

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: 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

Свои методы объектов

Доступ к объекту через `this`

Функция-конструктор, «`new`»

Создание методов в конструкторе

Приватные свойства

Итого

Контекст `this` в деталях

Вызов функции с `new`

Вызов в контексте объекта

Вызов в режиме обычной функции

Явное указание this: apply и call

Метод call

Метод apply

«Одалживание метода»

Делаем из arguments настоящий Array

«Переадресация» вызова через apply

Итого

Приём программирования "Декоратор"

Пример декоратора

Ещё пример

Зачем декораторы?

Задачи

Решения задач

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:

```
01 | var user = {  
02 |   name: 'Василий',  
03 |  
04 |   // метод  
05 |   sayHi: function() {
```

```
06     alert( 'Привет!' );
07 }
08
09 };
10
11 // Вызов метода
12 user.sayHi();
```

Можно создать метод и позже, явным присвоением:

```
01 var user = {
02     name: 'Василий'
03 };
04
05 user.sayHi = function() {
06     alert( 'Привет!' );
07 };
08
09 // Вызов метода:
10 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:

```
1 var user = {
2     name: 'Василий',
3
4     sayHi: function() {
5         alert( this.name );
6     }
7 };
8
9 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();
```



Задача: Создайте калькулятор

Создайте объект `calculator` с тремя методами:

Важность: 5

- `readValues()` запрашивает `prompt` два значения и сохраняет их как свойства объекта
- `sum()` возвращает сумму двух значений
- `mul()` возвращает произведение двух значений

На демо-страничке

<http://learn.javascript.ru/files/tutorial/intro/object/calculator.html>

показан результат кода:

```
1 var calculator = {
2   ... ваш код...
3 }
4
5 calculator.readValues();
```

```
6 | alert( calculator.sum() );  
7 | alert( calculator.mul() );
```

[Решение задачи "Создайте калькулятор" »»](#)



Задача: Цепочка вызовов

Есть объект «лестница» ladder:

Важность: 2

```
01 | var ladder = {  
02 |   step: 0,  
03 |   up: function() { // вверх по лестнице  
04 |     this.step++;  
05 |   },  
06 |   down: function() { // вниз по лестнице  
07 |     this.step--;  
08 |   },  
09 |   showStep: function() { // вывести текущую  
    ступеньку  
10 |     alert(this.step);  
11 |   }  
12 | };
```

Сейчас работать с ним скучно:

```
1 | ladder.up();  
2 | ladder.up();  
3 | ladder.down();  
4 | ladder.showStep(); // 1
```

Модифицируйте код, чтобы вызовы можно было делать цепочкой, вот так:

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

Такой подход используется, например, во фреймворке jQuery.

[Решение задачи "Цепочка вызовов" »»](#)

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 вернуть другой объект вместо `this`

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

Например:

```
1 function BigAnimal() {  
2  
3   this.name = 'Мышь';  
4  
5   return { name: 'Годзилла' }; // <-- будет  
   возвращено  
6 }  
7  
8 alert( new BigAnimal().name ); // Годзилла
```

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

```
1 function BigAnimal() {  
2  
3   this.name = 'Мышь';  
4  
5   return 'Годзилла'; // не объект, такой return в  
   режиме new ни на что не влияет  
6 }  
7  
8 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();
```



Задача: Две функции один объект

Возможны ли такие функции A и B в примере ниже, что соответствующие объекты a, b равны (см. код ниже)?

Важность: 2

```
1 function A() { ... }
2 function B() { ... }
3
4 var a = new A;
5 var b = new B;
6
7 alert( a == b ); // true
```

Если да — приведите пример кода с такими функциями.

[Решение задачи "Две функции один объект" »»](#)

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`:

```
1 function User(name) {
2   this.name = name;
3
4   this.sayHi = function() {
5     alert("Моё имя: " + this.name);
6   };
7 }
```

Пример использования:

```
01 var ivan = new User("Иван");
02
03 /* Объект ivan имеет вид:
```



```
04 {  
05   name: "Иван",  
06   sayHi: функция, обращение к имени идёт через this.name  
07 }  
08 */  
09  
10 ivan.sayHi(); // Моё имя: Иван
```

 **Задача:** Создать Summator при помощи конструктора

Напишите *функцию-конструктор* Summator, которая Важность: 5
создает объект с двумя методами:

- `sum(a,b)` возвращает сумму двух значений
- `run()` запрашивает два значения при помощи `prompt` и выводит их сумму, используя метод `sum`.

В итоге вызов `new Summator().run()` должен спрашивать два значения и выводить их сумму, вот так:


<http://learn.javascript.ru/files/tutorial/intro/object/summator2New.html>

Следует ли в вызове `run` сохранять введенные значения как свойства объекта?

При этом метод `sum()` будет без параметров, и обращения в нём будут к `this.a`, `this.b`.

[Решение задачи "Создать Summator при помощи конструктора"](#)
»»

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

 **Задача:** Создать Adder при помощи конструктора

Напишите *функцию-конструктор* Важность: 5
`Adder(startingValue)`.

Объекты, которые она создает, должны хранить текущую сумму и прибавлять к ней то, что вводит посетитель.

Более формально, объект должен:

- Хранить текущее значение в своём свойстве `value`.
Начальное значение свойства `value` ставится конструктором равным `startingValue`.
- Метод `addInput()` вызывает `prompt`, принимает число и прибавляет его к свойству `value`.
- Метод `showValue()` выводит текущее значение `value`.

Таким образом, свойство `value` является текущей суммой всего, что ввел посетитель при вызовах метода `addInput()`, с учетом начального значения `startingValue`.

По ссылке ниже вы можете посмотреть работу кода:

```
1 var adder = new Adder(1); // начальное значение 1
2 adder.addInput(); // прибавит ввод prompt к
  текущему значению
3 adder.addInput(); // прибавит ввод prompt к
  текущему значению
4 adder.showValue(); // выведет текущее значение
```

Демо результата:

<http://learn.javascript.ru/files/tutorial/intro/object/adderNew.html>

[Решение задачи "Создать Adder при помощи конструктора" »»](#)

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`.

```
1 function User(name) {
2
3   this.say = function(phrase) {
4     alert(name + ' сказал: ' + phrase);
```

```

5     };
6
7 }
8
9 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](#) мы говорили о скрытых

ссылках `[[Scope]]` на внешний объект переменных? В этом примере `user.toUpperCaseName. [[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.



Задача: Перекрытие переменной

Можно ли из `this.sayHi` получить доступ к `name` из `User` и при этом обойтись без переименований?

Важность: 4

```
1 function User(name) {  
2  
3   this.sayHi = function(name) {  
4     // ваш код... ?  
5   }  
6 }
```

[Решение задачи "Перекрытие переменной" »»](#)



Задача: Создайте калькулятор

Напишите конструктор `Calculator`, который создаёт расширяемые объекты-калькуляторы.

Важность: 5

1. Первый шаг задачи: вызов `calculate(str)` принимает строку, например «1 + 2», с жёстко заданным форматом «ЧИСЛО операция ЧИСЛО» (по одному пробелу вокруг операции), и возвращает результат. Понимает плюс + и минус -.

Пример использования:

```
var calc = new Calculator;  
alert(calc.calculate("3 + 7")); // 10
```

2. Второй шаг — добавить метод `addMethod(name, func)`,

который учит калькулятор новой операции. Он получает имя операции `name` и функцию от двух аргументов `func(a, b)`, которая должна её реализовывать.

