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: 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

Ранее, в главе Преобразование типов 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`:

```
var user = {
  firstName: 'Василий'
};

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.

```
var user = {

  firstName: 'Василий',

  toString: function() {
    return 'Пользователь ' + this.firstName;
  }
};

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:

```
var obj = {
  toString: function() { return 123; }
};

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:

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

```
её код
```

Численное преобразование

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

```
var room = {
  number: 777,

  valueOf: function() { return this.number; },
  toString: function() { return this.number; }
};

alert( +room );  // 777, вызвался valueOf

delete room.valueOf;

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:

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

Стандартный `valueOf`

Если посмотреть в стандарт, то в 15.2.4.4 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:

```
var obj = {
  valueOf: function() { return 1; }
};

// операции сравнения, кроме ===, приводят к примитиву
alert(obj == true); // true, объект приведён к примитиву 1

// бинарный + приводит к примитиву, а затем складывает
alert(obj + "test"); // 1test
```

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

```
var a = {
  valueOf: function() { return "1"; }
};
var b = {
  valueOf: function() { return true; }
};

alert(a + b); // "1" + true = "1true"
```

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

```
var a = {
  valueOf: function() { return "1"; }
```

```
};
var b = {
  valueOf: function() { return "2"; }
};

alert(a - b); // "1" - "2" = -1
```

Исключение: `Date`

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

```
// бинарный вариант, преобразование к примитиву
alert( new Date + "" ); // "строка даты"

// унарный вариант, наравне с - * / и другими приводит к числу
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:

```
var value = new Boolean(false);
if ( value ) {
  alert(true); // сработает!
}
```

Почему запустился `alert`? Ведь в `if` находится `false`...

Проверим:

```
var value = new Boolean(false);

alert(value); // выводит false, все ок..

if ( value ) {
  alert(true);  // ..но тогда почему выполняется alert в if ?!?
}
```

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 , 11.9.3 , а также 9.1 и 9.3 .

Задача: ['x'] == 'x'

Почему результат `true` ?

Важность: 5

```
alert( ['x'] == 'x' );
```

Решение задачи "['x'] == 'x'" »»

Задача: Преобразование

Объявлен объект с `toString` и `valueOf`.

Важность: 5

Какими будут результаты `alert`?

```
var foo = {
    toString: function () {
        return 'foo';
    },
    valueOf: function () {
        return 2;
    }
};

alert(foo);
alert(foo + 1);
alert(foo + "3");
```

Подумайте, прежде чем ответить.

Решение задачи "Преобразование" »»

Задача: Почему [] == [] неверно, а [ ] == ![ ] верно?

Почему первое равенство — неверно, а второе — верно?

Важность: 5

```
alert( [] == [] );  // false
alert( [] == ![] ); // true
```

Какие преобразования происходят при вычислении?

Решение задачи "Почему [] == [] неверно, а [ ] == ![ ] верно?" »»

Задача: Вопросник по преобразованиям, для объектов

Важность: 5

Подумайте, какой результат будет у выражений ниже. Когда закончите — сверьтесь с решением.

```
new Date(0) - 0
new Array(1)[0] + ""
({})[0]
[1] + 1
[1,2] + [3,4]
[] + null + 1
[[0]][0][0]
({} + {})
```

Решение задачи "Вопросник по преобразованиям, для объектов" »»

Задача: Сумма произвольного количества скобок

Важность: 2

Напишите функцию `sum`, которая будет работать так:

```
sum(1)(2) == 3; // 1 + 2
sum(1)(2)(3) == 6; // 1 + 2 + 3
sum(5)(-1)(2) == 6
sum(6)(-1)(-2)(-3) == 0
sum(0)(1)(2)(3)(4)(5) == 15
```

Количество скобок может быть любым.

Пример такой функции для двух аргументов — есть в решении задачи Сумма через замыкание.

Решение задачи "Сумма произвольного количества скобок" »»

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:

```
if (window.jQuery !== undefined) { ... }

//а если мы знаем, что это объект, то проверку можно упростить:
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` надежно 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'`:

```
alert( typeof {} ); // 'object'
alert( typeof [] ); // 'object'
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 прочитать его.

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

```
var obj = {};
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.

