JavaScript

НАРОДНЫЕ СОВЕТЫ

ВЛАДИМИР ДРОНОВ

ОБЩИЕ ПРИЕМЫ ПРОГРАММИРОВАНИЯ

РАБОТА С WEB-ОБОЗРЕВАТЕЛЕМ

РАБОТА С WEB-СТРАНИЦАМИ И ИХ ЭЛЕМЕНТАМИ

РАБОТА С ДАННЫМИ

СЛОЖНОЕ WEB-ПРОГРАММИРОВАНИЕ

bhv

Владимир Дронов

JAVASCRIPT
НАРОДНЫЕ СОВЕТЫ

Санкт-Петербург
«БХВ-Петербург»
2007

УДК 681.3.06
ББК 32.973.26-018.1
Д75

Дронов В. А.

Д75 JavaScript. Народные советы. — СПб.: БХВ-Петербург, 2007. — 464 с.: ил.

ISBN 978-5-94157-961-7

Книга представляет собой подборку решений, зачастую неочевидных, типичных проблем Web-программирования, приемов, советов и готовых Web-сценариев. Рассмотрены следующие темы: полезные функции и объекты языка JavaScript, написание сценариев и обработка событий, получение сведений о Web-обозревателе и управление им, манипуляции и эффекты с Web-страницами и их элементами, работа с графикой, гиперссылками и полосами навигации, вывод информации о таблицах, эффекты с фреймами, управление свободно позиционируемыми контейнерами, создание мультимедийных элементов и управление ими, простейший ввод-вывод, сохранение и передача данных, работа с формами и элементами управления, простейшие и более сложные приемы Web-программирования, отладка Web-приложений и др.

Для Web-дизайнеров и Web-программистов

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 20.10.06.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 37,4.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-94157-961-7

Оглавление

Введение

Язык JavaScript сейчас популярен как никогда. Сложные Web-страницы, реагирующие на действия посетителя, заполонили Интернет. Возникла профессия Web-программиста, специалиста по написанию *Web-сценариев* (или просто *сценариев*) — программ на языке JavaScript, встраиваемых в HTML-код страниц и собственно дающих им интерактивность. А на горизонте маячат новые технологии Web-программирования, дающие посетителям сайтов новые, невиданные доселе возможности.

И, разумеется, нет недостатка в книгах по JavaScript и Web-программированию. Толстые и тонкие, поверхностные и подробные, умные и бестолковые — они издаются и переиздаются в изрядных количествах. В любом более-менее приличном книжном магазине можно найти сразу несколько наименований таких книг — на любой вкус и любую потребность.

И вот еще одна книга. Все по тому же JavaScript. Стоило ли из-за нее переводить бумагу?

Чего не дают обычные руководства?

Большинство издаваемых ныне книг по JavaScript представляют собой либо руководства для начинающих, либо справочники. Справочники мы сразу исключим, так как, во-первых, они немногочисленны (увы!..), а во-вторых, рассчитаны все-таки на подготовленных читателей. Возьмем с полки магазина любое руководство и, пока продавец не возмутился: "Здесь вам не читальный зал!" — полистаем.

Итак, руководство по JavaScript. Описание языка плюс минимальные справочные сведения плюс кое-какие примеры сценариев, совсем несложных. Прочитав все это, желающий стать Web-программистом узнает сам язык, напишет несколько собственных сценариев и, возможно, продолжит свое обу-

чение по другим книгам или материалам, выложенным в Интернете (благо их сейчас хватает). И, разумеется, начнет писать свои сценарии.

И тут-то у него появятся первые проблемы. Дело в том, что зачастую в таких руководствах не рассказывается, как сделать ту или иную вещь. Описан, скажем, объект `String` (объектное представление строкового значения), но не рассказано, как выполнить замену всех вхождений заданной подстроки другой подстрокой. И приходится свежеиспеченному Web-программисту, едва научившемуся писать свои первые выражения JavaScript, придумывать, как это сделать, либо искать уже готовый сценарий.