Например, добавим операции умножить `*`, поделить `/` и возвести в степень `**`:

```
1 var powerCalc = new Calculator;
2 powerCalc.addMethod("*", function(a, b) { return
  a * b; });
3 powerCalc.addMethod("/", function(a, b) { return
  a / b; });
4 powerCalc.addMethod("**", function(a, b) {
  return Math.pow(a, b); });
5
6 var result = powerCalc.calculate("2 ** 3");
7 alert(result); // 8
```

- Поддержка скобок и более сложных математических выражений в этой задаче не требуется.
- Числа и операции могут состоять из нескольких символов. Между ними ровно один пробел.
- Предусмотрите обработку ошибок. Какая она должна быть - решите сами.

Решение задачи "Создайте калькулятор" »»

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 вернуть 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](#).

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`:

```

01 var user = { firstName: "Вася" };
02 var admin = { firstName: "Админ" };
03
04 function func() {
05     alert( this.firstName );
06 }
07
08 user.a = func; // присвоим одну функцию в свойства
09 admin.b = func; // двух разных объектов user и admin
10
11 user.a(); // Вася
12 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*.

```
1 function func() {
2   alert(this); // выведет [object Window] или [object
   global]
3 }
4
5 func();
```

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

```
1 function func() {
2   "use strict";
3   alert(this); // выведет undefined (кроме IE<10)
4 }
5
6 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`:

```
01 var user = {
02   firstName: "Василий",
03   lastName: "Петров"
```



```

04 };
05
06 function showName() {
07     alert( this.firstName + ' ' + this.lastName );
08 }
09
10 showName.call(user) // "Василий Петров"

```

Можно сделать её более универсальной, добавив аргументы:

```

01 var user = {
02     firstName: "Василий",
03     surname: "Петров"
04 };
05
06 function getName(a, b) {
07     alert( this[a] + ' ' + this[b] );
08 }
09
10 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:

```
1 var args = [];  
2 args.push('firstName');  
3 args.push('surname');  
4  
5 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`:

```
1 function f() { alert(this) }  
2  
3 f.call(null); // window
```

Это поведение исправлено в современном стандарте (15.3).
Если функция работает в строгом режиме, то `this` передаётся «как есть»:

```
1 function f() {  
2   "use strict";  
3  
4   alert(this); // null, "как есть"  
5 }  
6  
7 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 «одолживание 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.

```
1 function sayHi() {
2   arguments.join = [].join; // одолжили метод (1)
3
4   var argStr = arguments.join(':'); // (2)
5
6   alert(argStr); // сработает и выведет 1:2:3
7 }
8
9 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` 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](#), внутри `join` реализован примерно так:

```
01 function join(separator) {
02   if (!this.length) return '';
03
04   var str = this[0];
05
06   for (var i = 1; i < this.length; i++) {
07     str += separator + this[i];
08   }
09
10   return str;
```

```
11 | }
```

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:

```
1 | var obj = { // обычный объект с числовыми
  |   индексами и length
2 |   0: "А",
3 |   1: "Б",
4 |   2: "В",
5 |   length: 3
6 | };
7 |
8 | obj.join = [].join;
9 | 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`:

```
01 | function sayHi() {
02 |   var join = [].join; // ссылка на функцию теперь в
   |   переменной
03 |
04 |   // вызовем join с this=arguments,
05 |   // этот вызов эквивалентен arguments.join(':') из примера
   |   выше
06 |   var argStr = join.call(arguments, ':');
07 |
08 |   alert(argStr); // сработает и выведет 1:2:3
09 | }
10 |
11 | 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)` .

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:

```
1 function sayHi() {  
2   // вызов arr.slice() скопирует все элементы из this в  
   // новый массив  
3   var args = [].slice.call(arguments);  
4  
5   alert( args.join(':') ); // args -- массив аргументов  
6 }  
7  
8 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.



Задача: Передайте все аргументы, кроме первого

Напишите функцию `f`, которая будет обёрткой вокруг другой функции `g`. Функция `f` обрабатывает первый аргумент сама, а остальные аргументы передаёт в функцию `g`, сколько бы их ни было.

Важность: 5

Например:

```
1 function f() { ... }
2
3 function g(a, b, c) {
4   alert( (a || 0) + (b || 0) + (c || 0) );
5 }
6
7 f("тест", 1, 2); // f выведет "тест", дальше g
                  // посчитает сумму "3"
8 f("тест2", 1, 2, 3); // f выведет "тест2", дальше g
                  // посчитает сумму "6"
```

Код функции `f` не должен зависеть от количества аргументов.

[Решение задачи "Передайте все аргументы, кроме первого" »»](#)



Задача: Почему `this` присваивается именно так?

Вызовы (1) и (2) в примере ниже работают не так, как (3) и (4):

Важность: 3

```
01 "use strict"
02
03 var obj, f;
04
05 obj = {
06   go: function() { alert(this); }
07 };
08
09 obj.go();           // (1) object
10
11 (obj.go)();         // (2) object
12
13 (f = obj.go)();     // (3) undefined
14
15 (obj.x || obj.go)(); // (4) undefined
```

В чём дело? Объясните логику работы `this`.

[Решение задачи "Почему `this` присваивается именно так?" »»](#)



Задача: Проверка синтаксиса

Каков будет результат этого кода?

Важность: 2

```
1 var obj = {  
2   go: function() { alert(this) }  
3 }  
4  
5 (obj.go || 0)()
```

P.S. Есть подвох.

[Решение задачи "Проверка синтаксиса" »»](#)



Задача: Вызов в контексте массива

Каким будет результат? Почему?

Важность: 5

```
1 arr = ["a", "b"];  
2  
3 arr.push( function() { alert(this); } )  
4  
5 arr[2](); // ?
```

[Решение задачи "Вызов в контексте массива" »»](#)

Итого

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

При вызове функции как метода

```
obj.func(...)  
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 — 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` 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:

```

1 function checkPermissionDecorator(f) {
2     return function() {
3         if ( isAdmin() ) {
4             return f.apply(this, arguments);
5         }
6         alert( 'Недостаточно прав' );
7     }
8 }

```

Использование декоратора:

```
1 function save() { ... }  
2  
3 save = checkPermissionDecorator(save);  
4 // 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



Задача: Логирующий декоратор (1 аргумент)

Создайте декоратор `makeLogging(f, log)`, который берет функцию `f` и массив `log`. Важность: 5

Он должен возвращать обёртку вокруг `f`, которая при каждом вызове записывает («логирует») аргументы в `log`, а затем передает вызов в `f`.

В этой задаче можно считать, что у функции `f` ровно один аргумент.

Работать должно так:

```

01 function work(a) {
02     /* ... */ // work - произвольная функция, один
    аргумент
03 }
04
05 function makeLogging(f, log) { /* ваш код */ }
06
07 var log = [];
08 work = makeLogging(work, log);
09
10 work(1); // 1, добавлено в log
11 work(5); // 5, добавлено в log
12
13 for(var i=0; i<log.length; i++) {
14     alert( 'Лог:' + log[i] ); // "Лог:1", затем
    "Лог:5"
15 }

```

Решение задачи "Логирующий декоратор (1 аргумент)" »»



Задача: Логирующий декоратор (много аргументов)

Создайте декоратор `makeLogging(func, log)`, для функции `func` возвращающий обёртку, которая при каждом вызове добавляет её аргументы в массив `log`. Важность: 3

Условие аналогично задаче [Логирующий декоратор \(1 аргумент\)](#), но допускается `func` с любым набором аргументов.

Работать должно так:

```

01 function work(a, b) {
02     alert(a + b); // work - произвольная функция
03 }
04
05 function makeLogging(f, log) { /* ваш код */ }
06
07 var log = [];
08 work = makeLogging(work, log);
09
10 work(1, 2); // 3
11 work(4, 5); // 9
12
13 for(var i=0; i<log.length; i++) {

```

