

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: Получение и проверка типа

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

Содержание

Преобразование объектов: `toString` и `valueOf`

- Строковое преобразование

- Численное преобразование

- Преобразование в примитив

- Итого

Оператор `typeof`, `[[Class]]` и утиная типизация

- Оператор `typeof`

- `[[Class]]` для встроенных объектов

- «Утиная» типизация

- Проверка типа для пользовательских объектов

- Полиморфизм

- Итого

Решения задач

Преобразование объектов: `toString` и `valueOf`

Ранее, в главе [Преобразование типов для примитивов \[1\]](#) мы рассматривали преобразование типов, уделяя основное внимание примитивам. Теперь добавим в нашу стройную картину преобразования типов объекты.

При этом будут следующие изменения:

1. В объектах численное и строковое преобразования можно переопределить.
2. Если операция требует примитивное значение, то сначала объект преобразуется к примитиву, а затем — всё остальное. При этом для преобразования к примитиву используется численное преобразование.

Для того, чтобы лучше интегрировать их в общую картину, рассмотрим примеры.

Строковое преобразование

Строковое преобразование проще всего увидеть, если вывести объект при помощи alert:

```
1 var user = {
2   firstName: 'Василий'
3 };
4
5 alert(user); // [object Object]
```

Как видно, содержимое объекта не вывелось. Это потому, что стандартным строковым представлением пользовательского объекта является строка "[object Object]".

Такой вывод объекта не содержит интересной информации. Поэтому имеет смысл его поменять на что-то более полезное.

Если в объекте присутствует метод toString, который возвращает примитив, то он используется для преобразования.

```
01 var user = {
02
03   firstName: 'Василий',
04
05   toString: function() {
06     return 'Пользователь ' + this.firstName;
07   }
08 };
09
10 alert( user ); // Пользователь Василий
```



Результатом toString может быть любой примитив

Метод toString не обязан возвращать именно строку.

Его результат может быть любого примитивного типа. Например, это может быть число, как в примере ниже:

```
1 var obj = {
2   toString: function() { return 123; }
3 };
4
5 alert(obj); // 123
```

Поэтому мы и называем его здесь «*строковое преобразование*», а не «преобразование к строке».

Все объекты, включая встроенные, имеют свои реализации метода toString, например:

```
1 alert( [1,2] ); // toString для массивов выводит список элементов "1,2"
2 alert( new Date ); // toString для дат выводит дату в виде строки
3 alert( function() { } ); // toString для функции выводит её код
```

Численное преобразование

Для численного преобразования объекта используется метод `valueOf`, а если его нет — то `toString`:

```
01 var room = {
02   number: 777,
03
04   valueOf: function() { return this.number; },
05   toString: function() { return this.number; }
06 };
07
08 alert( +room ); // 777, вызвался valueOf
09
10 delete room.valueOf;
11
12 alert( +room ); // 777, вызвался toString
```

Метод `valueOf` обязан возвращать примитивное значение, иначе его результат будет проигнорирован. При этом — не обязательно числовое.

У большинства встроенных объектов такого `valueOf` нет, поэтому численное и строковое преобразования для них работают одинаково.

Исключением является объект `Date`, который поддерживает оба типа преобразований:

```
1 alert( new Date() ); // toString: Дата в виде читаемой строки
2 alert( +new Date() ); // valueOf: кол-во миллисекунд, прошедших с 01.01.1970
```



Стандартный `valueOf`

Если посмотреть в стандарт, то в [15.2.4.4 \[2\]](#) определён `valueOf` для любых объектов. Но он ничего не делает, просто возвращает сам объект (не-примитивное значение!), а потому игнорируется.

Преобразование в примитив

Большинство операций, которые ожидают примитивное значение, при виде объекта сначала преобразуют его к примитиву.

Например, арифметическая операция или сравнение `>` `<` `>=` `<=` сначала преобразует объект в примитив.

А затем уже идёт операция с примитивами, при этом возможны последующие преобразования.

При приведении объекта к примитиву используется численное преобразование.