Итак, главная проблема большинства руководств — они рассказывают, что можно сделать, но не рассказывают, как сделать ту или иную вещь. Можно сказать, что они дают россыпь отдельных деталей, но не прилагают к ним инструкцию по сборке.

Собственно, не стоит этого от них требовать — у руководств весьма специфические задачи. Здесь нужны совсем другие книги, этакие сборники советов.

Народные советы

А ведь тот же самый сценарий по замене одной подстроки другой уже наверняка кем-то написан! (В конце концов, задача эта типична для Web-программиста, имеющего дело, в основном, со строковыми данными.) А значит, писать его во второй (десятый, сотый, тысячный) раз не имеет смысла. Нужно только обратиться к программирующей общественности, к программирующему народу — и он обязательно поможет!

Так вот в этой книге как раз и приведены решения типичных задач Web-программирования и типичные Web-сценарии, которые можно сразу же вставить в Web-страницу. Этакая сокровищница знаний программирующего народа. Народные советы, как гласит само название этой книги.

> ***Народ замечает***
>
> Да-да, замечу без ложной скромности, это так. Правда, тут и сам автор книги здóрово постарался, но тоже не без моей помощи.

Итак, что же мы найдем в этой книге?..

- Решения типичных проблем Web-программирования.
- Типичные Web-сценарии, доступные для использования на Web-страницах без всяких ограничений.
- Собственно народные советы (примечания с заголовком "Народ советует").

- Замечания народа по различным вопросам ("Народ замечает").
- Предупреждения народа о возможных проблемах ("Народ предупреждает!").

А вот чего в этой книге даже не стоит искать, так это руководств по HTML, CSS, JavaScript и Web-обозревателям. Для этого существуют другие книги.

Все это работает?

Еще как! Проверено самим автором на следующих Web-обозревателях:

- Microsoft Internet Explorer 6.0 SP2 русский;
- Opera 8.54 русский;
- Mozilla Firefox 1.5.0.3 русский;
- Mozilla 1.7;
- Netscape Navigator 7.0 Preview Release 1.

Практически все Web-сценарии, приведенные в этой книге, тестировались только на первых трех Web-обозревателях как самых популярных на данный момент. Только сценарий, приведенный в листинге 3.1 (функция `jspsGetProgramInfo`, выдающая сведения о Web-обозревателе), проверялся на всех пяти программах.

Большинство приведенных в этой книге решений работает на всех Web-обозревателях (точнее, на первых трех: Internet Explorer, Opera и Firefox). Если же какое-то решение функционирует только на определенных Web-обозревателях, их названия будут указаны в скобках (например, "Решение (Opera)" для решения, работающего только на Opera).

Типографские соглашения

В этой книге часто приводятся форматы использования различных свойств и вызова различных функций и методов. Нам необходимо выучить типографские соглашения, используемые для их написания.

> ***Народ предупреждает!***
>
> Все эти типографские соглашения применяются автором только при описании формата использования свойств и вызова функций и методов. В коде примеров они не имеют смысла.

Так, в угловые скобки (< >) заключаются названия параметров или фрагментов кода, которые, в свою очередь, набраны курсивом. В код реального сце-

нария, разумеется, должен быть подставлен реальный параметр или реальный код. Например:

```
document.createElement(<Имя тега>);
```

Здесь вместо подстроки *<Имя тега>* должно быть подставлено реальное имя тега.

В квадратные скобки ([]) заключаются необязательные фрагменты кода. Например:

```
AnimatedElement(<Элемент страницы>, <Интервал>, [<Массив выражений>]);
```

Здесь *Массив выражений* может присутствовать, а может и отсутствовать.

Если в какое-либо место сценария должна быть подставлена команда, выбираемая из некоего ограниченного набора, в этом месте приводятся все команды данного набора, разделенные символом | ("вертикальная черта"). Например:

```
<PUBLIC:ATTACH EVENT="oncontentready|ondocumentready"
```

Здесь в качестве значения атрибута EVENT тега PUBLIC:ATTACH может присутствовать либо "oncontentready", либо "ondocumentready" (но не оба одновременно).