```
14     alert( 'Лог:' + [].join.call(log[i]) ); //  
    "Лог:1,2", "Лог:4,5"  
15 }
```

[Решение задачи "Логирующий декоратор \(много аргументов\)" »»](#)



Задача: Кеширующий декоратор

Создайте декоратор `makeCaching(f)`, который берет **Важность: 5** функцию `f` и возвращает обертку, которая кеширует её результаты.

В этой задаче функция `f` имеет только один аргумент, и он является числом.

1. При первом вызове обертки с определенным аргументом — она вызывает `f` и запоминает значение.
2. При втором и последующих вызовах с тем же аргументом возвращается запомненное значение.

Должно работать так:

```
01 function f(arg) {  
02     return Math.random()*arg; // может быть любой  
    функцией  
03 }  
04  
05 function makeCaching(f) { /* ваш код */ }  
06  
07 f = makeCaching(f);  
08  
09 var a, b;  
10  
11 a = f(1);  
12 b = f(1);  
13 alert( a == b ); // true (значение закешировано)  
14  
15 b = f(2);  
16 alert( a == b ); // false, другой аргумент =>  
    другое значение
```

[Решение задачи "Кеширующий декоратор" »»](#)

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:

```
01 var ladder = {  
02   step: 0,  
03   up: function() {  
04     this.step++;  
05     return this;  
06   },  
07   down: function() {  
08     this.step--;  
09     return this;  
10   },  
11   showStep: function() {  
12     alert(this.step);  
13     return this;  
14   }  
15 }  
16  
17 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:

```
1 var obj = {};  
2  
3 function A() { return obj; }  
4 function B() { return obj; }  
5  
6 var a = new A;  
7 var b = new B;  
8  
9 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

```
01 function f(a) {  
02     alert(a);  
03     var args = [].slice.call(arguments, 1);  
04     g.apply(this, args);  
05 }  
06  
07 function g(a, b, c) {  
08     alert( (a || 0) + (b || 0) + (c || 0) );  
09 }  
10  
11 f("тест", 1, 2);  
12 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`.

- Здесь также слева от точки находится выражение, вызов аналогичен двум строкам.

В спецификации это объясняется при помощи специального внутреннего типа **Reference Type**.

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

- Сначала получить свойство `obj.go`.
- 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:


```
1 var obj = {  
2   go: function() { alert(this) }  
3 }  
4  
5 (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?

```
1 obj = {  
2   go: function() { alert(this); }  
3 };  
4  
5 (obj.go || 0)();
```

Все ли будет в порядке? Каков будет результат?

Решение, шаг 2

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

```
1 obj = {  
2   go: function() { alert(this); }  
3 };  
4  
5 var f = obj.go || 0; // эти две строки - аналог  
   (obj.go || 0)();
```

```
6 | 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:

```
1 | arr = ["a", "b"];
2 |
3 | arr.push( function() { alert(this); } )
4 |
5 | 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`:

```
01 | function work(a) {
02 |     /*...*/ // work - произвольная функция, один
    | аргумент
03 | }
04 |
05 | function makeLogging(f, log) {
06 |
07 |     function wrapper(a) {
08 |         log.push(a);
09 |         return f.call(this, a);
10 |     }
11 |
12 |     return wrapper;
13 | }
14 |
15 | var log = [];
16 | work = makeLogging(work, log);
17 |
18 | work(1); // 1
19 | work(5); // 5
20 |
```

```
21 for(var i=0; i<log.length; i++) {  
22     alert( 'Лог:' + log[i] ); // "Лог:1", затем  
    "Лог:5"  
23 }
```

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 \(1 аргумент\)](#), 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 }.

```
01 function f(x) {
02   return Math.random()*x;
03 }
04
05 function makeCaching(f) {
06   var cache = {};
07
08   return function(x) {
09     if (!(x in cache)) {
10       cache[x] = f.call(this, x);
11     }
12     return cache[x];
13   };
14 }
15
16
17 f = makeCaching(f);
18
19 var a = f(1);
20 var b = f(1);
21 alert( a == b ); // true (значение закешировано)
22
23 b = f(2);
24 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.