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: Основы JavaScript

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

Содержание

Структура кода

В этой главе мы рассмотрим общую структуру кода, команды и их разделение.

Команды

Например, можно вместо одного вызова `alert` сделать два:

```
alert('Привет'); alert('Мир');
```

Как правило, новая команда занимает отдельную строку — так код лучше читается:

```
alert('Привет');
alert('Мир');
```

Точку с запятой во многих случаях можно не ставить, если есть переход на новую строку. Так тоже будет работать:

```
alert('Привет')
alert('Мир')
```

В этом случае JavaScript интерпретирует переход на новую строчку как разделитель команд и автоматически вставляет «виртуальную» точку с запятой между ними.

Однако, внутренние правила по вставке точки с запятой не идеальны. В примере выше они сработали, но в некоторых ситуациях JavaScript «забывает» вставить точку с запятой там, где она нужна. Таких ситуаций не так много, но они все же есть, и ошибки, которые при этом появляются, достаточно сложно исправлять.

Поэтому рекомендуется точки с запятой ставить. Сейчас это, фактически, стандарт.

Комментарии

Со временем программа становится большой и сложной. Появляется необходимость добавить *комментарии*, которые объясняют, что происходит и почему.

Комментарии могут находиться в любом месте программы и никак не влияют на ее выполнение. Интерпретатор JavaScript попросту игнорирует их.

Однострочные комментарии начинаются с двойного слэша `//`. Текст считается комментарием до конца строки:

```
// Команда ниже говорит "Привет"
alert('Привет');

alert('Мир'); // Второе сообщение выводим отдельно
```

Многострочные комментарии начинаются слешем-звездочкой `"/*"` и заканчиваются звездочкой-слэшем `"*/"`, вот так:

```
/* Пример с двумя сообщениями.
Это - многострочный комментарий.
*/
alert('Привет');
alert('Мир');
```

Все содержимое комментария игнорируется. Если поместить код внутрь `/* ... */` или после `//` — он не выполнится.

```
/* Закомментировали код
alert('Привет');
*/
alert('Мир');
```

Вложенные комментарии не поддерживаются!

В этом коде будет ошибка:

```
/*
alert('Привет'); /* вложенный комментарий ?!? */
*/
alert('Мир');
```

В многострочных комментариях всё очень просто — комментарий длится от открытия `/*` до закрытия `*/`. Таким образом, код выше будет интерпретирован так:

Комментарий открывается `/*` и закрывается `*/`:

```
/*
alert('Привет'); /* вложенный комментарий ?!? */
```

Код (лишние символы сверху вызывают ошибку):

```
*/
alert('Мир');
```

Виды комментариев

Существует три типа комментариев.

1. Первый тип отвечает на вопрос *«Что делает эта часть кода?»*.

   Эти комментарии бывают особенно полезны, если используются неочевидные алгоритмы.

2. Второй тип комментариев отвечает на вопрос *«Почему я выбрал этот вариант решения задачи?»*. И он гораздо важнее.

   При создании кода мы принимаем много решений, выбираем лучший вариант из нескольких возможных. Иногда для правильного выбора нужно многое изучить, посмотреть.

   Когда вы остановились на чём-то — не выбрасывайте проделанную работу, укажите, хотя бы кратко, что вы посмотрели и почему остановились именно на этом варианте.

   Особенно это важно, если выбранный вариант не очевиден, а существует другое, более очевидное, но неправильное решение. Ведь в будущем, вернувшись к этому коду, мы можем захотеть переписать «сложное» решение на более «явное» или «оптимальное», тут-то и комментарий и поможет понять, что к чему.

   Например: *«Я выбрал здесь анимацию при помощи JavaScript вместо CSS, поскольку IE именно в этом месте ведёт себя некорректно»*.

3. Третий тип комментариев возникает, когда мы в одном месте кода делаем вычисления или присвоения переменных, неочевидным образом использованные совсем в другом месте кода.

   Например: *«Эти значения отформатированы именно так, чтобы их можно было передать на сервер»*.

