Современный учебник JavaScript

© Илья Кантор

Сборка от 27 апреля 2014 для печати

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: Объекты и методы

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

Содержание

Свои объекты: конструкторы и методы

До этого мы говорили об объекте лишь как о хранилище значений. Теперь пойдём дальше и поговорим о добавлении в объекты собственных функций (методов), а также о конструкторах — функциях, создающих объекты.

Свои методы объектов

При объявлении в объект можно записать функцию. Она становится его методом, например:

```
var user = {
  name: 'Василий',

  // метод
  sayHi: function() {
    alert('Привет!');
  }

};

// Вызов метода
user.sayHi();
```

Можно создать метод и позже, явным присвоением:

```
var user = {
  name: 'Василий'
};

user.sayHi = function() {
  alert('Привет!');
};

// Вызов метода:
user.sayHi();
```

Доступ к объекту через `this`

Для полноценной работы метод должен иметь доступ к данным объекта. В частности, вызов `user.sayHi()` может захотеть вывести имя пользователя.

Для доступа к объекту из метода используется ключевое слово `this`. Значением `this` является объект, в контексте которого вызван метод, например:

```
var user = {
  name: 'Василий',

  sayHi: function() {
    alert( this.name );
  }
};

user.sayHi();
```

Здесь при выполнении функции `user.sayHi()` в `this` будет храниться ссылка на текущий объект `user`.

В данном случае вместо `this` можно было бы использовать и переменную: `alert( user.name )`, но объект `user` может быть куда-то передан, переменная `user` перезаписана и т.п. Использование `this` гарантирует, что функция работает именно с тем объектом, в контексте которого вызвана.

Через `this` можно обратиться к любому свойству объекта, а при желании и передать его куда-то:

```
01 var user = {
02   name: 'Василий',
03
04   sayHi: function() {
05     showName(this); // передать текущий объект в showName
06   }
07 };
08
09 function showName(obj) {
10   alert( obj.name );
11 }
12
13 user.sayHi();
```

Функция-конструктор, «new»

Обычный синтаксис `{...}` позволяет создать один объект. Но зачастую нужно создать много однотипных объектов.

Для этого используют функции, запуская их при помощи специального оператора `new`.

Конструктором становится любая функция, вызванная через `new`.

Например:

```
1 function Animal(name) {
2   this.name = name;
3   this.canWalk = true;
4 }
5
6 var animal = new Animal("ёжик");
```

Любую функцию можно вызвать при помощи `new`. При этом она работает несколько иным образом, чем обычно:

1. Автоматически создается новый, пустой объект.
2. Специальное ключевое слово `this` получает ссылку на этот объект.
3. Функция выполняется. Как правило, она модифицирует `this`, добавляет методы, свойства.
4. Возвращается `this`.

Так что результат выполнения примера выше — это объект:

```
1 animal = {
2   name: "ёжик",
3   canWalk: true
4 }
```

О создаваемом объекте говорят, что это «объект класса(или типа) `Animal`».

Термин «класс» здесь является профессиональным жаргоном. Во многих других языках программирования есть специальная сущность «класс». В JavaScript её нет, но кое-что похожее организовать можно, поэтому так и называют.

Функция может возвратить другой объект вместо `this`

Если функция явно возвращает объект, то будет возвращён он, а не `this`.

Например:

```
function BigAnimal() {

  this.name = 'Мышь';

  return { name: 'Годзилла' };  // <-- будет возвращено
}

alert( new BigAnimal().name );  // Годзилла
```

Если функция возвращает не объект, к примеру, число, то такой вызов `return` ни на что не повлияет. Например:

```
function BigAnimal() {

  this.name = 'Мышь';

  return 'Годзилла'; // не объект, такой return в режиме new ни на что не влияет
}

alert( new BigAnimal().name );  // Мышь
```

Эта особенность работы `new` прописана в стандарте, знать о ней полезно, но используется она весьма редко.

Названия функций, которые предназначены создавать объекты, как правило, начинают с большой буквы.

