Современный учебник JavaScript

© Илья Кантор

Сборка от 23 июля 2013 для чтения с устройств

Внимание, эта сборка может быть устаревшей и не соответствовать текущему тексту.
Актуальный онлайн-учебник, с интерактивными примерами, доступен по адресу http://learn.javascript.ru.

Вопросы по JavaScript можно задавать в комментариях на сайте или на форуме javascript.ru/forum.

Вопросы по сборке, предложения по её улучшению – можно писать мне, по адресу iliakan@javascript.ru .

Глава: Аргументы функций

В файле находится только одна глава учебника. Это сделано в целях уменьшения размера файла, для удобного чтения с устройств.

Содержание

Псевдо-массив arguments

В JavaScript любая функция может быть вызвана с произвольным количеством аргументов.

Например:

```
function go(a,b) {
  alert("a="+a+", b="+b);
}

go(1);     // a=1, b=undefined
go(1,2);   // a=1, b=2
go(1,2,3); // a=1, b=2, третий аргумент не вызовет ошибку
```

В JavaScript нет «перегрузки» функций

В некоторых языках программист может создать две функции с одинаковым именем, но разным набором аргументов, а при вызове интерпретатор сам выберет нужную:

```
function log(a) {
  ...
}

function log(a,b,c) {
  ...
}

log(a); // вызовется первая функция
log(a,b,c); // вызовется вторая функция
```

Это называется «полиморфизмом функций» или «перегрузкой функций». В JavaScript ничего подобного нет.

Может быть только одна функция с именем `log`, которая вызывается с любыми аргументами. А уже внутри она может посмотреть, с чем вызвана и по-разному отработать.

В примере выше второе объявление `log` просто переопределит первое.

Доступ к «лишним» аргументам

Как получить значения аргументов, которых нет в списке параметров?

Доступ к ним осуществляется через «псевдо-массив» arguments .

Он содержит список аргументов по номерам: `arguments[0]`, `arguments[1]`..., а также свойство `length`.

Например, выведем список всех аргументов:

```
function sayHi() {
  for (var i=0; i<arguments.length; i++) {
    alert("Привет, " + arguments[i]);
  }
}

sayHi("Винни", "Пятачок");  // 'Привет, Винни', 'Привет, Пятачок'
```

Все параметры находятся в `arguments`, даже если они есть в списке. Код выше сработал бы также, будь функция объявлена `sayHi(a,b,c)`.

Связь между `arguments` и параметрами

В старом стандарте JavaScript псевдо-массив `arguments` и переменные-параметры ссылаются на одни и те же значения.

В результате изменения `arguments` влияют на параметры и наоборот.

Например:

```
function f(x) {
  arguments[0] = 5; // меняет переменную x
  alert(x); // 5
}

f(1);
```

Наоборот:

```
function f(x) {
```

```
  x = 5;
  alert(arguments[0]); // 5, обновленный x
}

f(1);
```

В современной редакции стандарта это поведение изменено. Аргументы отделены от локальных переменных:

```
function f(x) {
  "use strict"; // для браузеров с поддержкой строгого режима

  arguments[0] = 5;
  alert(x); // не 5, а 1! Переменная "отвязана" от arguments
}

f(1);
```

Если вы не используете строгий режим, то чтобы переменные не менялись «неожиданно», **рекомендуется никогда не изменять `arguments`.**

Частая ошибка новичков — попытка применить методы `Array` к `arguments`. Это невозможно:

```
function sayHi() {
  var a = arguments.shift(); // ошибка! нет такого метода!
}

sayHi(1);
```

Дело в том, что `arguments` — это не массив `Array`.

В действительности, это обычный объект, просто ключи числовые и есть `length`. На этом сходство заканчивается. Никаких особых методов у него нет, и методы массивов он тоже не поддерживает.

Впрочем, никто не мешает сделать обычный массив из `arguments`:

```
var args = [];
for(var i=0; i<arguments.length; i++) {
  args[i] = arguments[i];
}
```

Пример использования: `copy(dst, src1,...)`

Иногда встаёт задача — скопировать в существующий объект свойства из одного или нескольких других.

Напишем для этого функцию `copy`. Она будет работать с любым числом аргументов, благодаря использованию `arguments`.

Синтаксис:

copy(dst, src1, src2...)

Копирует свойства из объектов `src1, src2,...` в объект `dst`.
Возвращает получившийся объект.