Слишком длинные, не помещающиеся на одной строке выражения JavaScript автор разрывает на несколько строк и в местах разрывов ставит знаки ↳. Например:

```
<PUBLIC:ATTACH EVENT="oncontentready|ondocumentready"
↳[FOR="<Элемент страницы>"]
↳ONEVENT="<Вызов функции-обработчика события>"/>
```

Приведенный выше код разбит на три строки, но должен быть набран в одну. Знаки ↳ при этом должны быть удалены.

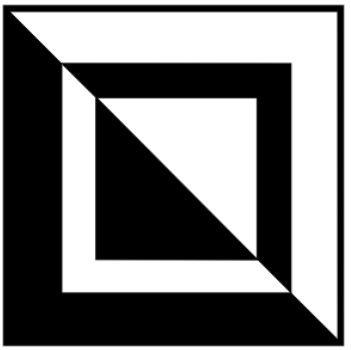
> ***Народ предупреждает в последний раз!***
>
> Все приведенные выше типографские соглашения имеют смысл только при описании формата использования свойств и вызова функций и методов. В коде примеров используется только знак ↳.

Что ж, автор вроде все сказал... Теперь пусть за него говорит всезнающий народ! Предоставим ему слово!..

Благодарности

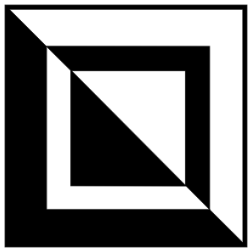
Автор приносит благодарности своим родителям, знакомым и коллегам по работе.

- Губине Наталье Анатольевне, начальнику отдела АСУ Волжского гуманитарного института (г. Волжский Волгоградской обл.), где работает автор, — за понимание и поддержку.
- Всем работникам отдела АСУ — за понимание и поддержку.
- Родителям — за терпение, понимание и поддержку.
- Архангельскому Дмитрию Борисовичу — за дружеское участие.
- Шапошникову Игорю Владимировичу — за содействие.
- Евгению из Волгограда — за фильмы ужасов — лучшее средство для развития чувства юмора.
- Рыбакову Евгению Евгеньевичу, заместителю главного редактора издательства "БХВ-Петербург", — за неоднократные побуждения к работе, без которых автор давно бы обленился.
- Издательству "БХВ-Петербург" — за издание моих книг.
- Всем своим читателям и почитателям — за прекрасные отзывы о моих книгах.
- Всем, кого я забыл здесь перечислить, — за все хорошее.
- Особая благодарность — программирующему народу!



ЧАСТЬ I

Общие приемы программирования



ГЛАВА 1

Полезные функции, методы и приемы программирования на JavaScript

В этой главе народ ответит на различные вопросы, связанные с самим языком JavaScript, а также предоставит несколько полезных функций и методов, которых в этом языке явно не хватает.

Работа с переменными и их значениями

Начнутся народные советы с решения проблем с переменными и хранящимися в них значениями.

Как проверить существование переменной?

Проблема

Как проверить, была ли уже объявлена какая-либо переменная?

Решение

Проще всего воспользоваться оператором `typeof`, возвращающим в строковом виде тип значения переменной. В случае еще не объявленной переменной он вернет строку "undefined".

Вот шаблон выражения, выполняющего проверку на существование переменной:

```
if (typeof(<Имя проверяемой переменной>) == "undefined")
  <Код, выполняемый, если переменная не объявлена>
else
  <Код, выполняемый, если переменная объявлена>
```

Пример

```
if (typeof(pi) == "undefined") pi = 3.14;
```

Как избежать конфликтов имен переменных?

Проблема

Имеется большой проект, содержащий множество глобальных переменных, причем различные части этого проекта написаны разными разработчиками. Имеется ли какой-либо способ избежать *конфликтов имен* переменных, т. е. исключить ситуации, когда один разработчик объявляет переменную с именем, уже использованным другим разработчиком в другой части того же сценария?