Например:

```
1 var obj = {
2   valueOf: function() { return 1; }
3 };
4
5 // операции сравнения, кроме ==, приводят к примитиву
6 alert(obj == true); // true, объект приведён к примитиву 1
7
8 // бинарный + приводит к примитиву, а затем складывает
9 alert(obj + "test"); // 1test
```

Пример ниже демонстрирует, что несмотря на то, что приведение «численное» — его результатом может быть любое примитивное значение, не обязательно число:

```
1 var a = {  
2   valueOf: function() { return "1"; }  
3 };  
4 var b = {  
5   valueOf: function() { return true; }  
6 };  
7  
8 alert(a + b); // "1" + true = "1true"
```

После того, как объект приведён к примитиву, могут быть дополнительные преобразования. Например:

```
1 var a = {  
2   valueOf: function() { return "1"; }  
3 };  
4 var b = {  
5   valueOf: function() { return "2"; }  
6 };  
7  
8 alert(a - b); // "1" - "2" = -1
```



Исключение: Date

Существует исключение: объект Date преобразуется в примитив, используя строковое преобразование. С этим можно столкнуться в операторе "+":

```
1 // бинарный вариант, преобразование к примитиву  
2 alert( new Date + "" ); // "строка даты"  
3  
4 // унарный вариант, наравне с - * / и другими приводит к числу  
5 alert( +new Date ); // число миллисекунд
```

Это исключение явно прописано в стандарте и является единственным в своём роде.



Как испугать Java-разработчика

В языке Java логические значения можно создавать, используя синтаксис `new Boolean(true/false)`. Также можно преобразовывать значения к логическому типу, применяя к ним `new Boolean`.

В JavaScript тоже есть подобная возможность, которая возвращает «объектную обёртку» для логического значения. Эта возможность сохраняется для совместимости и не используется на практике, поскольку приводит к странным результатам.

Например:

```
1 | var value = new Boolean(false);
2 | if ( value ) {
3 |     alert(true); // сработает!
4 | }
```

Почему запустился alert? Ведь в if находится false... Проверим:

```
1 | var value = new Boolean(false);
2 |
3 | alert(value); // выводит false, все ок..
4 |
5 | if ( value ) {
6 |     alert(true); // ..но тогда почему выполняется alert в if ???
7 | }
```

Дело в том, что `new Boolean` - это объект. В логическом контексте он, безусловно, `true`. Поэтому работает первый пример.

А второй пример вызывает alert, который преобразует объект к строке, и он становится `"false"`.

Чтобы преобразовать значение к логическому типу, нужно использовать двойное отрицание: `!!val` или прямой вызов `Boolean(val)`.

Итого

- ➡ При строковом преобразовании объекта используется его метод `toString`. Он должен возвращать примитивное значение, причём не обязательно именно строку.

В стандарте прописано, что если `toString` нет, или он возвращает не примитив, а объект, то вызывается `valueOf`, но обычно `toString` есть.

- ➡ При численном преобразовании объекта используется метод `valueOf`, а если его нет, то `toString`. У встроенных объектов `valueOf` обычно нет.
- ➡ При операции над объектом, которая требует примитивное значение, объект первым делом преобразуется в примитив. Для этого используется численное преобразование, исключение — встроенный объект `Date`.

Полный алгоритм преобразований есть в спецификации ECMAScript, смотрите пункты [11.8.5 \[3\]](#), [11.9.3 \[4\]](#), а также [9.1 \[5\]](#) и [9.3 \[6\]](#).

Оператор typeof, [[Class]] и утиная типизация

В этой главе мы рассмотрим, как создавать *полиморфные* функции, то есть такие, которые по-разному обрабатывают аргументы, в зависимости от их типа. Например, функция вывода может по-разному форматировать числа и даты.

Для реализации такой возможности нужен способ определить тип переменной. И здесь в JavaScript есть целый «зоопарк» способов, но мы в нём сейчас разберемся.

Как мы знаем, существует несколько *примитивных типов*:

null

Специальный тип, содержит только значение `null`.

undefined

Специальный тип, содержит только значение `undefined`.

number

Числа: 0, 3.14, а также значения NaN и Infinity

boolean

true, false.

string

Строки, такие как "Мяу" или пустая строка "".