Не бойтесь комментариев. Чем больше кода в проекте — тем они важнее. Что же касается увеличения размера кода — это не страшно, т.к. существуют инструменты сжатия JavaScript, которые при публикации кода легко их удалят.

На следующих занятиях мы поговорим о переменных, блоках и других структурных элементах программы на JavaScript.

Переменные

В зависимости от того, для чего вы делаете скрипт, понадобится работать с информацией.

Если это электронный магазин - то это товары, корзина. Если чат - посетители, сообщения и так далее.

Чтобы хранить информацию, используются *переменные*.

Переменная

***Переменная* состоит из имени и выделенной области памяти, которая ему соответствует**.

Для *объявления* или, другими словами, *создания переменной* используется ключевое слово `var`:

```
var message;
```

После объявления, можно записать в переменную данные:

```
var message;
message = 'Привет'; // сохраним в переменной строку
```

Эти данные будут сохранены в соответствующей области памяти и в дальнейшем доступны при обращении по имени:

```
var message;
message = 'Привет';

alert(message); // выведет содержимое переменной
```

Для краткости можно совместить объявление переменной и запись данных:

```
var message = 'Привет';
```

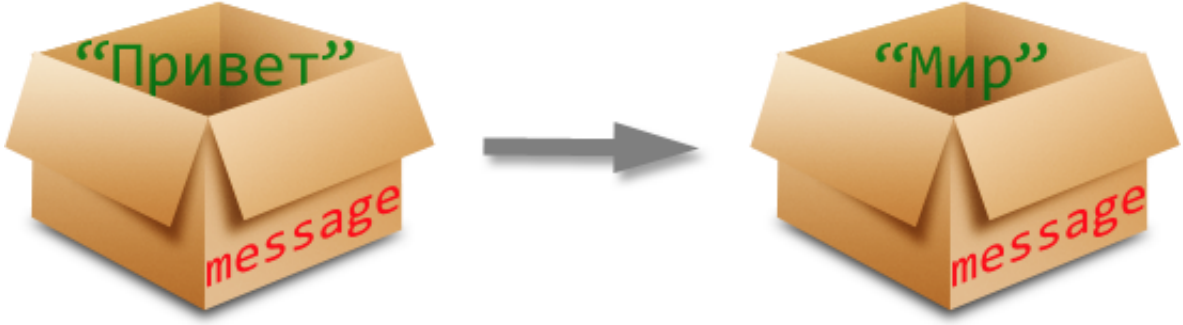
Аналогия из жизни

Проще всего понять переменную, если представить ее как «коробку» для данных, с уникальным именем.

Например, переменная `message` - это коробка, в которой хранится значение `"Привет"`:

В коробку можно положить любое значение, а позже - поменять его. Значение в переменной можно изменять сколько угодно раз:

```
var message;

message = 'Привет';

message = 'Мир';    // заменили значение

alert(message);
```



При изменении значения старое содержимое переменной удаляется.

Существуют функциональные [1] языки программирования, в которых значение переменной менять нельзя.

В таких языках положил один раз значение в коробку - и оно хранится там вечно, ни удалить ни изменить. А нужно что-то другое сохранить - изволь создать новую коробку (объявить новую переменную), повторное использование невозможно.

С виду - не очень удобно, но, как ни странно, и на таких языках вполне можно успешно программировать. Изучение какого-нибудь функционального языка рекомендуется для расширения кругозора. Отличный кандидат для этого — язык Erlang  .

Копирование значений

Переменные в JavaScript могут хранить не только строки, но и другие данные, например, числа.