Решение

Решение здесь будет чисто организационное. Нужно просто продумать систему именования глобальных переменных и строго ей следовать. Система эта может быть, скажем, такой:

```
<Аббревиатура проекта>_<Аббревиатура части проекта>_<Тип данных>
[_<Подтип данных>]
```

Здесь:

- *Аббревиатура проекта* содержит наименование проекта (скажем, `emag` — электронный магазин);
- *Аббревиатура части проекта* указывает на часть проекта, разрабатываемую данным программистом (например, `checkData` — код, выполняющий проверку введенных покупателем данных о себе);
- *Тип данных* обозначает тип хранящегося в переменной значения (так, если переменная хранит сведения о покупателе, *Тип данных* можно назвать `custData`);
- необязательный *Подтип данных* можно использовать для указания на отдельную часть данных, относящихся к заданному ранее *Типу данных* (для имени покупателя это может быть `custName`).

> ***Народ советует***
>
> Эту систему можно расширить. Так, функции, реализующие методы объектов (о них мы поговорим далее в этой главе), могут иметь имена, начинающиеся на букву "m", а параметры функций — на букву "p". Также в имена функций, реализующих методы объектов, можно включить аббревиатуру, обозначающую объект, для которого они предназначены. Кстати, далее в своей книге автор так и делает.

Пример

```
var emag_checkData_custData_custName = getCustName();
```

Народ советует

Вообще, чем длиннее имя какой-либо переменной, тем меньше вероятность, что где-то в коде встретится другая переменная с таким же именем. Следует иметь это в виду.

Как создать константу?

Проблема

В коде сценария JavaScript часто используется одно и то же строковое или числовое значение. Можно ли вместо того, чтобы каждый раз указывать это значение, создать на его основе именованную *константу* и впоследствии использовать именно ее?

Решение

К сожалению, в отличие от других языков программирования, JavaScript не предоставляет никаких способов создать именованные константы. Но можно создать переменную, назвать ее каким-либо особым образом и присвоить ее нужное значение, создав, если можно так сказать, *псевдоконстанту*. Единственное "но": впоследствии не следует изменить значение этой переменной (константа она, в конце концов, или нет!).

Константы имеют одно неоспоримое преимущество. Если нам понадобится в дальнейшем изменить значение какой-нибудь константы, нам будет достаточно сделать это всего один раз — в выражении, объявляющем соответствующую переменную-псевдоконстанту. Если же мы не использовали константу, то будем вынуждены просмотреть все наши сценарии и исправить все использующие нужное нам значение выражения. А это зачастую очень трудоемко.

Пример

```
var EMAG_VERSION = "1.0";
. . .
document.write("Электронный магазин " + EMAG_VERSION);
. . .
document.write("Версия: " + EMAG_VERSION);
```

Этот сценарий сначала объявляет переменную-псевдоконстанту `EMAG_VERSION`, хранящую номер версии электронного магазина, а потом выводит этот номер на Web-страницу в двух местах.

Народ советует

Хорошим тоном программирования считается именование констант только большими буквами. Так мы, с одной стороны, сразу дадим своим коллегам понять, что это именно константа, а с другой — избежим возможных конфликтов имен.

Также хорошим тоном программирования считается вынесение всех глобальных констант, используемых в нескольких сценариях, в отдельный файл сценариев. Этот файл можно назвать, скажем, constants.js. Разумеется, этот файл придется потом подключить к Web-странице с помощью тега

```
<SCRIPT SRC="constants.js"></SCRIPT>
```

О файлах сценариев будет подробно рассказано в *главе 2*.

Работа с функциями

Здесь народ расскажет о различных полезных, но неочевидных приемах написания функций.

Как создать необязательные параметры функции?

Проблема

Есть функции, некоторые параметры которых очень часто принимают одни и те же значения. Нельзя ли сделать эти параметры *необязательными*, чтобы при вызове функции их можно было пропускать, и при этом они сами принимали бы нужные значения?

Решение

Если при вызове функции какой-либо из ее параметров был пропущен, в коде тела функции ему будет присвоено значение `undefined`. Так что последовательность действий ясна.

