
```

Повторение цикла по-научному называется *«итерация»*. Цикл в примере выше совершает три итерации.

Если бы `i++` в коде выше не было, то цикл выполнялся бы (в теории) вечно. На практике, браузер выведет сообщение о «зависшем» скрипте и посетитель его остановит.

Условие в скобках интерпретируется как логическое значение, поэтому вместо `while (i!=0)`обычно пишут `while (i)`:

Цикл do...while

Проверку условия можно поставить *под* телом цикла, используя специальный синтаксис `do..while`:

```
do {
  // тело цикла
} while (условие);
```

Цикл, описанный, таким образом, сначала выполняет тело, а затем проверяет условие.

Например:

```
var i = 0;
do {
  alert( i );
  i++;
} while (i < 3);
```

Синтаксис `do..while` редко используется, т.к.
обычный `while` нагляднее – в нём не приходится искать глазами условие и ломать голову, почему оно проверяется именно в конце.

Цикл for

Чаще всего применяется цикл `for`. Выглядит он так:

```
for (начало; условие; шаг) {
  // ... тело цикла ...
}
```

Пример цикла, который
выполняет `alert(i)` для `i` от `0` до `2` включительно (до `3`):

```
var i;

for (i = 0; i < 3; i++) {
  alert( i );
}
```

Здесь:

- **Начало:** `i=0`.
- **Условие:** `i<3`.
- **Шаг:** `i++`.
- **Тело:** `alert(i)`, т.е. код внутри фигурных скобок (они не обязательны, если только одна операция)

Цикл выполняется так:

1. Начало: `i=0` выполняется один-единственный раз, при заходе в цикл.
2. Условие: `i<3` проверяется перед каждой итерацией и при входе в цикл, если оно нарушено, то происходит выход.
3. Тело: `alert(i)`.
4. Шаг: `i++` выполняется после *тела* на каждой итерации, но перед проверкой условия.
5. Идти на шаг 2.

Иными словами, поток выполнения: `начало` →
(если `условие` → `тело` → `шаг`) → (если `условие` → `тело`→ `шаг`) → ... и так далее, пока верно `условие`.

На заметку:
В цикле также можно определить переменную:

```
    for (var i = 0; i < 3; i++) {
  alert(i); // 0, 1, 2
}
```

Эта переменная будет видна и за границами цикла, в частности, после окончания цикла `i` станет равно `3`.

Пропуск частей for

Любая часть `for` может быть пропущена.

Например, можно убрать `начало`. Цикл в примере ниже полностью идентичен приведённому выше:

```
var i = 0;

for (; i < 3; i++) {
  alert( i ); // 0, 1, 2
}
```

Можно убрать и `шаг`:

```
var i = 0;

for (; i < 3;) {
  alert( i );
  i++;
  // цикл превратился в аналог while (i<3)
}
```

А можно и вообще убрать всё, получив бесконечный цикл:

```
for (;;) {
  // будет выполняться вечно
}
```

При этом сами точки с запятой `;` обязательно должны присутствовать, иначе будет ошибка синтаксиса.

Прерывание цикла: break

Выйти из цикла можно не только при проверке условия но и, вообще, в любой момент. Эту возможность обеспечивает директива `break`.

Например, следующий код подсчитывает сумму вводимых чисел до тех пор, пока посетитель их вводит, а затем – выдаёт:

```
var sum = 0;

while (true) {

  var value = +prompt("Введите число", '');

  if (!value) break; // (*)

  sum += value;

}
alert( 'Сумма: ' + sum );
```

Директива `break` в строке `(*)`, если посетитель ничего не ввёл, полностью прекращает выполнение цикла и передаёт управление на строку за его телом, то есть на `alert`.

Вообще, сочетание «бесконечный цикл + break» – отличная штука для тех ситуаций, когда условие, по которому нужно прерваться, находится не в начале-конце цикла, а посередине.

Следующая итерация: continue

Директива `continue` прекращает выполнение *текущей итерации* цикла.

Она – в некотором роде «младшая сестра» директивы `break`: прерывает не весь цикл, а только текущее выполнение его тела, как будто оно закончилось.

Её используют, если понятно, что на текущем повторе цикла делать больше нечего.

Например, цикл ниже использует `continue`, чтобы не выводить чётные значения:

```
for (var i = 0; i < 10; i++) {

  if (i % 2 == 0) continue;

  alert(i);
}
```

Для чётных `i` срабатывает `continue`, выполнение тела прекращается и управление передаётся на следующий проход `for`.

Директива `continue` позволяет обойтись без скобок
Цикл, который обрабатывает только нечётные значения, мог бы выглядеть так:

```
for (var i = 0; i < 10; i++) {

  if (i % 2) {
    alert( i );
  }
}
```

С технической точки зрения он полностью идентичен. Действительно, вместо `continue` можно просто завернуть действия в блок `if`. Однако, мы получили дополнительный уровень вложенности фигурных скобок. Если код внутри `if` более длинный, то это ухудшает читаемость, в отличие от варианта с `continue`.

Нельзя использовать break/continue справа от оператора „?“
Обычно мы можем заменить `if` на оператор вопросительный знак `'?'`.

То есть, запись:

```
if (условие) {
  a();
} else {
  b();
```

```
}
```

...Аналогична записи:

```
условие ? a() : b();
```

В обоих случаях в зависимости от условия выполняется либо `a()` либо `b()`.

Но разница состоит в том, что оператор вопросительный знак `'?'`, использованный во второй записи, возвращает значение.

Синтаксические конструкции, которые не возвращают значений, нельзя использовать в операторе `'?'`.

К таким относятся большинство конструкций и, в частности, `break/continue`.

Поэтому такой код приведёт к ошибке:

```
   (i > 5) ? alert(i) : continue;
```

Впрочем, как уже говорилось ранее, оператор вопросительный знак `'?'` не стоит использовать таким образом. Это – всего лишь ещё одна причина, почему для проверки условия предпочтителен `if`.

Метки для break/continue

Бывает нужно выйти одновременно из нескольких уровней цикла.

Например, внутри цикла по `i` находится цикл по `j`, и при выполнении некоторого условия мы бы хотели выйти из обоих циклов сразу:

```
outer: for (var i = 0; i < 3; i++) {

  for (var j = 0; j < 3; j++) {

    var input = prompt('Значение в координатах '+i+','+j, '');

    // если отмена ввода или пустая строка -
```

```
    // завершить оба цикла
    if (!input) break outer; // (*)

  }
}
alert('Готово!');
```

В коде выше для этого использована *метка*.

Метка имеет вид `"имя:"`, имя должно быть уникальным. Она ставится перед циклом, вот так:

```
outer: for (var i = 0; i < 3; i++) { ... }
```

Можно также выносить её на отдельную строку:

```
outer:
for (var i = 0; i < 3; i++) { ... }
```

Вызов `break outer` ищет ближайший внешний цикл с такой меткой и переходит в его конец.

В примере выше это означает, что будет разорван самый внешний цикл и управление перейдёт на `alert`.

Директива `continue` также может быть использована с меткой, в этом случае управление перепрыгнет на следующую итерацию цикла с меткой.

Итого

JavaScript поддерживает три вида циклов:

- `while` – проверка условия перед каждым выполнением.
- `do..while` – проверка условия после каждого выполнения.
- `for` – проверка условия перед каждым выполнением, а также дополнительные настройки.

Чтобы организовать бесконечный цикл, используют конструкцию `while(true)`. При этом он, как и любой другой цикл, может быть прерван директивой `break`.

Если на данной итерации цикла делать больше ничего не надо, но полностью прекращать цикл не следует – используют директиву `continue`.

Обе этих директивы поддерживают «метки», которые ставятся перед циклом. Метки – единственный способ для `break/continue` повлиять на выполнение внешнего цикла.

Заметим, что метки не позволяют прыгнуть в произвольное место кода, в JavaScript нет такой возможности.

Конструкция switch

Конструкция `switch` заменяет собой сразу несколько `if`.

Она представляет собой более наглядный способ сравнить выражение сразу с несколькими вариантами.

Синтаксис

Выглядит она так:

```
switch(x) {
  case 'value1':  // if (x === 'value1')
    ...
    [break]

  case 'value2':  // if (x === 'value2')
    ...
    [break]

  default:
    ...
    [break]
}
```

- Переменная `x` проверяется на строгое равенство первому значению `value1`, затем второму `value2` и так далее.

- Если соответствие установлено – switch начинает выполняться от соответствующей директивы `case` и далее, *до ближайшего* `break` (или до конца `switch`).
- Если ни один `case` не совпал – выполняется (если есть) вариант `default`.

При этом `case` называют *вариантами* `switch`.

Пример работы

Пример использования `switch` (сработавший код выделен):

```
var a = 2 + 2;

switch (a) {
  case 3:
    alert( 'Маловато' );
    break;
  case 4:
    alert( 'В точку!' );
    break;
  case 5:
    alert( 'Перебор' );
    break;
  default:
    alert( 'Я таких значений не знаю' );
}
```

Здесь оператор `switch` последовательно сравнит `a` со всеми вариантами из `case`.

Сначала `3`, затем – так как нет совпадения – `4`. Совпадение найдено, будет выполнен этот вариант, со строки `alert('В точку!')` и далее, до ближайшего `break`, который прервёт выполнение.

Если `break` нет, то выполнение пойдёт ниже по следующим `case`, при этом остальные проверки игнорируются.

Пример без `break`:

```
var a = 2 + 2;

switch (a) {
  case 3:
```

```
    alert( 'Маловато' );
  case 4:
    alert( 'В точку!' );
  case 5:
    alert( 'Перебор' );
  default:
    alert( 'Я таких значений не знаю' );
}
```

В примере выше последовательно выполнятся три `alert`:

```
alert( 'В точку!' );
alert( 'Перебор' );
alert( 'Я таких значений не знаю' );
```

В `case` могут быть любые выражения, в том числе включающие в себя переменные и функции.

Например:

```
var a = 1;
var b = 0;

switch (a) {
  case b + 1:
    alert( 1 );
    break;

  default:
    alert('нет-нет, выполнится вариант выше')
}
```

Группировка case

Несколько значений case можно группировать.

В примере ниже `case 3` и `case 5` выполняют один и тот же код:

```
var a = 2+2;

switch (a) {
  case 4:
    alert('Верно!');
    break;
```

```
  case 3:                     // (*)
  case 5:                     // (**)
    alert('Неверно!');
    alert('Немного ошиблись, бывает.');
    break;

  default:
    alert('Странный результат, очень странный');
}
```

При `case 3` выполнение идёт со строки `(*)`, при `case 5` – со строки `(**)`.

Тип имеет значение

Следующий пример принимает значение от посетителя.