Объявим две переменные, положим в одну - строку, а в другую - число.
Как вы можете видеть, переменной без разницы, что хранить:

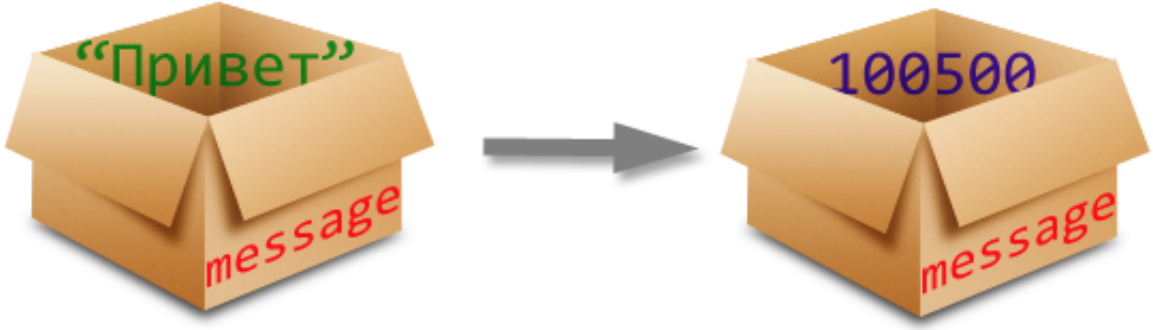
```
var num = 100500;
var message = 'Привет';
```

Значение можно копировать из одной переменной в другую.

```
1 var num = 100500;
2 var message = 'Привет';
3
4 message = num;
```

Значение из `num` перезаписывает текущее в `message`.

В «коробке» `message` меняется значение:



После этого присваивания в обеих коробках `num` и `message` находится одно и то же значение `100500`.

Важность директивы `var`

В JavaScript вы можете создать переменную и без `var`, достаточно просто присвоить ей значение:

```
x = "value"; // переменная создана, если ее не было
```

Технически, это не вызовет ошибки, но делать так все-таки не стоит.

Всегда определяйте переменные через `var`. Это хороший тон в программировании и помогает избежать ошибок.

Откройте пример в IE в новом окне:

```
<html>
<head>
  <meta http-equiv="X-UA-Compatible" content="IE=8">
</head>
<body>
  <div id="test"></div>

  <script>
    test = 5;
    alert(test);
  </script>

</body>
</html>
```

Значение не выведется, будет ошибка. Если в IE включена отладка или открыта панель разработки - вы увидите ее.

Дело в том, что почти все современные браузеры создают переменные для элементов, у которых есть `id`.

Переменная `test` в них в любом случае существует, запустите, к примеру:

```
<div id="test"></div>

<script>
  alert(test); // выведет элемент
</script>
```

..Но в IE<9 такую переменную изменять нельзя.

Всё будет хорошо, если объявить `test`, используя `var`:
Правильный код:

```
<html>
<body>
  <div id="test"></div>

  <script>
    var test = 5;
    alert(test);
  </script>

</body>
</html>
```

Самое «забавное» — то, что, эта ошибка будет только в IE<9, и только если на странице присутствует элемент с совпадающим `id`.

Такие ошибки особенно весело исправлять и отлаживать.

Есть и еще ситуации, когда отсутствие `var` может привести к ошибкам. Надеюсь, вы убедились в необходимости всегда ставить `var`.

Константы

Константа — это переменная, которая никогда не меняется. Как правило, их называют большими буквами, через подчёркивание.
Например:

```
var COLOR_RED = "#F00";
var COLOR_GREEN = "#0F0";
var COLOR_BLUE = "#00F";
var COLOR_ORANGE = "#FF7F00";

alert(COLOR_RED); // #F00
```

Технически, константа является обычной переменной, то есть её можно изменить. Но мы договариваемся этого не делать.

Зачем нужны константы? Почему бы просто не использовать `"#F00"` или `"#0F0"`?

1. Во-первых, константа — это понятное имя, в отличие от строки `"#FF7F00"`.
2. Во-вторых, опечатка в строке может быть не замечена, а в имени константы её упустить невозможно — будет ошибка при выполнении.

Константы используют вместо строк и цифр, чтобы сделать программу понятнее и избежать ошибок.

Имена переменных

Один из самых важных навыков программиста — умение называть переменные *правильно*.

Имена переменных

На имя переменной в JavaScript наложены всего два ограничения.