К сожалению, большинство объектов переопределяют `toString` на свой собственный. 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:

```
var toClass = {}.toString; // (1)

var arr = [1,2];
alert( toClass.call(arr) ); // (2) [object Array]

var date = new Date;
alert( toClass.call(date) ); // [object Date]

var type = toClass.call(date).slice(8, -1); // (3)
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:

```
var arr = [1,2];
arr.toClass = {}.toString;

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:

```
alert( {}.toString.call(123) ); // [object Number]
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"`:

```
function Animal(name) {
  this.name = name;
}
var animal = new Animal("Винни-пух");

var type = {}.toString.call( animal );

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 сработает нормально.

«Утиная» типизация

Утиная типизация основана на одной известной пословице: *«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`:

```
var x = [1,2,3];

if (x.splice) {
  alert('Массив!');
}
```

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`:

```
var x = new Date();

```

```
if (x.getTime) {
  alert('Дата!');
}
```

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:

```
function Animal(name) {
  this.name = name;
}
var animal = new Animal("Винни-пух");

alert( animal instanceof Animal ); // true
```

Оператор `instanceof` также работает для встроенных объектов:

```
var d = new Date();
alert(d instanceof Date); // true

function f() { }
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:

```
var arr = [1,2,3];
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.

Задача: Полиморфная функция outputDate

Напишите функцию `outputDate(date)`, которая выводит дату в формате `dd.mm.yy`. Важность: 5

Ее первый аргумент должен содержать дату в одном из видов:

1. Как объект `Date`.

2. Как строку в формате `yyyy-mm-dd`.
3. Как число *секунд* с `01.01.1970`.
4. Как массив `[гггг, мм, дд]`, месяц начинается с нуля

Для этого вам понадобится определить тип данных аргумента и, при необходимости, преобразовать входные данные в нужный формат.

Пример работы:

```
function outputDate(date) { /* ваш код */ }

outputDate( '2011-10-02' );  // 02.10.11
outputDate( 1234567890 );  // 14.02.09
outputDate( [2000,0,1] ); // 01.01.00
outputDate( new Date(2000,0,1) ); // 01.01.00
```

Решение задачи "Полиморфная функция outputDate" »»

Задача: Как будет работать isObject(undefined)?

Важность: 2

Есть функция для проверки, является ли значение встроенным объектом `Object`:

```
function isObject(x) {
  if (x == null) return false; // (1)

  return {}.toString.call(x) == "[object Object]"; // (2)
}
```

Как будет работать эта функция в случае вызова `isObject(undefined)`?

1. Вернёт ли она ответ сразу в строке: `(1)`?
2. Если да, то что будет, если убрать строку `(1)`? То есть, что вернёт `{}.toString.call(undefined)` и почему?

Решение задачи "Как будет работать isObject(undefined)?" »»

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:

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

```
({})[0] = undefined // (3)
[1] + 1 = "11" // (4)
[1,2] + [3,4] = "1,23,4" // (5)
[] + null + 1 = "null1" // (6)
[[0]][0][0] = 0 // (7)
({} + {}) = "[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
```

```
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:

```
function f(b) {
  currentSum += b;
  return f(); // <-- подвызов
}
```

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

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

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.

Примеры его работы:

```
alert(typeof 123); // "number"
alert(typeof "строка"); // "string"
```

```
alert(typeof new Date()); // "object"
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:

```
function outputDate(date) {
  if (typeof date == 'number') {
    // перевести секунды в миллисекунды и преобразовать к Date
    date = new Date(date*1000);
  } else if(typeof date == 'string') {
    // разобрать строку и преобразовать к Date
    date = date.split('-');
    date = new Date(date[0], date[1]-1, date[2]);
  } else if ( {}.toString.call(date) == '[object Array]' ) {
    date = new Date(date[0], date[1], date[2]);
  }

  var day = date.getDate();
  if (day < 10) day = '0' + day;

  var month = date.getMonth()+1;
  if (month < 10) month = '0' + month;

  // взять 2 последние цифры года
  var year = date.getFullYear() % 100;
  if (year < 10) year = '0' + year;

  var formattedDate = day + '.' + month + '.' + year;

  alert(formattedDate);
}

outputDate( '2011-10-02' );  // 02.10.11
outputDate( 1234567890 );  // 14.02.09
outputDate( [2000,0,1] ); // 01.01.00
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..

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