Все остальные значения являются **объектами**, включая функции и массивы.

Оператор typeof

Оператор `typeof` возвращает тип аргумента. У него есть два синтаксиса:

1. Синтаксис оператора: `typeof x`.
2. Синтаксис функции: `typeof(x)`.

Работают они одинаково, но первый синтаксис короче.

Результатом `typeof` является строка, содержащая тип:

```
01 typeof undefined // "undefined"
02
03 typeof 0          // "number"
04
05 typeof true       // "boolean"
06
07 typeof "foo"      // "string"
08
09 typeof {}         // "object"
10
11 typeof null       // "object"
12 typeof function(){} // "function"
```

Последние две строки помечены, потому что `typeof` ведет себя в них по-особому.

1. Результат `typeof null == "object"` — это официально признанная ошибка в языке, которая сохраняется для совместимости.

На самом деле `null` — это не объект, а примитив. Это сразу видно, если попытаться присвоить ему свойство:

```
1 var x = null;
2 x.prop = 1; // ошибка, т.к. нельзя присвоить свойство примитиву
```

2. Для функции `f` значением `typeof f` является `"function"`. Конечно же, функция является объектом. С другой стороны, такое выделение функций на практике скорее плюс, т.к. позволяет легко определить функцию.



Не используйте `typeof` для проверки переменной

В старом коде можно иногда увидеть код вроде такого:

```
if (typeof jQuery !== 'undefined') {
  ...
}
```

Его автор, видимо, хочет проверить, существует ли переменная `jQuery`. Причём, он имеет в виду именно глобальную переменную `jQuery`, которая создаётся во внешнем скрипте, так как про свои локальные он и так всё знает.

Более короткий код `if (jQuery)` выдаст ошибку, если переменная не определена, а `typeof jQuery` в таких случаях ошибку не выдаёт, а возвращает `undefined`.

Но как раз здесь `typeof` не нужен! Есть другой способ:

```
1 if (window.jQuery !== undefined) { ... }
2
3 //а если мы знаем, что это объект, то проверку можно упростить:
4 if (window.jQuery) { ... }
```

При доступе к глобальной переменной через `window` не будет ошибки, ведь по синтаксису это — обращение к свойству объекта, а такие обращения при отсутствующем свойстве просто возвращают `undefined`.

Оператор `typeof` надёжно работает с примитивными типами, кроме `null`, а также с функциями. Но обычные объекты, массивы и даты для `typeof` все на одно лицо, они имеют тип `'object'`:

```
1 alert( typeof {} ); // 'object'
2 alert( typeof [] ); // 'object'
3 alert( typeof new Date ); // 'object'
```

Поэтому различить их при помощи `typeof` нельзя.

[[Class]] для встроенных объектов

Основная проблема `typeof` — неумение различать объекты, кроме функций. Но есть и другой способ получения типа.

У всех встроенных объектов есть скрытое свойство `[[Class]]`. Оно равно `"Array"` для массивов, `"Date"` для дат и т.п.

Это свойство нельзя получить напрямую, есть трюк для того, чтобы прочесть его.

Дело в том, что `toString` от стандартного объекта выводит `[[Class]]` в небольшой обертке. Например:

```
1 var obj = {};  
2 alert( obj ); // [object Object]
```

Здесь внутри `[object ...]` указано как раз значение `[[Class]]`, которое для обычного объекта как раз и есть "Object". Для дат оно будет Date, для массивов — Array и т.п.

Большинство встроенных объектов в JavaScript имеют свой собственный метод `toString`. Поэтому мы будем использовать технику, которая называется «*одалживание метода*» («*method borrowing*»).

Мы возьмем функцию `toString` от стандартного объекта и будем запускать его в контексте тех значений, для которых нужно получить тип:

```
01 var toClass = {}.toString; // (1)  
02  
03 var arr = [1,2];  
04 alert( toClass.call(arr) ); // (2) [object Array]  
05  
06 var date = new Date;  
07 alert( toClass.call(date) ); // [object Date]  
08  
09 var type = toClass.call(date).slice(8, -1); // (3)  
10 alert(type); // Date
```

Разберем происходящее более подробно.