1. Проверяем, имеет ли этот параметр значение `undefined` (т. е. был ли он передан функции явно). Если да, переходим к шагу 2, если нет — к шагу 3.
2. Присваиваем параметру значение по умолчанию.
3. Используем значение этого параметра в вычислениях внутри тела функции.

Пример 1

Вот объявление функции, второй параметр которой — `par2` — является необязательным и имеющим значение по умолчанию 0.

```
function func1(par1, par2)
  {
    if (typeof(par2) == "undefined") par2 = 0;
    . . .
  }
```

Мы можем вызвать эту функцию как с указанием всех параметров:

```
func1(5, 2);
```

так и с указанием только первого — обязательного — параметра:

```
func1(10);
```

В последнем случае второй параметр получит в теле функции значение 0. Эффект будет тот же, если бы мы вызвали функцию с такими параметрами:

```
func1(10, 0);
```

Пример 2

Выражение проверки параметра на значение `undefined` в теле функции можно записать короче. Вот так:

```
function func1(par1, par2)
  {
    if (!par2) par2 = 0;
    . . .
  }
```

Значение `undefined`, будучи составной частью логического выражения, всегда преобразуется в значение `false`. В приведенном выше примере мы это и использовали.

> ***Внимание!***
>
> Необязательные параметры должны находиться в самом конце списка параметров функции (см. приведенные ранее примеры). Если же мы, например, сделаем необязательным первый параметр функции и вызовем ее таким образом:
>
> ```
> func1(, 2);
> ```
>
> пропустив его, это вызовет синтаксическую ошибку. Интерпретатор JavaScript не любит "пустых мест" в начале и середине списка параметров вызываемой функции.

Как передать функции произвольное число параметров?

Проблема

Нужно написать функцию, которая принимает произвольное число параметров. Как это сделать?

Решение 1

Внутри тела функции доступен массив `arguments`, элементы которого содержат все переданные функции параметры. Этот массив имеет свойство `length`, возвращающее количество элементов массива.

Пример 1

Вот объявление функции, принимающей произвольное количество строковых параметров и возвращающей строку, составленную из их значений:

```
function func2()
  {
    var s = "";
    var i = 0;
    var c = arguments.length;
    if (c > 0)
      for (i = 0; i < c; i++)
        s += arguments[i];
    return s;
  }
```

Выражение

```
var str = func2("Java", "Script");
```

поместит в переменную `str` строку "JavaScript".

Пример 2

Еще одна функция, принимающая произвольное количество строковых параметров и возвращающая строку, составленную из их значений. Но на этот раз первым, обязательным, параметром она принимает символ-разделитель, который будет помещаться между этими строками.

```
function func3(pDelimiter)
  {
    var s = "";
    var i = 0;
    var c = arguments.length;
    if (c > 1)
      for (i = 1; i < c; i++)
        {
          s += arguments[i];
          if (i != c - 1) s += pDelimiter;
        }
    return s;
  }
```

Выражение

```
var str = func3("+", "Java", "Script");
```

поместит в переменную `str` строку "Java+Script".

Решение 2

В конце концов, можно передать в функцию в качестве параметра массив с нужным количеством элементов.

Пример

Немного измененный пример функции `func3`:

```
function func3(pDelimiter, pars)
  {
    var s = "";
    var i = 0;
    var c = pars.length;
    if (c > 0)
      for (i = 0; i < c; i++)
        {
          s += pars[i];
          if (i != c - 1) s += pDelimiter;
        }
    return s;
  }
```

Вызываться эта функция будет так:

```
var str = func3(" ", new Array("Java", "Script"));
```

после чего в переменной `str` окажется строка "Java Script".

Возврат из функции нескольких значений

Проблема

Что делать, если какая-либо функция вычисляет не одно, а сразу несколько значений, и их все нужно вернуть? Стандартными средствами JavaScript (а именно оператором `return`) можно вернуть только одно значение.

Решение 1

1. Вычисляем в теле функции все нужные значения.
2. Создаем в теле функции массив.

3. Помещаем вычисленные значения в элементы этого массива.

4. Возвращаем полученный массив оператором `return`.

А уж извлечь значения элементов массива, возвращенного функцией, никакого труда не составит.