```
var arg = prompt("Введите arg?")
switch (arg) {
  case '0':
  case '1':
    alert( 'Один или ноль' );

  case '2':
    alert( 'Два' );
    break;

  case 3:
    alert( 'Никогда не выполнится' );

  default:
    alert('Неизвестное значение: ' + arg)
}
```

Что оно выведет при вводе числа 0? Числа 1? 2? 3?

Подумайте, выпишите свои ответы, исходя из текущего понимания работы `switch` и *потом* читайте дальше...

- При вводе `0` выполнится первый `alert`, далее выполнение продолжится вниз до первого `break` и выведет второй `alert('Два')`. Итого, два вывода `alert`.
- При вводе `1` произойдёт то же самое.

- При вводе `2`, `switch` перейдет к `case '2'`, и сработает единственный `alert('Два')`.
- **При вводе `3`, `switch` перейдет на `default`.** Это потому, что `prompt` возвращает строку `'3'`, а не число. Типы разные. Оператор `switch` предполагает строгое равенство `===`, так что совпадения не будет.

Функции

Зачастую нам надо повторять одно и то же действие во многих частях программы.

Например, красиво вывести сообщение необходимо при приветствии посетителя, при выходе посетителя с сайта, ещё где-нибудь.

Чтобы не повторять один и тот же код во многих местах, придуманы функции. Функции являются основными «строительными блоками» программы.

Примеры встроенных функций вы уже видели – это `alert(message)`, `prompt(message, default)` и `confirm(question)`. Но можно создавать и свои.

Объявление

Пример объявления функции:

```
function showMessage() {
  alert( 'Привет всем присутствующим!' );
}
```

Вначале идет ключевое слово `function`, после него *имя функции*, затем *список параметров* в скобках (в примере выше он пустой) и *тело функции* – код, который выполняется при её вызове.

Объявленная функция доступна по имени, например:

```
function showMessage() {
```

```
  alert( 'Привет всем присутствующим!' );
}

showMessage();
showMessage();
```

Этот код выведет сообщение два раза. Уже здесь видна **главная цель создания функций: избавление от дублирования кода**.

Если понадобится поменять сообщение или способ его вывода – достаточно изменить его в одном месте: в функции, которая его выводит.

Локальные переменные

Функция может содержать *локальные* переменные, объявленные через `var`. Такие переменные видны только внутри функции:

```
function showMessage() {
  var message = 'Привет, я - Вася!'; // локальная переменная

  alert( message );
}

showMessage(); // 'Привет, я - Вася!'

alert( message ); // <-- будет ошибка, т.к. переменная видна только внутри
```

Блоки `if/else`, `switch`, `for`, `while`, `do..while` не влияют на область видимости переменных.

При объявлении переменной в таких блоках, она всё равно будет видна во всей функции.

Например:

```
function count() {
  // переменные i,j не будут уничтожены по окончании цикла
  for (var i = 0; i < 3; i++) {
    var j = i * 2;
  }
```

```
  alert( i ); // i=3, последнее значение i, при нём цикл перестал работать
  alert( j ); // j=4, последнее значение j, которое вычислил цикл
}
```

Неважно, где именно в функции и сколько раз объявляется переменная. Любое объявление срабатывает один раз и распространяется на всю функцию.

Объявления переменных в примере выше можно передвинуть вверх, это ни на что не повлияет:

```
function count() {
  var i, j; // передвинули объявления var в начало
  for (i = 0; i < 3; i++) {
    j = i * 2;
  }

  alert( i ); // i=3
  alert( j ); // j=4
}
```

Внешние переменные

Функция может обратиться ко внешней переменной, например:

```
var userName = 'Вася';

function showMessage() {
  var message = 'Привет, я ' + userName;
  alert(message);
}

showMessage(); // Привет, я Вася
```

Доступ возможен не только на чтение, но и на запись. При этом, так как переменная внешняя, то изменения будут видны и снаружи функции:

```
var userName = 'Вася';

function showMessage() {
  userName = 'Петя'; // (1) присвоение во внешнюю переменную
```

```
  var message = 'Привет, я ' + userName;
  alert( message );
}

showMessage();

alert( userName ); // Петя, значение внешней переменной изменено функцией
```

Конечно, если бы внутри функции, в строке `(1)`, была бы объявлена своя локальная переменная `var userName`, то все обращения использовали бы её, и внешняя переменная осталась бы неизменной.

Переменные, объявленные на уровне всего скрипта, называют *«глобальными переменными»*.

В примере выше переменная `userName` – глобальная.

Делайте глобальными только те переменные, которые действительно имеют общее значение для вашего проекта, а нужные для решения конкретной задачи – пусть будут локальными в соответствующей функции.

Внимание: неявное объявление глобальных переменных!

В старом стандарте JavaScript существовала возможность неявного объявления переменных присвоением значения.

Например:

```
function showMessage() {
  message = 'Привет'; // без var!
}

showMessage();

alert( message ); // Привет
```

В коде выше переменная `message` нигде не объявлена, а сразу присваивается. Скорее всего, программист просто забыл поставить `var`.

При `use strict` такой код привёл бы к ошибке, но без него переменная будет создана автоматически, причём в примере выше она создаётся не в функции, а на уровне всего скрипта.

Избегайте этого.

Здесь опасность даже не в автоматическом создании переменной, а в том, что глобальные переменные должны использоваться тогда, когда действительно нужны «общескриптовые» параметры.

Забыли `var` в одном месте, потом в другом – в результате одна функция неожиданно поменяла глобальную переменную, которую использует другая. И поди разберись, кто и когда её поменял, не самая приятная ошибка для отладки.

В будущем, когда мы лучше познакомимся с основами JavaScript, в главе Замыкания, функции изнутри, мы более детально рассмотрим внутренние механизмы работы переменных и функций.

Параметры

При вызове функции ей можно передать данные, которые та использует по своему усмотрению.

Например, этот код выводит два сообщения:

```
function showMessage(from, text) { // параметры from, text

  from = "** " + from + " **"; // здесь может быть сложный код оформления

  alert(from + ': ' + text);
}

showMessage('Маша', 'Привет!');
showMessage('Маша', 'Как дела?');
```

Параметры копируются в локальные переменные функции.

Например, в коде ниже есть внешняя переменная `from`, значение которой при запуске функции копируется в параметр функции с тем же именем. Далее функция работает уже с параметром:

```
function showMessage(from, text) {
  from = '**' + from + '**'; // меняем локальную переменную from
  alert( from + ': ' + text );
}

var from = "Маша";

showMessage(from, "Привет");

alert( from ); // старое значение from без изменений, в функции была изменена копия
```

Аргументы по умолчанию

Функцию можно вызвать с любым количеством аргументов.

Если параметр не передан при вызове – он считается равным `undefined`.

Например, функцию показа сообщения `showMessage(from, text)` можно вызвать с одним аргументом:

```
showMessage("Маша");
```

При этом можно проверить, и если параметр не передан – присвоить ему значение «по умолчанию»:

```
function showMessage(from, text) {
  if (text === undefined) {
    text = 'текст не передан';
  }

  alert( from + ": " + text );
}

showMessage("Маша", "Привет!"); // Маша: Привет!
showMessage("Маша"); // Маша: текст не передан
```

При объявлении функции необязательные аргументы, как правило, располагают в конце списка.

Для указания значения «по умолчанию», то есть, такого, которое используется, если аргумент не указан, используется два способа:

1. Можно проверить, равен ли аргумент `undefined`, и если да – то записать в него значение по умолчанию. Этот способ продемонстрирован в примере выше.
2. Использовать оператор `||`:

```
function showMessage(from, text) {
  text = text || 'текст не передан';

  ...
}
```

Второй способ считает, что аргумент отсутствует, если передана пустая строка, `0`, или вообще любое значение, которое в логическом контексте является `false`.

Если аргументов передано больше, чем надо, например `showMessage("Маша", "привет", 1, 2, 3)`, то ошибки не будет. Но, чтобы получить такие «лишние» аргументы, нужно будет прочитать их из специального объекта `arguments`, который мы рассмотрим в главе Псевдомассив аргументов "arguments".

Возврат значения

Функция может возвратить результат, который будет передан в вызвавший её код.

Например, создадим функцию `calcD`, которая будет возвращать дискриминант квадратного уравнения по формуле $b^2 - 4ac$:

```
function calcD(a, b, c) {
   return b*b - 4*a*c;
}
var test = calcD(-4, 2, 1);
alert(test); // 20
```

Для возврата значения используется директива `return`.

Она может находиться в любом месте функции. Как только до неё доходит управление – функция завершается и значение передается обратно.

Вызовов `return` может быть и несколько, например:

```
function checkAge(age) {
  if (age > 18) {
    return true;
  } else {
    return confirm('Родители разрешили?');
  }
}

var age = prompt('Ваш возраст?');

if (checkAge(age)) {
  alert( 'Доступ разрешен' );
} else {
  alert( 'В доступе отказано' );
}
```

Директива `return` может также использоваться без значения, чтобы прекратить выполнение и выйти из функции.

Например:

```
function showMovie(age) {
  if (!checkAge(age)) {
    return;
  }

  alert( "Фильм не для всех" ); // (*)
  // ...
}
```

В коде выше, если сработал `if`, то строка `(*)` и весь код под ней никогда не выполнится, так как `return`завершает выполнение функции.

Значение функции без `return` и с пустым `return`

В случае, когда функция не вернула значение или `return` был без аргументов, считается что она вернула `undefined`:

```
function doNothing() { /* пусто */ }

alert( doNothing() ); // undefined
```

Обратите внимание, никакой ошибки нет. Просто возвращается `undefined`.

Ещё пример, на этот раз с `return` без аргумента:

```
function doNothing() {
  return;
}

alert( doNothing() === undefined ); // true
```

Выбор имени функции

Имя функции следует тем же правилам, что и имя переменной. Основное отличие – оно должно быть глаголом, т.к. функция – это действие.

Как правило, используются глагольные префиксы, обозначающие общий характер действия, после которых следует уточнение.

Функции, которые начинаются с `"show"` – что-то показывают:

```
showMessage(..)     // префикс show, "показать" сообщение
```

Функции, начинающиеся с `"get"` – получают, и т.п.:

```
getAge(..)          // get, "получает" возраст
calcD(..)           // calc, "вычисляет" дискриминант
createForm(..)      // create, "создает" форму
checkPermission(..) // check, "проверяет" разрешение, возвращает true/false
```

Это очень удобно, поскольку взглянув на функцию – мы уже примерно представляем, что она делает, даже если функцию написал совсем другой человек, а в отдельных случаях – и какого вида значение она возвращает.

Одна функция – одно действие

Функция должна делать только то, что явно подразумевается её названием. И это должно быть одно действие.

Если оно сложное и подразумевает поддействия – может быть имеет смысл выделить их в отдельные функции? Зачастую это имеет смысл, чтобы лучше структурировать код.

...Но самое главное – в функции не должно быть ничего, кроме самого действия и поддействий, неразрывно связанных с ним.

Например, функция проверки данных (скажем, `"validate"`) не должна показывать сообщение об ошибке. Её действие – проверить.

Сверхкороткие имена функций

Имена функций, которые используются *очень часто*, иногда делают сверхкороткими.

Например, во фреймворке jQuery есть функция `$`, во фреймворке Prototype – функция `$$`, а в библиотеке LoDash очень активно используется функция с названием из одного символа подчеркивания `_`.

Итого

Объявление функции имеет вид:

```
function имя(параметры, через, запятую) {
  код функции
}
```

- Передаваемые значения копируются в параметры функции и становятся локальными переменными.
- Параметры функции копируются в её локальные переменные.
- Можно объявить новые локальные переменные при помощи `var`.
- Значение возвращается оператором `return ...`.

- Вызов `return` тут же прекращает функцию.
- Если `return;` вызван без значения, или функция завершилась без `return`, то её результат равен `undefined`.

  При обращении к необъявленной переменной функция будет искать внешнюю переменную с таким именем, но лучше, если функция использует только локальные переменные:

- Это делает очевидным общий поток выполнения – что передаётся в функцию и какой получаем результат.
- Это предотвращает возможные конфликты доступа, когда две функции, возможно написанные в разное время или разными людьми, неожиданно друг для друга меняют одну и ту же внешнюю переменную.

  Именование функций:

- Имя функции должно понятно и чётко отражать, что она делает. Увидев её вызов в коде, вы должны тут же понимать, что она делает.
- Функция – это действие, поэтому для имён функций, как правило, используются глаголы.

Дата и Время

Для работы с датой и временем в JavaScript используются объекты Date.

Создание

Для создания нового объекта типа `Date` используется один из синтаксисов:

`new Date()`

Создает объект `Date` с текущей датой и временем:

```
var now = new Date();
alert( now );
```

`new Date(milliseconds)`

Создает объект `Date`, значение которого равно количеству миллисекунд (1/1000 секунды), прошедших с 1 января 1970 года GMT+0.

```
// 24 часа после 01.01.1970 GMT+0
var Jan02_1970 = new Date(3600 * 24 * 1000);
alert( Jan02_1970 );
```

`new Date(datestring)`

Если единственный аргумент – строка, используется вызов `Date.parse` (см. далее) для чтения даты из неё.

`new Date(year, month, date, hours, minutes, seconds, ms)`

Дату можно создать, используя компоненты в местной временной зоне. Для этого формата обязательны только первые два аргумента. Отсутствующие параметры, начиная с `hours` считаются равными нулю, а `date` – единице.

Заметим:

- Год `year` должен быть из 4 цифр.

- Отсчет месяцев `month` начинается с нуля 0.

  Например:

```
new Date(2011, 0, 1, 0, 0, 0, 0); // // 1 января 2011, 00:00:00
new Date(2011, 0, 1); // то же самое, часы/секунды по умолчанию равны 0
```

  Дата задана с точностью до миллисекунд:

```
var date = new Date(2011, 0, 1, 2, 3, 4, 567);
alert( date ); // 1.01.2011, 02:03:04.567
```

Получение компонентов даты

Для доступа к компонентам даты-времени объекта `Date` используются следующие методы:

`getFullYear()`

Получить год (из 4 цифр)

`getMonth()`

Получить месяц, **от 0 до 11**.

`getDate()`

Получить число месяца, от 1 до 31.

`getHours(), getMinutes(), getSeconds(), getMilliseconds()`

Получить соответствующие компоненты.

Не `getYear()`, а `getFullYear()`
Некоторые браузеры реализуют нестандартный метод `getYear()`. Где-то он возвращает только две цифры из года, где-то четыре. Так или иначе, этот метод отсутствует в стандарте JavaScript. Не используйте его. Для получения года есть `getFullYear()`.

Дополнительно можно получить день недели:

`getDay()`

Получить номер дня в неделе. Неделя в JavaScript начинается с воскресенья, так что результат будет числом **от 0(воскресенье) до 6(суббота)**.

Все методы, указанные выше, возвращают результат для местной временной зоны.

Существуют также UTC-варианты этих методов, возвращающие день, месяц, год и т.п. для зоны GMT+0 (UTC): `getUTCFullYear()`, `getUTCMonth()`, `getUTCDay()`. То есть, сразу после `"get"` вставляется `"UTC"`.

Если ваше локальное время сдвинуто относительно UTC, то следующий код покажет разные часы:

```
// текущая дата
var date = new Date();

// час в текущей временной зоне
alert( date.getHours() );

// сколько сейчас времени в Лондоне?
// час в зоне GMT+0
alert( date.getUTCHours() );
```

Кроме описанных выше, существуют два специальных метода без UTC-варианта:

`getTime()`

Возвращает число миллисекунд, прошедших с 1 января 1970 года GMT+0, то есть того же вида, который используется в конструкторе `new Date(milliseconds)`.

`getTimezoneOffset()`

Возвращает разницу между местным и UTC-временем, в минутах.

```
alert( new Date().getTimezoneOffset() ); // Для GMT-1 выведет 60
```

Установка компонентов даты

Следующие методы позволяют устанавливать компоненты даты и времени:

- `setFullYear(year [, month, date])`
- `setMonth(month [, date])`
- `setDate(date)`
- `setHours(hour [, min, sec, ms])`
- `setMinutes(min [, sec, ms])`
- `setSeconds(sec [, ms])`
- `setMilliseconds(ms)`
- `setTime(milliseconds)` (устанавливает всю дату по миллисекундам с 01.01.1970 UTC)

Все они, кроме `setTime()`, обладают также UTC-вариантом, например: `setUTCHours()`.

Как видно, некоторые методы могут устанавливать несколько компонентов даты одновременно, в частности, `setHours`. При этом если какая-то компонента не указана, она не меняется. Например:

```
var today = new Date;

today.setHours(0);
alert( today ); // сегодня, но час изменён на 0

today.setHours(0, 0, 0, 0);
alert( today ); // сегодня, ровно 00:00:00.
```

Автоисправление даты

Автоисправление – очень удобное свойство объектов `Date`. Оно заключается в том, что можно устанавливать заведомо некорректные компоненты (например 32 января), а объект сам себя поправит.

```
  var d = new Date(2013, 0, 32); // 32 января 2013 ?!?
alert(d); // ... это 1 февраля 2013!
```

Неправильные компоненты даты автоматически распределяются по остальным.

Например, нужно увеличить на 2 дня дату «28 февраля 2011». Может быть так, что это будет 2 марта, а может быть и 1 марта, если год високосный. Но нам обо всем этом думать не нужно. Просто прибавляем два дня. Остальное сделает `Date`:

```
var d = new Date(2011, 1, 28);
d.setDate(d.getDate() + 2);

alert( d ); // 2 марта, 2011
```

Также это используют для получения даты, отдаленной от имеющейся на нужный промежуток времени. Например, получим дату на 70 секунд большую текущей:

```
var d = new Date();
d.setSeconds(d.getSeconds() + 70);

alert( d ); // выведет корректную дату
```

Можно установить и нулевые, и даже отрицательные компоненты. Например:

```
var d = new Date;

d.setDate(1); // поставить первое число месяца
alert( d );

d.setDate(0); // нулевого числа нет, будет последнее число
предыдущего месяца
alert( d );
var d = new Date;

d.setDate(-1); // предпоследнее число предыдущего месяца
alert( d );
```

Преобразование к числу, разность дат

Когда объект `Date` используется в числовом контексте, он преобразуется в количество миллисекунд:

```
alert(+new Date) // +date то же самое, что: +date.valueOf()
```

Важный побочный эффект: даты можно вычитать, результат вычитания объектов `Date` – их временная разница, в миллисекундах.

Это используют для измерения времени:

```
var start = new Date; // засекли время

// что-то сделать
for (var i = 0; i < 100000; i++) {
  var doSomething = i * i * i;
}

var end = new Date; // конец измерения

alert( "Цикл занял " + (end - start) + " ms" );
```

Бенчмаркинг

Допустим, у нас есть несколько вариантов решения задачи, каждый описан функцией.

Как узнать, какой быстрее?

Для примера возьмем две функции, которые бегают по массиву:

```
function walkIn(arr) {
  for (var key in arr) arr[key]++
}

function walkLength(arr) {
  for (var i = 0; i < arr.length; i++) arr[i]++;
}
```

Чтобы померить, какая из них быстрее, нельзя запустить один раз `walkIn`, один раз `walkLength` и замерить разницу. Одноразовый запуск ненадежен, любая мини-помеха исказит результат.

Для правильного бенчмаркинга функция запускается много раз, чтобы сам тест занял существенное время. Это сведет влияние

помех к минимуму. Сложную функцию можно запускать 100 раз, простую – 1000 раз...

Померяем, какая из функций быстрее:

```
var arr = [];
for (var i = 0; i < 1000; i++) arr[i] = 0;

function walkIn(arr) {
  for (var key in arr) arr[key]++;
}

function walkLength(arr) {
  for (var i = 0; i < arr.length; i++) arr[i]++;
}

function bench(f) {
  var date = new Date();
  for (var i = 0; i < 10000; i++) f(arr);
  return new Date() - date;
}

alert( 'Время walkIn: ' + bench(walkIn) + 'мс' );
alert( 'Время walkLength: ' + bench(walkLength) + 'мс' );
```

Теперь представим себе, что во время первого бенчмаркинга `bench(walkIn)` компьютер что-то делал параллельно важное (вдруг) и это занимало ресурсы, а во время второго – перестал. Реальная ситуация? Конечно реальна, особенно на современных ОС, где много процессов одновременно.

Гораздо более надёжные результаты можно получить, если весь пакет тестов прогнать много раз.

```
var arr = [];
for (var i = 0; i < 1000; i++) arr[i] = 0;

function walkIn(arr) {
  for (var key in arr) arr[key]++;
}

function walkLength(arr) {
  for (var i = 0; i < arr.length; i++) arr[i]++;
}
```

```
function bench(f) {
  var date = new Date();
  for (var i = 0; i < 1000; i++) f(arr);
  return new Date() - date;
}

// bench для каждого теста запустим много раз, чередуя
var timeIn = 0,
  timeLength = 0;
for (var i = 0; i < 100; i++) {
  timeIn += bench(walkIn);
  timeLength += bench(walkLength);
}

alert( 'Время walkIn: ' + timeIn + 'мс' );
alert( 'Время walkLength: ' + timeLength + 'мс' );
```

Более точное время с `performance.now()`

В современных браузерах (кроме IE9-)
вызов performance.now() возвращает количество миллисекунд, прошедшее с начала загрузки страницы. Причём именно с самого начала, до того, как загрузился HTML-файл, если точнее – с момента выгрузки предыдущей страницы из памяти.

Так что это время включает в себя всё, включая начальное обращение к серверу.

Его можно посмотреть в любом месте страницы, даже в `<head>`, чтобы узнать, сколько времени потребовалось браузеру, чтобы до него добраться, включая загрузку HTML.

Возвращаемое значение измеряется в миллисекундах, но дополнительно имеет точность 3 знака после запятой (до миллионных долей секунды!), поэтому можно использовать его и для более точного бенчмаркинга в том числе.

`console.time(метка)` и `console.timeEnd(метка)`

Для измерения с одновременным выводом результатов в консоли есть методы:

- `console.time(метка)` – включить внутренний хронометр браузера с меткой.
- `console.timeEnd(метка)` – выключить внутренний хронометр браузера с меткой и вывести результат.

Параметр `"метка"` используется для идентификации таймера, чтобы можно было делать много замеров одновременно и даже вкладывать измерения друг в друга.

В коде ниже таймеры `walkIn`, `walkLength` – конкретные тесты, а таймер «All Benchmarks» – время «на всё про всё»:

```
var arr = [];
for (var i = 0; i < 1000; i++) arr[i] = 0;

function walkIn(arr) {
  for (var key in arr) arr[key]++;
}

function walkLength(arr) {
  for (var i = 0; i < arr.length; i++) arr[i]++;
}

function bench(f) {
  for (var i = 0; i < 10000; i++) f(arr);
}

console.time("All Benchmarks");

console.time("walkIn");
bench(walkIn);
console.timeEnd("walkIn");

console.time("walkLength");
bench(walkLength);
console.timeEnd("walkLength");

console.timeEnd("All Benchmarks");
```

При запуске этого примера нужно открыть консоль, иначе вы ничего не увидите.

Внимание, оптимизатор!

Современные интерпретаторы JavaScript делают массу оптимизаций, например:

1. Автоматически выносят инвариант, то есть постоянное в цикле значение типа `arr.length`, за пределы цикла.
2. Стараются понять, значения какого типа хранит данная переменная или массив, какую структуру имеет объект и, исходя из этого, оптимизировать внутренние алгоритмы.
3. Выполняют простейшие операции, например сложение явно заданных чисел и строк, на этапе компиляции.
4. Могут обнаружить, что некий код, например присваивание к неиспользуемой локальной переменной, ни на что не влияет и вообще исключить его из выполнения, хотя делают это редко.

Эти оптимизации могут влиять на результаты тестов, поэтому измерять скорость базовых операций JavaScript («проводить микробенчмаркинг») до того, как вы изучите внутренности JavaScript-интерпретаторов и поймёте, что они реально делают на таком коде, не рекомендуется.

Форматирование и вывод дат

Во всех браузерах, кроме IE10-, поддерживается новый стандарт Ecma 402, который добавляет специальные методы для форматирования дат.

Это делается вызовом `date.toLocaleString(локаль, опции)`, в котором можно задать много настроек. Он позволяет указать, какие параметры даты нужно вывести, и ряд настроек вывода, после чего интерпретатор сам сформирует строку.

Пример с почти всеми параметрами даты и русским, затем английским (США) форматированием:

```
var date = new Date(2014, 11, 31, 12, 30, 0);

var options = {
  era: 'long',
  year: 'numeric',
```

```
  month: 'long',
  day: 'numeric',
  weekday: 'long',
  timezone: 'UTC',
  hour: 'numeric',
  minute: 'numeric',
  second: 'numeric'
};

alert( date.toLocaleString("ru", options) ); // среда, 31 декабря
2014 г. н.э. 12:30:00
alert( date.toLocaleString("en-US", options) ); // Wednesday,
December 31, 2014 Anno Domini 12:30:00 PM
```

Вы сможете подробно узнать о них в статье Intl: интернационализация в JavaScript, которая посвящена этому стандарту.

Методы вывода без локализации:

`toString()`, `toDateString()`, `toTimeString()` Возвращают стандартное строчное представление, не заданное жёстко в стандарте, а зависящее от браузера. Единственное требование к нему – читаемость человеком. Метод `toString` возвращает дату целиком, `toDateString()` и `toTimeString()` – только дату и время соответственно.

```
var d = new Date();

alert( d.toString() ); // вывод, похожий на 'Wed Jan 26 2011
16:40:50 GMT+0300'
```

`toUTCString()` То же самое, что `toString()`, но дата в зоне UTC.

`toISOString()` Возвращает дату в формате ISO Детали формата будут далее. Поддерживается современными браузерами, не поддерживается IE8-.

```
var d = new Date();

alert( d.toISOString() ); // вывод, похожий на '2011-01-
26T13:51:50.417Z'
```

Если хочется иметь большую гибкость и кросс-браузерность, то также можно воспользоваться специальной библиотекой, например Moment.JS или написать свою функцию форматирования.

Разбор строки, Date.parse

Все современные браузеры, включая IE9+, понимают даты в упрощённом формате ISO 8601 Extended.

Этот формат выглядит так: `YYYY-MM-DDTHH:mm:ss.sssZ`, где:

- `YYYY-MM-DD` – дата в формате год-месяц-день.
- Обычный символ `T` используется как разделитель.
- `HH:mm:ss.sss` – время: часы-минуты-секунды-миллисекунды.
- Часть `'Z'` обозначает временную зону – в формате `+-hh:mm`, либо символ `Z`, обозначающий UTC. По стандарту её можно не указывать, тогда UTC, но в Safari с этим ошибка, так что лучше указывать всегда.

Также возможны укороченные варианты, например `YYYY-MM-DD` или `YYYY-MM` или даже только `YYYY`.

Метод `Date.parse(str)` разбирает строку `str` в таком формате и возвращает соответствующее ей количество миллисекунд. Если это невозможно, `Date.parse` возвращает `NaN`.

Например:

```
var msUTC = Date.parse('2012-01-26T13:51:50.417Z'); // зона UTC

alert( msUTC ); // 1327571510417 (число миллисекунд)
```

С таймзоной `-07:00 GMT`:

```
var ms = Date.parse('2012-01-26T13:51:50.417-07:00');

alert( ms ); // 1327611110417 (число миллисекунд)
```

Формат дат для IE8-

До появления спецификации ECMAScript 5 формат не был стандартизован, и браузеры, включая IE8-, имели свои собственные форматы дат. Частично, эти форматы пересекаются.

Например, код ниже работает везде, включая старые IE:

```
var ms = Date.parse("January 26, 2011 13:51:50");

alert( ms );
```

Вы также можете почитать о старых форматах IE в документации к методу MSDN Date.parse.

Конечно же, сейчас лучше использовать современный формат. Если же нужна поддержка IE8-, то метод `Date.parse`, как и ряд других современных методов, добавляется библиотекой es5-shim.

Метод Date.now()

Метод `Date.now()` возвращает дату сразу в виде миллисекунд.

Технически, он аналогичен вызову `+new Date()`, но в отличие от него не создаёт промежуточный объект даты, а поэтому – во много раз быстрее.

Его использование особенно рекомендуется там, где производительность при работе с датами критична. Обычно это не на веб-страницах, а, к примеру, в разработке игр на JavaScript.

Итого

- Дата и время представлены в JavaScript одним объектом: Date. Создать «только время» при этом нельзя, оно должно быть с датой. Список методов `Date` вы можете найти в справочнике Date или выше.
- Отсчёт месяцев начинается с нуля.
- Отсчёт дней недели (для `getDay()`) тоже начинается с нуля (и это воскресенье).

- Объект `Date` удобен тем, что автокорректируется. Благодаря этому легко сдвигать даты.
- При преобразовании к числу объект `Date` даёт количество миллисекунд, прошедших с 1 января 1970 UTC. Побочное следствие – даты можно вычитать, результатом будет разница в миллисекундах.
- Для получения текущей даты в миллисекундах лучше использовать `Date.now()`, чтобы не создавать лишний объект `Date` (кроме IE8-)
- Для бенчмаркинга лучше использовать `performance.now()` (кроме IE9-), он в 1000 раз точнее.