1. Можно переписать эту строку в две:

```
var obj = {};  
var toClass = obj.toString;
```

Иначе говоря, мы создаём пустой объект `{}` и копируем ссылку на его метод `toString` в переменную `toClass`.

Мы делаем это, потому что **внутренняя реализация `toString` стандартного объекта `Object` возвращает `[[Class]]`**. У других объектов (Date, Array и т.п.) `toString` свой и для этой цели не подойдёт.

2. Вызываем скопированный метод в контексте нужного объекта `obj`.

Мы могли бы поступить проще:

```
1 var arr = [1,2];  
2 arr.toClass = {}.toString;  
3  
4 alert( arr.toClass() ); // [object Array]
```

...Но зачем копировать лишнее свойство в объект? Синтаксис `toClass.call(arr)` делает то же самое, поэтому используем его.

3. Всё, класс получен. При желании можно убрать обёртку `[object ...]`, взяв подстроку вызовом `slice(8, -1)`.

Метод также работает с примитивами:

```
1 alert( {}.toString.call(123) ); // [object Number]  
2 alert( {}.toString.call("строка") ); // [object String]
```

...Но без `use strict` вызов `call` с аргументами `null` или `undefined` передает `this = window`. Таково поведение старого стандарта JavaScript.

Этот метод может дать тип только для встроенных объектов. Для пользовательских конструкторов всегда

[[Class]] = "Object":

```
1 function Animal(name) {  
2   this.name = name;  
3 }  
4 var animal = new Animal("Винни-пух");  
5  
6 var type = {}.toString.call( animal );  
7  
8 alert(type); // [object Object]
```



Вызов `{}.toString` в консоли может выдать ошибку

При тестировании кода в консоли вы можете обнаружить, что если ввести в командную строку `{}.toString.call(...)` — будет ошибка. С другой стороны, вызов `alert({}.toString...)` — работает.

Эта ошибка возникает потому, что фигурные скобки `{ }` в основном потоке кода интерпретируются как блок. Интерпретатор читает `{}.toString.call(...)` так:

```
{ } // пустой блок кода  
.toString.call(...) // а что это за точка в начале? не понимаю, ошибка!
```

Фигурные скобки считаются объектом, только если они находятся в контексте выражения. В частности, оборачивание в скобки `({}.toString...)` тоже работает нормально.

«Утиная» типизация

Утиная типизация основана на одной известной поговорке: *«If it looks like a duck, swims like a duck and quacks like a duck, then it probably is a duck (who cares what it really is)»*.

В переводе: *«Если это выглядит как утка, плавает как утка и крякает как утка, то, вероятно, это утка (какая разница, что это на самом деле)»*.

Смысл утиной типизации — в проверке методов и свойств, безотносительно типа объекта.

Проверить массив мы можем, уточнив наличие метода `splice`:

```
1 var x = [1,2,3];  
2  
3 if (x.splice) {  
4   alert('Массив!');  
5 }
```

Обратите внимание — в `if(x.splice)` мы не вызываем метод `x.splice()`, а пробуем получить само свойство `x.splice`. Для массивов оно всегда есть и является функцией, т.е. даст в логическом контексте `true`.

Проверить на дату можно, проверив наличие метода `getTime`:

```
1 var x = new Date();
2
3 if (x.getTime) {
4     alert('Дата!');
5 }
```

С виду такая проверка хрупка, ее можно сломать, передав похожий объект с тем же методом. Но на практике утиная типизация хорошо и стабильно работает, особенно когда важен не сам тип, а поддержка методов.

Проверка типа для пользовательских объектов

Для проверки, кем был создан объект, есть оператор `instanceof`.

Синтаксис: `obj instanceof Func`.