Использование:

- Для добавления свойств в объект `user`:

```
var user = {
  name: "Вася",
};

// добавить свойства
copy(user, {
  age: 25,
  surname:"Петров"
});
```

Использование `copy` позволяет сократить код, который потребовался бы для ручного копирования свойств:

```
user.age = ...
user.surname = ...
...
```

Объект `user` пишется только один раз, и общий смысл кода более ясен.

- Для создания копии объекта `user`:

```
// скопирует все свойства в пустой объект
var userClone = copy({}, user);
```

Такой «клон» объекта может пригодиться там, где мы хотим изменять его свойства, при этом не трогая исходный объект `user`. В нашей реализации мы будем копировать только свойства первого уровня, то есть вложенные объекты не обрабатываются. Впрочем, её можно расширить.

Теперь перейдём к реализации.

Первый аргумент у `copy` всегда есть, поэтому укажем его в определении. А остальные будем получать из `arguments`, вот так:

```
function copy(dst) {
  for (var i=1; i<arguments.length; i++) {
    var obj = arguments[i];
    for (var key in obj) {
      dst[key] = obj[key];
    }
  }
  return dst;
}
```

При желании, такое копирование можно реализовать рекурсивно.

`arguments.callee` и `arguments.callee.caller`

Объект `arguments` не только хранит список аргументов, но и обеспечивает доступ к ряду интересных свойств. В современном стандарте JavaScript они отсутствуют, но часто встречаются в старом коде.

`arguments.callee`

Свойство `arguments.callee` содержит ссылку на функцию, которая

выполняется в данный момент.

Это свойство устарело. Современная спецификация рекомендует использовать именованные функциональные выражения (NFE).

Браузеры могут более эффективно оптимизировать код, если `arguments.callee` не используется.

Тем не менее, свойство `arguments.callee` зачастую удобнее, так как функцию с ним можно переименовывать как угодно и не надо менять ничего внутри. Кроме того, NFE некорректно работают в IE<9.

Например:

```
function f() {
  alert( arguments.callee === f); // true
}

f();
```

Зачем нужно делать какое-то свойство `callee`, если можно использовать просто `f`? Чтобы это понять, рассмотрим несколько другой пример.

В JavaScript есть встроенная функция `setTimeout(func, ms)`, которая вызывает `func` через `ms` миллисекунд, например:

```
// выведет 1 через 1000 ms (1 секунда)
setTimeout( function() { alert(1) }, 1000);
```

Функция, которую вызывает `setTimeout`, объявлена как `Function Expression`, без имени.

А что если хочется из самой функции вызвать себя еще раз? По имени-то обратиться нельзя. Как раз для таких случаев и придуман `arguments.callee`, который гарантирует обращение изнутри функции к самой себе.

То есть, рекурсивный вызов будет выглядеть так:

```
setTimeout(
```

```
  function() {
    alert(1);
    arguments.callee(); // вызвать себя
  },
  1000
)
```

Аргументы можно передавать в `arguments.callee()` так же, как в обычную функцию.

Пример с факториалом:

```
// factorial(n) = n*factorial(n-1)
var factorial = function(n) {
  return n==1 ? 1 : n*arguments.callee(n-1);
}
```

Функция `factorial` не использует свое имя внутри , поэтому рекурсивные вызовы будут идти правильно, даже если функция «уехала» в другую переменную.

```
// factorial(n) = n*factorial(n-1)
var factorial = function(n) {
  return n==1 ? 1 : n*arguments.callee(n-1);
}

var g = factorial;
factorial = 0; // функция переместилась в переменную g

alert( g(5) ); // 120, работает!
```

Рекомендованной альтернативой `arguments.callee` являются именованные функциональные выражения.

`arguments.callee.caller`

Свойство `arguments.callee.caller` хранит ссылку на *функцию, которая вызвала данную*.

Это свойство устарело, аналогично `arguments.callee`.

Также существует похожее свойство `arguments.caller` (без `callee`). Оно не кросс-браузерное, не используйте его. Свойство

`arguments.callee.caller` поддерживается везде.

Например:

```
01 f1();
02
03 function f1() {
04   alert(arguments.callee.caller); // null
05   f2();
06 }
07
08 function f2() {
09   alert(arguments.callee.caller); // f1, функция вызвавшая
   меня
10 }
```

На практике это свойство используется очень редко, например для получения информации о стеке (текущей цепочке вложенных вызовов) в целях логирования ошибок. Но в современных браузерах существуют и другие способы это сделать.