1. Имя может состоять из: букв, цифр, символов `$` и `_`
2. Первый символ не должен быть цифрой.

Например:

```
var myName;
var test123;
```

Что здесь особенно интересно - доллар `'$'` и знак подчеркивания `'_'` являются такими же обычными символами, как буквы:

```
var $ = 5;  // объявили переменную с именем '$'
var _ = 15; // переменная с именем '_'

alert($);
```

А такие переменные были бы неправильными:

```
var 1a; // начало не может быть цифрой

var my-name; // дефис '-' не является разрешенным символом
```

Регистр букв имеет значение

Переменные `apple` и `AppLE` - две разные переменные.

Русские буквы допустимы, но не рекомендуются

Можно использовать и русские буквы:

```
var имя = "Вася";
alert(имя);
```

Технически, ошибки здесь нет, но на практике сложилась традиция использовать в именах только английские буквы.

Зарезервированные имена

Существует список зарезервированных слов, которые нельзя использовать при именовании переменных, так как они используются самим языком, например: `var, class, return, implements` и др.

Некоторые слова, например, `class`, не используются в современном JavaScript, но они заняты на будущее. Некоторые браузеры позволяют их использовать, но это может привести к ошибкам.

Следующий пример будет работать во многих старых браузерах, которые допускают использование слова `'class'` и не сработает в современных. Они выдадут синтаксическую ошибку, попробуйте, запустите:

```
var class = 5;
alert(class);
```

Правильный выбор имени

Правильный выбор имени переменной - одна из самых важных и сложных вещей в программировании, которая отличает начинающего от гуру.

Дело в том, что большинство времени мы тратим не на изначальное написание кода, а на его развитие.

А что такое развитие? Это когда я вчера написал код, а сегодня (или спустя неделю) прихожу и хочу его поменять. Например, вывести сообщение не так, а эдак.. Обработать товары по-другому, добавить функционал.. А где у меня там сообщение хранится?..

Гораздо проще найти нужные данные, если они правильно помечены, т.е. переменная названа *правильно*.

- **Правило 1.** Никакого транслита. Только английский.

  Неприемлемы:

  ```
  var moiTovari;
  var cena;
  var ssilka;
  ```

  Подойдут:

  ```
  var myGoods;
  var price;
  var link;
  ```

  Если вы вдруг не знаете английский - самое время выучить. Все равно ведь придется…

- **Правило 2.** Использовать короткие имена только для переменных «местного значения».

  Называть переменные именами, не несущими смысловой нагрузки, например `a`, `e`, `p`, `mg` - можно только в том случае, если они используются в небольшом фрагменте кода и их применение очевидно.

  Вообще же, название переменной должно быть понятным. Иногда для этого нужно использовать несколько слов.

- **Правило 3.** Переменные из нескольких слов пишутся `вместеВотТак`.

  Например:

  ```
  var borderLeftWidth;
  ```

  Этот способ записи называется «верблюжьей нотацией» или, по-английски, «camelCase».

  Существует альтернативный стандарт, когда несколько слов пишутся через знак подчеркивания `'_'`:

  ```
  var border_left_width;
  ```

  Преимущественно в JavaScript используется вариант `borderLeftWidth`, в частности во встроенных языковых и браузерных функциях. Поэтому целесообразно остановиться на нём.

  Ещё одна причина выбрать «верблюжью нотацию» — запись в ней немного короче, чем с подчеркиванием, т.к. не нужно вставлять `'_'`.

➡ **Правило последнее, главное.** Имя переменной должно максимально четко соответствовать хранимым в ней данным.

Придумывание таких имен - одновременно коротких и точных, приходит с опытом, но только если сознательно стремиться к этому.

Позвольте поделиться одним небольшим секретом, который позволит улучшить ваши названия переменных и сэкономит вам время.

Бывает так, что вы написали код, через некоторое время к нему возвращаетесь, и вам надо что-то поправить. Например, изменить какую-то рамку «`border`...»