Например:

```
1 function Animal(name) {
2     this.name = name;
3 }
4 var animal = new Animal("Винни-пух");
5
6 alert( animal instanceof Animal ); // true
```

Оператор `instanceof` также работает для встроенных объектов:

```
1 var d = new Date();
2 alert(d instanceof Date); // true
3
4 function f() { }
5 alert(f instanceof Function); // true
```

Оператор `instanceof` может лишь осуществить проверку, он не позволяет получить тип в виде строки, но в большинстве случаев этого и не требуется.



Почему `[[Class]]` надежнее `instanceof`?

Как мы видели, `instanceof` успешно работает со встроенными объектами:

```
1 var arr = [1,2,3];
2 alert(arr instanceof Array); // true
```

...Однако, есть случай, когда такой способ нас подведет. А именно, если объект создан в другом окне или `iframe`, а оттуда передан в текущее окно.

При этом `arr instanceof Array` вернет `false`, т.к. **в каждом окне и фрейме — свой собственный набор встроенных объектов**. Массив `arr` является `Array` в контексте *того window*.

Метод `[[Class]]` свободен от этого недостатка.

Полиморфизм

Используем проверку типов для того, чтобы создать полиморфную функцию sayHi.

Она будет работать в трёх режимах:

1. Без аргументов: выводит Привет.
2. С аргументом, который не является массивом: выводит «привет» и этот аргумент.
3. С аргументом, который является массивом — говорит всем «привет».

Пример такой функции:

```
01 function sayHi(who) {
02   if (!arguments.length) {
03     alert('Привет');
04     return;
05   }
06
07   if ( {}.toString.call(who) == '[object Array]' ) {
08     for(var i=0; i<who.length; i++) sayHi(who[i]);
09     return;
10   }
11
12   alert('Привет, ' + who);
13 }
14
15 // Использование:
16 sayHi(); // Привет
17 sayHi("Вася"); // Привет, Вася
18
19 sayHi( ["Саша", "Петя", ["Маша", "Юля"] ] ); // Привет Саша..Петя..Маша..Юля
```

Обратите внимание, получилась даже поддержка вложенных массивов 😊

Итого

Для *получения типа* есть два способа:

typeof

Хорош для примитивов и функций, врёт про null.

Свойство [[Class]]

Можно получить, используя `{}.toString.call(obj)`. Это свойство содержит тип для встроенных объектов и примитивов, кроме null и undefined.

Для *проверки типов* есть еще два способа:

Утиная типизация

Можно проверить поддержку метода.

Оператор instanceof

Работает с любыми объектами, встроенными и созданными посетителем при помощи конструкторов:

```
if (obj instanceof User) { ... }.
```

Используется проверка типа, как правило, для создания полиморфных функций, то есть таких, которые по-разному работают в зависимости от типа аргумента.

Решения задач



Решение задачи: ['x'] == 'x'

Если с одной стороны — объект, а с другой — нет, то сначала приводится объект.

В данном случае сравнение означает численное приведение. У массивов нет `valueOf`, поэтому вызывается `toString`, который возвращает список элементов через запятую.

В данном случае, элемент только один - он и возвращается. Так что `['x']` становится `'x'`. Получилось `'x' == 'x'`, верно.

P.S.

По той же причине верны равенства:

```
1 alert( ['x','y'] == 'x,y' ); // true
2 alert( [] == '' ); // true
```



Решение задачи: Преобразование

Ответы, один за другим.

`alert(foo)`

Возвращает строковое представление объекта, используя `toString`, т.е. `"foo"`.

`alert(foo + 1)`

Оператор `+` преобразует объект к примитиву, используя `valueOf`, так что результат: 3.

`alert(foo + '3')`

То же самое, что и предыдущий случай, объект превращается в примитив 2. Затем происходит сложение `2 + '3'`. Оператор `+` при сложении чего-либо со строкой приводит и второй операнд к строке, а затем применяет конкатенацию, так что результат — строка `"23"`.



Решение задачи: Почему `[] == []` неверно, а `[] == ![]` верно?

Ответ по первому равенству

Два объекта равны только тогда, когда это один и тот же объект.