Примеры

Вот объявление функции, вычисляющей несколько значений и возвращающих их в массиве:

```
function someFunc()
  {
    // Здесь помещаются выражения, вычисляющие нужные значения.
    // Предположим, что всего их три, и они помещаются в переменные
    // value1, value2 и value3.
    var arr = new Array();
    arr[0] = value1;
    arr[1] = value2;
    arr[2] = value3;
    return arr;
  }
```

Использование этой функции:

```
var a = someFunc();
var v1 = a[0];
var v2 = a[1];
var v3 = a[2];
```

Вместо обычных массивов с числовыми индексами мы можем использовать *хэши* — массивы со строковыми индексами элементов. Например:

```
function someFunc()
  {
    // Здесь помещаются выражения, вычисляющие нужные значения
    // value1, value2 и value3.
    var arr = new Array();
    arr["value1"] = value1;
    arr["value2"] = value2;
    arr["value3"] = value3;
    return arr;
  }
var a = someFunc();
var v1 = a["value1"];
var v2 = a["value2"];
var v3 = a["value3"];
```

Пожалуй, так будет даже нагляднее, особенно если дать элементам массива вразумительные строковые индексы.

Народ советует

Если для возврата значений из функций используются хэши, логично оформить индексы их элементов в виде псевдоконстант. (Псевдоконстанты JavaScript были описаны ранее.) Например:

```
var JSPS_SFARR_1 = "value1";
var JSPS_SFARR_2 = "value2";
var JSPS_SFARR_3 = "value3";
```

А потом использовать для доступа к элементам хэша именно эти константы:

```
function someFunc()
  {
    // Здесь помещаются выражения, вычисляющие нужные значения
    // value1, value2 и value3.
    var arr = new Array();
    arr[JSPS_SFARR_1] = value1;
    arr[JSPS_SFARR_2] = value2;
    arr[JSPS_SFARR_3] = value3;
    return arr;
  }
var a = someFunc();
var v1 = a[JSPS_SFARR_1];
var v2 = a[JSPS_SFARR_2];
var v3 = a[JSPS_SFARR_3];
```

Решение 2

1. Вычисляем в теле функции все нужные значения.
2. Создаем в теле функции экземпляр объекта `Object`.
3. Помещаем вычисленные значения в свойства этого экземпляра.
4. Возвращаем полученный экземпляр оператором `return`.

Извлечь вычисленные функцией значения можно, обратившись к свойствам возвращенного экземпляра объекта `Object`.

Пример

Объявление функции, возвращающей несколько значений с помощью экземпляра объекта `Object`:

```
function someFunc()
  {
    // Здесь помещаются выражения, вычисляющие нужные значения
    // value1, value2 и value3.
```

```
    var obj = new Object();
    obj.value1 = value1;
    obj.value2 = value2;
    obj.value3 = value3;
    return obj;
  }
```

Использование ее:

```
var o = someFunc();
var v1 = o.value1;
var v2 = o.value2;
var v3 = o.value3;
```

Народ советует

Желающие могут даже создать особый объект, предназначенный специально для возврата из функции вычисленных ею значений. (Подробнее о создании объектов будет рассказано далее.)

```
function jspsSFReturnHolder(v1, v2, v3)
  {
    this.value1 = v1;
    this.value2 = v2;
    this.value3 = v3;
  }
function someFunc()
  {
    // Здесь помещаются выражения, вычисляющие нужные значения
    // value1, value2 и value3.
    var obj = new jspsSFReturnHolder(value1, value2, value3);
    return obj;
  }
```

Решение 3

Не столь изящное, как первые два, но используется, пожалуй, чаще всего.

1. Создаем массив или экземпляр объекта `Object`, который передается в функцию в качестве одного из параметров.
2. В теле функции заносим вычисленные ею значения в этот массив или экземпляр объекта.
3. После завершения работы функции извлекаем полученные значения из элементов массива или свойств экземпляра объекта.

При этом из функции вообще может не возвращаться никакое значение (имеется в виду возврат с помощью оператора `return`).