... И вы помните, что переменная, в которой хранится нужное вам значение, называется примерно так: `borderLeftWidth`. Вы ищете ее в коде, не находите, разбираетесь, и обнаруживаете, что на самом деле переменная называлась вот так: `leftBorderWidth`. После чего вносите нужные правки.

В этом случае, **самый лучший ход - это *переименовать* переменную на ту, которую вы искали изначально**. То есть, у вас в коде `leftBorderWidth`, а вы ее переименовываете на `borderLeftWidth`.

Зачем? Дело в том, что в следующий раз, когда вы захотите что-то поправить, то вы будете искать по тому же самому имени. Соответственно, это сэкономит вам время.

Кроме того, поскольку именно это имя переменной пришло вам в голову - скорее всего, оно больше соответствует хранимым там данным, чем то, которое вы придумали изначально.

Смысл имени переменной - это «имя на коробке», по которому мы сможем максимально быстро находить нужные нам данные.

Не нужно бояться переименовывать переменные, если вы придумали имя получше. Современные редакторы позволяют делать это очень удобно. Это в конечном счете сэкономит вам время.

Храните в переменной то, что следует

Бывают ленивые программисты, которые, вместо того чтобы объявить новую переменную, используют существующую.

В результате получается, что такая переменная — как коробка, в которую кидают то одно, то другое, то третье, при этом не меняя название. Что в ней лежит сейчас? А кто его знает.. Нужно подойти, проверить.

Сэкономит такой программист время на объявлении переменной — потеряет в два раза больше на отладке кода.

«Лишняя» переменная — добро, а не зло.

Введение в типы данных

В JavaScript существует несколько основных типов данных.

Типы данных

1. **Число `number`**:

```
var n = 123;
n = 12.345;
```

   Единый тип *число* используется как для целых, так и для дробных чисел.

   Существуют специальные числовые значения `Infinity` (бесконечность) и `NaN` (ошибка вычислений). Они также принадлежат типу «число».

   Например, бесконечность `Infinity` получается при делении на ноль:

```
alert( 1 / 0 ); // Infinity
```

   Ошибка вычислений `NaN` будет результатом некорректной математической операции, например:

```
alert( "нечисло" * 2 ); // NaN, ошибка
```

2. **Строка `string`**:

```
var str = "Мама мыла раму";
str = 'Одинарные кавычки тоже подойдут';
```

   В JavaScript одинарные и двойные кавычки равноправны. Можно использовать или те или другие.

   Тип *символ* не существует, есть только *строка*

   В некоторых языках программирования есть специальный тип данных для одного символа. Например, в языке C это `char`. В JavaScript есть только тип «строка» `string`. Что, надо сказать, вполне удобно..

3. **Булевый (логический) тип `boolean`**. У него всего два значения - `true` (истина) и `false` (ложь).

   Как правило, такой тип используется для хранения значения типа да/нет, например:

```
var checked = true; // поле формы помечено галочкой
checked = false;    // поле формы не содержит галочки
```

   О нём мы поговорим более подробно, когда будем обсуждать логические вычисления и условные операторы.

4. **`null`** — специальное значение. Оно имеет смысл «ничего». Значение `null` не относится ни к одному из типов выше, а образует свой отдельный тип, состоящий из единственного значения `null`:

```
var age = null;
```

   В JavaScript `null` не является «ссылкой на несуществующий объект» или «нулевым указателем», как в некоторых других языках. Это просто специальное значение, которое имеет смысл «ничего» или «значение неизвестно».

   В частности, код выше говорит о том, что возраст `age` неизвестен.

5. **`undefined`** — специальное значение, которое, как и `null`, образует свой собственный тип. Оно имеет смысл «значение не присвоено».

Если переменная объявлена, но в неё ничего не записано, то ее значение как раз и есть `undefined`:

```
var u;
alert(u); // выведет "undefined"
```

Можно присвоить `undefined` и в явном виде, хотя это делается редко:

```
var x = 123;
x = undefined;
```

В явном виде `undefined` обычно не присваивают, так как это противоречит его смыслу. Для записи в переменную «пустого значения» используется `null`.