В первом равенстве создаются два массива, это разные объекты, так что они неравны.

Ответ по второму равенству

1. Первым делом, обе части сравнения вычисляются. Справа находится `![]`. Логическое НЕ `!` преобразует аргумент к логическому типу. Массив является объектом, так что это `true`. Значит, правая часть становится `![] = !true = false`. Так что получили:

```
alert( [] == false );
```

2. Проверка равенства между объектом и примитивом вызывает численное преобразование объекта.

У массива нет `valueOf`, сработает `toString` и преобразует массив в список элементов, то есть - в пустую строку:

```
alert( '' == false );
```

3. Сравнение различных типов вызывает численное преобразование слева и справа:

```
alert( 0 == 0 );
```

Теперь результат очевиден.



Решение задачи: Вопросник по преобразованиям, для объектов

```
1 new Date(0) - 0 = 0 // (1)
2 new Array(1)[0] + "" = "undefined" // (2)
3 ({})[0] = undefined // (3)
4 [1] + 1 = "11" // (4)
5 [1,2] + [3,4] = "1,23,4" // (5)
6 [] + null + 1 = "null1" // (6)
7 [[0]][0][0] = 0 // (7)
8 ({ } + { }) = "[object Object][object Object]" // (8)
```

1. `new Date(0)` — дата, созданная по миллисекундам и соответствующая 0мс от 1 января 1970 года 00:00:00 UTC. Оператор минус - преобразует дату обратно в число миллисекунд, то есть в 0.
2. `new Array(num)` при вызове с единственным аргументом-числом создаёт массив данной длины, без элементов. Поэтому его нулевой элемент равен `undefined`, при сложении со строкой получается строка `"undefined"`.
3. Фигурные скобки — это создание пустого объекта, у него нет свойства `'0'`. Так что значением будет `undefined`. Обратите внимание на внешние, круглые скобки. Если их убрать и запустить `{ }[0]` в отладочной консоли браузера — будет 0, т.к. скобки `{ }` будут восприняты как пустой блок кода, после которого идёт массив.
4. Массив преобразуется в строку `"1"`. Оператор `+` при сложении со строкой приводит второй аргумент к строке — значит будет `"1" + "1" = "11"`.
5. Массивы приводятся к строке и складываются.
6. Массив преобразуется в пустую строку `"" + null + 1`, оператор `+` видит, что слева строка и преобразует `null` к строке, получается `"null" + 1`, и в итоге `"null1"`.
7. `[[0]]` — это вложенный массив `[0]` внутри внешнего `[]`. Затем мы берём от него нулевой элемент, и потом еще раз.

Если это непонятно, то посмотрите на такой пример:

```
alert( [1,[0],2][1] );
```

Квадратные скобки после массива/объекта обозначают не другой массив, а взятие элемента.

8. Каждый объект преобразуется к примитиву. У встроенных объектов `Object` нет подходящего `valueOf`, поэтому используется `toString`, так что складываются в итоге строковые представления объектов.



Решение задачи: Сумма произвольного количества скобок

Решение, шаг 1

Чтобы `sum(1)`, а также `sum(1)(2)` можно было вызвать новыми скобками — результатом `sum` должна быть функция.

Но эта функция также должна уметь превращаться в число. Для этого нужно дать ей соответствующий `valueOf`. А если мы хотим, чтобы и в строковом контексте она вела себя так же — то `toString`.

Решение, шаг 2

Функция, которая возвращается `sum`, должна накапливать значение при каждом вызове.

Удобнее всего хранить его в замыкании, в переменной `currentSum`. Каждый вызов прибавляет к ней очередное значение:

```

01 function sum(a) {
02
03     var currentSum = a;
04
05     function f(b) {
06         currentSum += b;
07         return f;
08     }
09
10     f.toString = function() { return currentSum; };
11
12     return f;
13 }
14
15 alert( sum(1)(2) ); // 3
16 alert( sum(5)(-1)(2) ); // 6
17 alert( sum(6)(-1)(-2)(-3) ); // 0
18 alert( sum(0)(1)(2)(3)(4)(5) ); // 15

```

При внимательном взгляде на решение легко заметить, что функция `sum` срабатывает только один раз. Она возвращает функцию `f`.