Почему `callee` и `caller` устарели?

В современном стандарте эти свойства объявлены устаревшими, и использовать их не рекомендуется. Хотя де-факто они используются хотя бы потому, что IE до 9 версии не поддерживает Named Function Expression так, как должен.

Причина отказа от этих свойств простая — интерпретатор может оптимизировать JavaScript более эффективно. Тем более, что для `arguments.callee` есть замена — NFE.

Задача: Проверка на аргумент-undefined

Как в функции отличить отсутствующий аргумент от `undefined`?

Важность: 5

```
1 function f(x) {
2   // ..ваш код..
3   // выведите 1, если первый аргумент есть, и 0 -
  если нет
```

```
}

f(undefined); // 1
f(); // 0
```

Решение задачи "Проверка на аргумент-undefined" »»

Задача: Сумма аргументов

Напишите функцию `sum(...)`, которая возвращает сумму всех своих аргументов:

Важность: 5

```
sum() = 0
sum(1) = 1
sum(1, 2) = 3
sum(1, 2 ,3) = 6
sum(1, 2, 3, 4) = 10
```

Решение задачи "Сумма аргументов" »»

Именованные аргументы

Именованные аргументы не имеют отношения к `arguments`.

Это альтернативная техника работы с аргументами, которая позволяет обращаться к ним по имени, а не по номеру. Зачастую это гораздо удобнее.

Представьте себе, что у вас есть функция с несколькими аргументами, причем большинство из них имеют значения по умолчанию.

Например:

```
function showWarning(width, height, title, contents, showYesNo) {
  width = width || 200; // почти все значения - по умолчанию
  height = height || 100;

  title = title || "Предупреждение";

  //...
```

```
}
```

Функция `showWarning` позволяет указать ширину и высоту `width, height`, заголовок `title`, содержание `contents` и создает дополнительную кнопку, если `showYesNo == true`. Большинство этих параметров имеют значение по умолчанию.

В примере выше значения по умолчанию:

- `width = 200`,
- `height = 100`,
- `title = "Предупреждение"`.

Если необязательный параметр находится в середине списка аргументов, то для передачи «значения по умолчанию» обычно используют `null`:

```
// width, height, title - по умолчанию
showWarning(null, null, null, "Предупреждение", true);
```

Неудобство такого вызова заключается в том, что **порядок аргументов легко забыть или перепутать.** Кроме того, «дырки» в списке аргументов — это не красиво.

Обычно необязательные параметры переносятся в конец списка, но если таких большинство, то это невозможно.

Для решения этой проблемы в Python, Ruby и многих языках существуют *именованные аргументы* (keyword arguments, named arguments).

В JavaScript именованные параметры реализуются при помощи объекта. Вместо списка аргумента передается *объект с параметрами*, вот так:

```
function showWarning(options) {
  var width = options.width || 200;  // по умолчанию
  var height = options.height || 100;

  var title = options.title || "Предупреждение";

  // ...
}
```

Вызвать такую функцию очень легко. Достаточно передать объект

аргументов, указав в нем только нужные:

```
showWarning({
  contents: "Вы вызвали функцию",
  showYesNo: true
});
```

Еще один бонус кроме красивой записи — возможность повторного использования объекта аргументов:

```
var opts = {
  width: 400,
  height: 200,
  contents: "Текст",
  showYesNo: true
};

showWarning(opts);

opts.contents = "Другой текст";

showWarning(opts); // не нужно копировать остальные аргументы в вызов
```

Именованные аргументы применяются в большинстве JavaScript-фреймворков.

Решения задач

Решение задачи: Проверка на аргумент-undefined

Узнать количество реально переданных аргументов можно по значению `arguments.length`:

```
function f(x) {
  alert(arguments.length ? 1 : 0);
}

f(undefined);
f();
```

Решение задачи: Сумма аргументов

```
01 function sum() {
02   var result = 0;
03
04   for(var i=0; i<arguments.length; i++) {
05     result += arguments[i];
06   }
07
08   return result;
09 }
10
11 alert( sum() ); // 0
12 alert( sum(1) ); // 1
13 alert( sum(1, 2) ); // 3
14 alert( sum(1, 2, 3) ); // 6
15 alert( sum(1, 2, 3, 4) ); // 10
```