Кстати, при вызове `new` без аргументов скобки можно не ставить:

```
var animal = new BigAnimal; // <-- без скобок
// то же самое что
var animal = new BigAnimal();
```

Создание методов в конструкторе

Использование функций для создания объекта дает большую гибкость. Можно передавать функции свойства создаваемого объекта и

параметры, определяющие как его создавать.

Например, функция `User(name)` создает объект с заданным значением свойства `name` и методом `sayHi`:

```
function User(name) {
  this.name = name;

  this.sayHi = function() {
    alert("Моё имя: " + this.name);
  };
}
```

Пример использования:

```
var ivan = new User("Иван");

/* Объект ivan имеет вид:
 {
   name: "Иван",
   sayHi: функция, обращение к имени идёт через this.name
 }
*/

ivan.sayHi(); // Моё имя: Иван
```

Свойства объекта могут со временем изменяться. Используйте это в следующей задаче.

Приватные свойства

Локальные переменные функции-конструктора, с одной стороны, доступны вложенным функциям, с другой — недоступны снаружи.

В объектно-ориентированном программировании это называется «приватный (private) доступ».

Например, в коде ниже к `name` имеет доступ только метод `say`. Со стороны объекта, после его создания, больше никто не может получить `name`.

```
function User(name) {

  this.say = function(phrase) {
    alert(name + ' сказал: ' + phrase);
  };

}

var user = new User('Вася');
```

Если бы `name` было свойством `this.name` — можно было бы получить его как `user.name`, а тут — локальная переменная. Приватный доступ.

Замыкания никак не связаны с this

Доступ через замыкание осуществляется к локальной переменной, находящейся «выше» по области видимости.

А this содержит ссылку на «текущий» объект — контекст вызова, и позволяет обращаться к его свойствам. С локальными переменными это никак не связано.

Приватные свойства можно менять, например ниже метод this.upperCaseName() меняет приватное свойство name:

```
01 function User(name) {
02
03   this.upperCaseName = function() {
04     name = name.toUpperCase(); // <-- изменяет name из User
05   };
06
07   this.say = function(phrase) {
08     alert(name + ' сказал: ' + phrase); // <-- получает name из User
09   };
10
11 }
12
13 var user = new User('Вася');
14
15 user.upperCaseName();
16
17 user.say("Да здравствует ООП!") // ВАСЯ сказал: Да здравствует ООП!
```

Вы помните, в главе Замыкания, функции изнутри [1] мы говорили о скрытых ссылках [[Scope]] на внешний объект переменных? В этом примере user.upperCaseName.[[Scope]] и user.say.[[Scope]] как раз ссылаются на один и тот же объект LexicalEnvironment, в контексте которого они были созданы. За счёт этого обе функции имеют доступ к name и другим локальным переменным.

Все переменные конструктора User становятся приватными, так как доступны только через замыкание, из внутренних функций.

Итого

У объекта могут быть методы:

- Свойство, значение которого - функция, называется *методом объекта* и может быть вызвано как obj.method(). При этом объект доступен как this.

Объекты могут быть созданы при помощи функций-конструкторов:

- Любая функция может быть вызвана с new, при этом она получает новый пустой объект в качестве this, в который она добавляет свойства. Если функция не решит возвратить свой объект, то её результатом будет this.
- Функции, которые предназначены для создания объектов, называются *конструкторами*. Их названия пишут с большой буквы, чтобы отличать от обычных.

Контекст this в деталях

Значение `this` в JavaScript не зависит от объекта, в котором создана функция. Оно определяется *во время вызова*.

Любая функция может иметь в себе `this`.

Совершенно неважно, объявлена она в объекте или вне него.

Значение `this` называется *контекстом вызова* и будет определено в момент вызова функции.

Например: такая функция вполне допустима:

```
function sayHi() {
  alert( this.firstName );
}
```

Эта функция ещё не знает, каким будет `this`. Это выяснится при выполнении программы.

Есть несколько правил, по которым JavaScript устанавливает `this`.

Вызов функции с `new`

При вызове функции с `new`, значением `this` является новосоздаваемый объект. Мы уже обсуждали это в разделе о создании объектов [2].