6. **Объекты `object`**.

   Первые 5 типов называют *«примитивными»*.

   Особняком стоит шестой тип: *«объекты»*. К нему относятся, например, даты, он используется для коллекций данных и для многого другого. Позже мы вернёмся к этому типу и рассмотрим его принципиальные отличия от примитивов.

Итого

Есть 5 «примитивных» типов: `number`, `string`, `boolean`, `null`, `undefined` и 6-й тип — объекты `object`.

Основные операторы

Для работы с переменными, со значениями, JavaScript поддерживает все стандартные операторы, большинство которых есть и в других языках программирования.

Термины: «унарный», «бинарный», «операнд»

У операторов есть своя терминология, которая используется во всех языках программирования.

- *Операнд* — то, к чему применяется оператор. Например: `5 * 2` — оператор умножения с левым и правым операндами. Другое название: «аргумент оператора».
- *Унарным* называется оператор, который применяется к одному выражению. Например, оператор унарный минус `"-"` меняет знак числа на противоположный:

  ```
  var x = 1;
  alert( -x );     // -1, унарный минус
  alert( -(x+2) ); // -3, унарный минус применён к результату сложения x+2
  alert( -(-3) );  // 3
  ```

- *Бинарным* называется оператор, который применяется к двум операндам. Тот же минус существует и в бинарной форме:

  ```
  var x = 1, y = 3;
  alert( y - x ); // 2, бинарный минус
  ```

Работа унарного `"+"` и бинарного `"+"` в JavaScript существенно различается.

Это действительно разные операторы. Бинарный плюс складывает операнды, а унарный — ничего не делает в арифметическом плане,

но зато приводит операнд к числовому типу. Далее мы увидим примеры.

Арифметические операторы

Базовые арифметические операторы знакомы нам с детства: это плюс `+`, минус `-`, умножить `*`, поделить `/`.

Например:

```
alert(2 + 2); // 4
```

Или чуть сложнее:

```
var i = 2;

i = (2 + i) * 3 / i;

alert(i); // 6
```

Более редкий арифметический оператор `%` интересен тем, что никакого отношения к процентам не имеет. Его результат `a % b` — это остаток от деления `a` на `b`.

Например:

```
alert(5 % 2); // 1, остаток от деления 5 на 2
alert(8 % 3); // 2, остаток от деления 8 на 3
alert(6 % 3); // 0, остаток от деления 6 на 3
```

Сложение строк, бинарный `+`

Если бинарный оператор `+` применить к строкам, то он их объединяет в одну:

```
var a = "моя" + "строка";
alert(a); // моястрока
```

Если хотя бы один аргумент является строкой, то второй будет также преобразован к строке!

Причем не важно, справа или слева находится операнд-строка, в любом случае нестроковый аргумент будет преобразован. Например:

```
alert( '1' + 2 ); // "12"
alert( 2 + '1' ); // "21"
```

Это приведение к строке — особенность бинарного оператора `"+"`.

Остальные арифметические операторы работают только с числами и всегда приводят аргументы к числу.

Например:

```
alert( '1' - 2 ); // -1
alert( 6 / '2'); // 3
```

Унарный плюс `+`

Унарный плюс как арифметический оператор ничего не делает:

```
alert( +1 ); // 1
alert( +(1-2) ); // -1
```

Как видно, плюс ничего не изменил в выражениях. Результат — такой же, как и без него.

Тем не менее, он широко применяется, так как его «побочный эффект» — преобразование значения в число.