Пример

Объявление функции, возвращающей несколько значений в массиве, переданном ей в качестве параметра:

```
function someFunc(arr)
  {
    // Здесь помещаются выражения, вычисляющие нужные значения
    // value1, value2 и value3.
    arr[0] = value1;
    arr[1] = value2;
    arr[2] = value3;
  }
```

Ее использование:

```
var a = new Array();
someFunc(a);
var v1 = a[0];
var v2 = a[1];
var v3 = a[2];
```

Реализация статических переменных

Проблема

Локальные переменные, объявленные в теле функции, не сохраняют свои значения между вызовами этой функции. Но иногда бывает необходимо создать так называемую *статическую переменную* — локальную переменную, которая сохраняет свое значение между вызовами функции. Как это сделать?

Решение

Увы!.. JavaScript не предоставляет для этого стандартных средств. Нам придется объявить глобальную переменную и использовать ее в теле функции.

Пример

Приведенная далее функция сохраняет число своих вызовов в глобальной переменной `counter`.

```
var counter = 0;
function func4()
  {
    . . .
    counter++;
    . . .
  }
```

Работа с объектами

Здесь описаны некоторые приемы создания пользовательских и расширения встроенных объектов JavaScript.

Как создать пользовательский объект?

Проблема

Нужно создать пользовательский объект, имеющий определенные свойства и методы.

Решение

Объявление пользовательского объекта в JavaScript суть объявление особой функции, которая создает все его свойства и методы и заполняет их начальными значениями. Эта функция называется *конструктором*.

Формат объявления функции-конструктора ничем не отличается от формата объявления обычной функции:

```
function <Имя пользовательского объекта> ([<Список параметров>])
  {
    <Объявление свойств объекта>
    <Объявление методов объекта>
  }
```

Заметим, что имя функции-конструктора должно совпадать с именем объекта, который мы хотим создать.

Формат выражения, создающего новое свойство пользовательского объекта, таков:

```
this.<Имя свойства> = <Значение свойства>;
```

В данном случае ключевое слово `this` обозначает создаваемый объект.

> ***Внимание!***
>
> Ключевое слово `this` имеет смысл только в теле функции-конструктора.

Формат выражения, создающего метод, аналогичен:

```
this.<Имя метода> = <Функция, реализующая этот метод>;
```

Фактически здесь мы присваиваем функцию созданному свойству. Заметим при этом, что в данном случае имя функции указывается без списка параметров и без скобок. Разумеется, реализующая метод функция должна быть объявлена.

В функциях, реализующих методы создаваемого объекта, мы также можем использовать ключевое слово this. Собственно, даже не можем, а должны — ведь это единственный способ получить доступ к свойствам и методам этого объекта.

Созданный таким образом объект мы можем использовать в сценариях. Для создания на его основе экземпляра мы воспользуемся знакомым нам оператором new, а для удаления этого экземпляра — оператором delete.

Народ замечает

Описанный выше трюк с присвоением функции переменной работает и вне тела функции-конструктора. Так, мы можем присвоить любую функцию переменной:

```
function func5()
  {
    // Тело функции
  }
var varFunc = func5;
```

а потом вызвать ее, обратившись к этой переменной:

```
varFunc();
```

Повторим — при присвоении функции переменной имя присваиваемой функции указывается без списка параметров и без скобок, иначе интерпретатор JavaScript посчитает его за вызов этой функции.

Пример

Давайте создадим объект Point, обозначающий точку на экране. Этот объект будет содержать свойства x и y (координаты точки) и метод setCoords (помещение значений координат, переданных в качестве параметров, в соответствующие свойства).

Функция-конструктор этого объекта будет принимать два параметра — координаты точки. Эти координаты будут помещены в свойства x и y создаваемого экземпляра объекта.

```
// Объявление функции-конструктора объекта Point
function Point(ix, iy)
  {
    this.x = ix;
    this.y = iy;
    this.setCoords = fSetCoords;
  }
// Объявление функции, реализующей метод setCoords объекта Point
function fSetCoords(ix, iy)
```