Вызов в контексте объекта

Самый распространенный случай — когда функция объявлена в объекте или присваивается ему, как в примере ниже:

```
01 var user = {
02   firstName: "Вася"
03 };
04
05 function func() {
06   alert( this.firstName );
07 }
08
09 user.sayHi = func;
10
11 user.sayHi();  // this = user
```

При вызове функции *как метода объекта*, через точку или квадратные скобки — функция получает в `this` этот объект. В данном случае `user.sayHi()` присвоит `this = user`.

Если одну и ту же функцию запускать в контексте разных объектов, она будет получать разный `this`:

```
var user = { firstName: "Вася" };
var admin = { firstName: "Админ" };

function func() {
  alert( this.firstName );
}

user.a = func;  // присвоим одну функцию в свойства
admin.b = func; // двух разных объектов user и admin

user.a(); // Вася
admin['b'](); // Админ (не важно, доступ через точку или квадратные скобки)
```

Значение `this` не зависит от того, как функция была создана, оно определяется исключительно в момент вызова.

Вызов в режиме обычной функции

Если функция использует `this` - это подразумевает работу с объектом. Но и прямой вызов `func()` технически возможен.

Как правило, такая ситуация возникает при ошибке в разработке.

При этом `this` получает значение `window`, *глобального объекта*.

```
function func() {
  alert(this); // выведет [object Window] или [object global]
}

func();
```

В современном стандарте языка это поведение изменено, вместо глобального объекта `this` будет `undefined`.

```
function func() {
  "use strict";
  alert(this); // выведет undefined (кроме IE<10)
}

func();
```

…Но по умолчанию браузеры ведут себя по-старому.

Явное указание this: apply и call

Функцию можно вызвать, явно указав значение `this`.

Для этого у неё есть два метода: `call` и `apply`.

Метод `call`

Синтаксис метода `call`:

```
func.call(context, arg1, arg2,...)
```

При этом вызывается функция `func`, первый аргумент `call` становится её `this`, а остальные передаются «как есть».

Вызов `func.call(context, a, b...)` — то же, что обычный вызов `func(a, b...)`, но с явно указанным контекстом `context`.

Например, функция `showName` в примере ниже вызывается через `call` в контексте объекта `user`:

```
var user = {
  firstName: "Василий",
  lastName: "Петров"
};

function showName() {
  alert( this.firstName + ' ' + this.lastName );
}

showName.call(user)  // "Василий Петров"
```

Можно сделать её более универсальной, добавив аргументы:

```
var user = {
  firstName: "Василий",
  surname: "Петров"
};

function getName(a, b) {
  alert( this[a] + ' ' + this[b] );
}

getName.call(user, 'firstName', 'surname')  // "Василий Петров"
```

Здесь функция `getName` вызвана с контекстом `this = user` и выводит `user['firstName']` и `user['surname']`.

Метод `apply`

Метод `call` жёстко фиксирует количество аргументов, через запятую:

```
f.call(context, 1, 2, 3);
```

..А что, если мы захотим вызвать функцию с четырьмя аргументами? А что, если количество аргументов заранее неизвестно, и определяется во время выполнения?

Для решения этой задачи существует метод `apply`.

Вызов функции при помощи `func.apply` работает аналогично `func.call`, но принимает массив аргументов вместо списка:

```
func.call(context, arg1, arg2...)
// то же что и:
func.apply(context, [arg1, arg2 ... ]);
```

Эти две строчки сработают одинаково:

```
getName.call(user, 'firstName', 'surname');

getName.apply(user, ['firstName', 'surname']);
```

Метод `apply` гораздо мощнее, чем `call`, так как можно сформировать массив аргументов динамически:

```
var args = [];
args.push('firstName');
args.push('surname');

func.apply(user, args); // вызовет func('firstName', 'surname') с this=user
```

Вызов `call/apply` с `null` или `undefined`

При указании первого аргумента `null` или `undefined` в `call/apply`, функция получает `this = window`:

```
function f() { alert(this) }

f.call(null); // window
```

Это поведение исправлено в современном стандарте (15.3 [3]).