Затем, при каждом запуске функция `f` добавляет параметр к сумме `currentSum`, хранящейся в замыкании, и возвращает сама себя.

В последней строчке `f` нет рекурсивного вызова.

Вот так была бы рекурсия:

```

1 function f(b) {
2     currentSum += b;
3     return f(); // <-- подвызов
4 }

```

А в нашем случае, мы просто возвращаем саму функцию, ничего не вызывая.

```

1 function f(b) {
2     currentSum += b;
3     return f; // <-- не вызывает сама себя, а возвращает ссылку на себя
4 }

```

Эта `f` используется при следующем вызове, опять возвратит себя, и так сколько нужно раз. Затем, при использовании в строчном или численном контексте — сработает `toString`, который вернет текущую сумму `currentSum`.



Решение задачи: Полиморфная функция outputDate

Для определения примитивного типа строка/число подойдет оператор `typeof` [7].

Примеры его работы:

```
1 alert(typeof 123); // "number"
2 alert(typeof "строка"); // "string"
3 alert(typeof new Date()); // "object"
4 alert(typeof []); // "object"
```

Оператор `typeof` не умеет различать разные типы объектов, они для него все на одно лицо: "object". Поэтому он не сможет отличить `Date` от `Array`.

Используем для того, чтобы их различать, свойство `[[Class]]`.

Функция:

```
01 function outputDate(date) {
02   if (typeof date == 'number') {
03     // перевести секунды в миллисекунды и преобразовать к Date
04     date = new Date(date*1000);
05   } else if (typeof date == 'string') {
06     // разобрать строку и преобразовать к Date
07     date = date.split('-');
08     date = new Date(date[0], date[1]-1, date[2]);
09   } else if ( {}.toString.call(date) == '[object Array]' ) {
10     date = new Date(date[0], date[1], date[2]);
11   }
12
13   var day = date.getDate();
14   if (day < 10) day = '0' + day;
15
16   var month = date.getMonth()+1;
17   if (month < 10) month = '0' + month;
18
19   // взять 2 последние цифры года
20   var year = date.getFullYear() % 100;
21   if (year < 10) year = '0' + year;
22
23   var formattedDate = day + '.' + month + '.' + year;
24
25   alert(formattedDate);
26 }
27
28 outputDate( '2011-10-02' ); // 02.10.11
29 outputDate( 1234567890 ); // 14.02.09
30 outputDate( [2000,0,1] ); // 01.01.00
31 outputDate( new Date(2000,0,1) ); // 01.01.00
```




Решение задачи: Как будет работать `isObject(undefined)`?

1. Да. Так как `undefined == null`, то строка (1) завершит выполнение возвратом `false`.
2. Если убрать строку (1), то браузеры ведут себя более интересно...

По идее, если нестрогий режим, то вызов `f.call(null/undefined)` должен передать в `f` глобальный объект в качестве контекста `this`. Так что браузер должен метод `{}.toString` запустить в контексте `window`. А далее — результат зависит от того, какое у него свойство `[[Class]]`.

В реальности же — большинство браузеров применяют здесь современный стандарт и возвращают `[object Undefined]`. Попробуйте сами..

```
1 // возможны варианты (в зависимости от браузера)
2 // [object Undefined]
3 // [object Object]
4 // [object DOMWindow]
5 alert( {}.toString.call(undefined) );
```

Ссылки

1. Преобразование типов для примитивов <http://learn.javascript.ru/types-conversion>
2. 15.2.4.4 <http://es5.github.com/x15.2.html#x15.2.4.4>
3. 11.8.5 <http://es5.github.com/x11.html#x11.8.5>
4. 11.9.3 <http://es5.github.com/x11.html#x11.9.3>
5. 9.1 <http://es5.github.com/x9.html#x9.1>
6. 9.3 <http://es5.github.com/x9.html#x9.3>
7. Typeof <http://learn.javascript.ru/type-detection#type-typeof>