Если функция работает в строгом режиме, то `this` передаётся «как есть»:

```
function f() {
  "use strict";

  alert(this); // null, "как есть"
}

f.call(null);
```

«Одалживание метода»

При помощи `call/apply` можно легко взять метод одного объекта, в том числе встроенного, и вызвать в контексте другого.

В JavaScript методы объекта, даже встроенные — это функции. Поэтому можно скопировать функцию, даже встроенную, из одного объекта в другой.

Это называется «одалживание метода» (на англ. *method borrowing*).

Используем эту технику для упрощения манипуляций с `arguments`. Как мы знаем, это не массив, а обычный объект.. Но как бы хотелось вызывать на нём методы массива.

```
function sayHi() {
  arguments.join = [].join; // одолжили метод (1)

  var argStr = arguments.join(':');  // (2)

  alert(argStr);  // сработает и выведет 1:2:3
}

sayHi(1, 2, 3);
```

В строке `(1)` создали массив. У него есть метод `[].join(..)`, но мы не вызываем его, а копируем, как и любое другое свойство в объект `arguments`. В строке `(2)` запустили его, как будто он всегда там был.

Почему вызов сработает?

Здесь метод join [4] массива скопирован и вызван в контексте `arguments`. Не произойдёт ли что-то плохое от того, что `arguments` — не массив? Почему он, вообще, сработал?

Ответ на эти вопросы простой. В соответствии со спецификацией [5] , внутри `join` реализован примерно так:

```
function join(separator) {
  if (!this.length) return '';

  var str = this[0];

  for (var i = 1; i<this.length; i++) {
    str += separator + this[i];
  }

  return str;
}
```

Как видно, используется `this`, числовые индексы и свойство `length`. Если эти свойства есть, то все в порядке. А больше ничего и не нужно. Подходит даже обычный объект:

```
var obj = {  // обычный объект с числовыми индексами и length
  0: "А",
  1: "Б",
  2: "В",
  length: 3
};

obj.join = [].join;
alert( obj.join(';') ); // "А;Б;В"
```

…Однако, прямое копирование метода не всегда приемлемо.

Представим на минуту, что вместо `arguments` у нас — произвольный объект, и мы хотим вызвать в его контексте метод `[].join`. Копировать этот метод, как мы делали выше, опасно: вдруг у объекта есть свой собственный `join`? Перезапишем, а потом что-то сломается..

Для безопасного вызова используем `apply/call`:

```
function sayHi() {
  var join = [].join; // ссылка на функцию теперь в переменной

  // вызовем join с this=arguments,
  // этот вызов эквивалентен arguments.join(':') из примера выше
  var argStr = join.call(arguments, ':');

  alert(argStr);  // сработает и выведет 1:2:3
}

sayHi(1, 2, 3);
```

Мы вызвали метод без копирования. Чисто, безопасно.

Делаем из `arguments` настоящий `Array`

В JavaScript есть очень простой способ сделать из `arguments` настоящий массив. Для этого возьмём метод массива: arr.slice(start, end) [6] .

По стандарту он копирует часть массива `arr` от `start` до `end` в новый массив. А если `start` и `end` не указаны, то копирует весь массив.

Вызовем его в контексте `arguments`:

```
function sayHi() {
  // вызов arr.slice() скопирует все элементы из this в новый массив
  var args = [].slice.call(arguments);

  alert( args.join(':') ); // args -- массив аргументов
}

sayHi(1,2);
```

Как и в случае с `join`, такой вызов возможен потому, что `slice` использует от массива только нумерованные свойства и `length`. Всё это в `arguments` есть.

«Переадресация» вызова через `apply`

При помощи `apply` мы можем сделать универсальную «переадресацию» вызова из одной функции в другую.

Например, функция `f` вызывает `g` в том же контексте, с теми же аргументами:

```
function f(a, b) {
  g.apply(this, arguments);
}
```

Плюс этого подхода — в том, что он полностью универсален:

- Его не понадобится менять, если в `f` добавятся новые аргументы.
- Если `f` является методом объекта, то текущий контекст также будет передан. Если не является — то `this` здесь вроде как не при чём, но и вреда от него не будет.

Итого

Значение `this` устанавливается в зависимости от того, как вызвана функция:

При вызове функции как метода

```
obj.func(...)    // this = obj
obj["func"](...)
```

При обычном вызове

```
func(...)        // this = window
```

В `new`

```
new func()       // this = {} (новый объект)
```

Явное указание

```
func.apply(ctx, args) // this = ctx (новый объект)
func.call(ctx, arg1, arg2, ...)
```

Приём программирования "Декоратор"

Декоратор [7] — приём программирования, который позволяет взять существующую функцию и изменить/расширить ее поведение.

Декоратор получает функцию и возвращает обертку, которая модифицирует (*декорирует*) её поведение, оставляя синтаксис вызова тем же.

Пример декоратора

Например, у нас есть функция `sum(a,b)`:

```
function sum(a, b) {
  return a + b;
}
```

Создадим декоратор `doublingDecorator`, который меняет поведение, увеличивая результат работы функции в два раза:

```
01 function doublingDecorator(f) {
02   return function() {
03     return 2*f.apply(this, arguments); // (*)
04   };
05 }
06
07 // Использование:
08
09 function sum(a, b) {
10   return a + b;
11 }
12
13 sum = doublingDecorator(sum);
14
15 alert( sum(1,2) ); // 6
16 alert( sum(2,3) ); // 10
```

Декоратор `doublingDecorator` создает анонимную функцию-обертку, которая в строке `(*)` вызывает `f` при помощи apply [8] с тем же контекстом `this` и аргументами `arguments`, а затем удваивает результат.

Этот декоратор можно применить два раза:

```
1 sum = doublingDecorator(sum);
2 sum = doublingDecorator(sum);
3
4 alert( sum(1,2) ); // 12, т.е. 3 умножается на 4
```

Контекст `this` в `sum` никак не используется, поэтому можно бы было вызвать `f.apply(null, arguments)`.

Ещё пример

Посмотрим еще пример. Предположим, у нас есть функция `isAdmin()`, которая возвращает `true`, если у посетителя есть права администратора.

Можно создать универсальный декоратор, который добавляет в функцию проверку прав:

Например, создадим декоратор `checkPermissionDecorator(f)`. Он будет возвращать обертку, которая передает вызов `f` в том случае, если у посетителя достаточно прав:

```
function checkPermissionDecorator(f) {
  return function() {
    if ( isAdmin() ) {
      return f.apply(this, arguments);
    }
    alert('Недостаточно прав');
  }
}
```

Использование декоратора:

```
function save() { ... }

save = checkPermissionDecorator(save);
// Теперь вызов функции save() проверяет права
```

Декораторы можно использовать в любых комбинациях:

```
sum = checkPermissionDecorator(sum);
sum = doublingDecorator(sum);
// ...
```

Зачем декораторы?

Декораторы меняют поведение функции прозрачным образом.

1. Декораторы можно повторно использовать. Например, `doublingDecorator` можно применить не только к `sum`, но и к `multiply`, `divide`. Декоратор для проверки прав можно применить к любой функции.
2. Несколько декораторов можно скомбинировать. Это придает дополнительную гибкость коду.

Примеры использования есть в задачах.

Задачи

Решения задач

Решение задачи: Создайте калькулятор

http://learn.javascript.ru/play/tutorial/intro/object/calculator.html

Решение задачи: Цепочка вызовов

Решение состоит в том, чтобы каждый раз возвращать текущий объект. Это делается добавлением `return this` в конце каждого метода:

```
var ladder = {
  step: 0,
  up: function() {
    this.step++;
    return this;
  },
  down: function() {
    this.step--;
    return this;
  },
  showStep: function() {
    alert(this.step);
    return this;
  }
}

ladder.up().up().down().up().down().showStep();  // 1
```

Решение задачи: Две функции один объект

Да, возможны.

Они должны возвращать одинаковый объект. При этом если функция возвращает объект, то `this` не используется.

Например, они могут вернуть один и тот же объект `obj`, определённый снаружи:

```
var obj = {};

function A() { return obj; }
function B() { return obj; }

var a = new A;
var b = new B;

alert( a == b ); // true
```

Решение задачи: Создать Summator при помощи конструктора

Код решения

http://learn.javascript.ru/play/tutorial/intro/object/summator2New.html

Ответ на вопрос

Если `a,b` будут свойствами — то объект `Summator` получит «состояние». Их можно будет использовать в других функциях этого же объекта.

В данном случае это, скорее, полезно. С другой стороны, поток управления примитивнее, проще, если `a,b` — локальные переменные. А проще — это хорошо.

Окончательный выбор делается в зависимости от дальнейших планов. Если имеет смысл сохранить эти переменные как состояние и использовать, то пусть будут свойства.

Решение задачи: Создать Adder при помощи конструктора

http://learn.javascript.ru/play/tutorial/intro/object/adderNew.html

Решение задачи: Перекрытие переменной

Нет, нельзя.

Локальная переменная полностью перекрывает внешнюю, обращение к ней становится невозможным.

Решение задачи: Создайте калькулятор

Решение: http://learn.javascript.ru/play/tutorial/intro/object/calculator-extendable.html

- Обратите внимание на хранение методов. Они просто добавляются к внутреннему объекту.
- Все проверки и преобразование к числу производятся в методе `calculate`. В дальнейшем он может быть расширен для поддержки более сложных выражений.

Решение задачи: Передайте все аргументы, кроме первого

```
function f(a) {
  alert(a);
  var args = [].slice.call(arguments, 1);
  g.apply(this, args);
}

function g(a, b, c) {
  alert( a + b + (c || 0) );
}

f("тест", 1, 2);
f("тест2", 1, 2, 3);
```

Решение задачи: Почему this присваивается именно так?

1. Обычный вызов функции в контексте объекта.
2. То же самое, скобки ни на что не влияют.
3. Здесь не просто вызов `obj.method()`, а более сложный вызов вида `(выражение).method()`. Такой вызов работает, как если бы он был разбит на две строки:

   ```
   f = obj.go; // сначала вычислить выражение
   f();              // потом вызвать то, что получилось
   ```

   При этом `f()` выполняется как обычная функция, без передачи `this`.
4. Здесь также слева от точки находится выражение, вызов аналогичен двум строкам.

В спецификации это объясняется при помощи специального внутреннего типа Reference Type [9] .

Если подробнее — то `obj.go()` состоит из двух операций:

1. Сначала получить свойство `obj.go`.
2. Потом вызвать его как функцию.

Но откуда на шаге 2 получить `this`? Как раз для этого операция получения свойства `obj.go` возвращает значение особого типа `Reference Type`, который в дополнение к свойству `go` содержит информацию об `obj`. Далее, на втором шаге, вызов его при помощи скобок `()` правильно устанавливает `this`.

Любые другие операции, кроме вызова, превращают `Reference Type` в обычный тип, в данном случае — функцию `go` (так уж этот тип устроен).

Поэтому получается, что `(a = obj.go)` присваивает в переменную `a` функцию `go`, уже без всякой информации об объекте `obj`.

Аналогичная ситуация и в случае `(4)`: оператор ИЛИ `||` делает из `Reference Type` обычную функцию.

Решение задачи: Проверка синтаксиса

Решение, шаг 1

Ошибка!

Попробуйте:

```
var obj = {
  go: function() { alert(this) }
}

(obj.go || 0)()  // error!
```

Причем сообщение об ошибке - *очень странное*. В большинстве браузеров это `obj is undefined`.

Дело, как ни странно, ни в самом объявлении `obj`, а в том, что после него пропущена точка с запятой.

JavaScript игнорирует перевод строки перед скобкой `(obj.go || ..)` и читает этот код как:

```
var obj = { go:... }(obj.go || 0)()
```

Интерпретатор попытается вычислить это выражение, которое обозначает вызов объекта `{ go: ... }` как функции с аргументом `(obj.go || 0)`. При этом, естественно, возникнет ошибка.

А что будет, если добавить точку с запятой?

```
obj = {
  go: function() { alert(this); }
};

(obj.go || 0)();
```

Все ли будет в порядке? Каков будет результат?

Решение, шаг 2

Результат — `window`, поскольку вызов `obj.go || 0` аналогичен коду:

```
obj = {
  go: function() { alert(this); }
};

var f = obj.go || 0; // эти две строки - аналог (obj.go || 0)();
f();  // window
```

Решение задачи: Вызов в контексте массива

Вызов `arr[2]()` — это обращение к методу объекта `obj[method]()`, в роли `obj` выступает `arr`, а в роли метода: `2`.

Поэтому, как это бывает при вызове функции как метода, функция `arr[2]` получит `this = arr` и выведет массив:

```
arr = ["a", "b"];

arr.push( function() { alert(this); } )

arr[2](); // "a","b",function
```

Решение задачи: Логирующий декоратор (1 аргумент)

Возвратим декоратор `wrapper` который будет записывать аргумент в `log` и передавать вызов в `f`:

```
function work(a) {
  /*...*/ // work - произвольная функция, один аргумент
}

function makeLogging(f, log) {

  function wrapper(a) {
    log.push(a);
    return f.call(this, a);
  }

  return wrapper;
}

var log = [];
work = makeLogging(work, log);

work(1); // 1
work(5); // 5

for(var i=0; i<log.length; i++) {
  alert( 'Лог:' + log[i] ); // "Лог:1", затем "Лог:5"
}
```

При вызове `f.call` на всякий случай передадим и `this`, ведь функция может быть вызвана и в контексте объекта.

Решение задачи: Логирующий декоратор (много аргументов)

Решение аналогично задаче Логирующий декоратор (1 аргумент) [10], разница в том, что в лог вместо одного аргумента идет весь объект `arguments`.

Для передачи вызова с произвольным количеством аргументов используем `f.apply(this, arguments)`.

```
01 function work(a, b) {
02   alert(a + b); // work - произвольная функция
03 }
04
05 function makeLogging(f, log) {
06
07   function wrapper() {
08     log.push(arguments);
09     return f.apply(this, arguments);
10   }
11
12   return wrapper;
13 }
14
15 var log = [];
16 work = makeLogging(work, log);
17
18 work(1, 2); // 3
19 work(4, 5); // 9
20
21 for(var i=0; i<log.length; i++) {
22   alert( 'Лог:' + [].join.call(log[i]) ); // "Лог:1,2", "Лог:4,5"
23 }
```

Решение задачи: Кеширующий декоратор

Запоминать результаты вызова функции будем в замыкании, в объекте `cache: { ключ:значение }`.

```
function f(x) {
  return Math.random()*x;
}

function makeCaching(f) {
  var cache = {};

  return function(x) {
    if (!(x in cache)) {
      cache[x] = f.call(this, x);
    }
    return cache[x];
  };

}

f = makeCaching(f);

var a = f(1);
var b = f(1);
alert( a == b ); // true (значение закешировано)

b = f(2);
alert( a == b ); // false, другой аргумент => другое значение
```

Обратите внимание: проверка на наличие уже подсчитанного значения выглядит так: `if (x in cache)`. Менее универсально можно проверить так: `if (cache[x])`, это если мы точно знаем, что `cache[x]` никогда не будет `false`, `0` и т.п.

Ссылки

1. Замыкания, функции изнутри http://learn.javascript.ru/closures
2. Разделе о создании объектов http://learn.javascript.ru/object-methods#new
3. 15.3 http://es5.github.com/x15.3.html#x15.3.4.3
4. Join https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/join
5. Со спецификацией http://es5.github.com/x15.4.html#x15.4.4.5
6. Arr.slice(start, end) https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/slice
7. Декоратор http://en.wikipedia.org/wiki/Decorator_pattern
8. Apply http://javascript.ru/Function/apply
9. Reference Type http://es5.github.com/x8.html#x8.7
10. Логирующий декоратор (1 аргумент) http://learn.javascript.ru/task/logiruushij-dekorator-1-argument