Например, у нас есть два числа, в форме строк, и нужно их сложить. Бинарный плюс сложит их как строки, поэтому используем унарный плюс, чтобы преобразовать к числу:

```
var a = "2";
var b = "3";

alert( a + b ); // "23", так как бинарный плюс складывает строки
alert( +a + b ); // "23", второй операнд - всё ещё строка

alert( +a + +b); // 5, число, так как оба операнда предварительно преобразованы в числа
```

Присваивание

Оператор присваивания выглядит как знак равенства `=`:

```
var i = 1 + 2;

alert(i); // 3
```

Он вычисляет выражение, которое находится справа, и присваивает результат переменной. Это выражение может быть достаточно сложным и включать в себя любые другие переменные:

```
var a = 1;
var b = 2;

a = b + a + 3;  // (*)

alert(a); // 6
```

В строке `(*)` сначала произойдет вычисление, использующее текущее значение `a` (т.е. `1`), после чего результат перезапишет старое значение `a`.

Возможно присваивание по цепочке:

```
var a, b, c;

a = b = c = 2 + 2;

alert(a); // 4
alert(b); // 4
alert(c); // 4
```

Такое присваивание работает справа-налево, то есть сначала вычислятся самое правое выражение `2+2`, присвоится в `c`, затем выполнится `b = c` и, наконец, `a = b`.

Оператор `"="` возвращает значение

Все операторы возвращают значение. Вызов `x = выражение` записывает выражение в `x`, а затем возвращает его. Благодаря этому присваивание можно использовать как часть более сложного выражения:

```
1 var a = 1;
2 var b = 2;
3
4 var c = 3 - (a = b + 1);
5
6 alert(a); // 3
7 alert(c); // 0
```

В примере выше результатом `(a = b + 1)` является значение, которое записывается в `a` (т.е. `3`). Оно используется для вычисления `c`.

Забавное применение присваивания, не так ли?

Знать, как это работает — стоит обязательно, а вот писать самому — только если вы уверены, что это сделает код более читаемым и понятным.

Приоритет

В том случае, если в выражении есть несколько операторов - порядок их выполнения определяется *приоритетом*.

Из школы мы знаем, что умножение в выражении `2 * 2 + 1` выполнится раньше сложения, т.к. его *приоритет* выше, а скобки явно задают порядок выполнения. Но в JavaScript — гораздо больше операторов, поэтому существует целая таблица приоритетов [2] .

Она содержит как уже пройденные операторы, так и те, которые мы еще не проходили. В ней каждому оператору задан числовой приоритет. Тот, у кого число меньше — выполнится раньше. Если приоритет одинаковый, то порядок выполнения — слева направо.

Отрывок из таблицы:

…	…	…
5	умножение	`*`
5	деление	`/`
6	сложение	`+`
6	вычитание	`-`
17	присвоение	`=`
…	…	…

Посмотрим на таблицу в действии.

В выражении `x = 2 * 2 + 1` есть три оператора: присвоение `=`, умножение `*` и сложение `+`. Приоритет умножения `*` равен `5`, оно выполнится первым, затем произойдёт сложение `+`, у которого приоритет `6`, и после них — присвоение `=`, с приоритетом 17.

Инкремент/декремент: `++, --`

Одной из наиболее частых операций в JavaScript, как и во многих других языках программирования, является увеличение или уменьшение

переменной на единицу.

Для этого существуют даже специальные операторы:

- **Инкремент** `++` увеличивает на 1:

```
var i = 2;
i++;      // более короткая запись для i = i + 1.
alert(i); // 3
```

- **Декремент** `--` уменьшает на 1:

```
var i = 2;
i--;      // более короткая запись для i = i - 1.
alert(i); // 1
```

Инкремент/декремент можно применить только к переменной.
Код `5++` даст ошибку.

Вызывать эти операторы можно не только после, но и перед переменной: `i++` (называется «постфиксная форма») или `++i` («префиксная форма»).

Обе эти формы записи делают одно и то же: увеличивают на `1`.

Тем не менее, между ними существует разница. Она видна только в том случае, когда мы хотим не только увеличить/уменьшить переменную, но и использовать результат в том же выражении.

Например:

```
var i = 1;
var a = ++i; // (*)

alert(a); // 2
```

В строке `(*)` вызов `++i` увеличит переменную, а *затем* вернёт ее значение в `a`. **То есть, в `a` попадёт значение `i` *после* увеличения**.

Постфиксная форма `i++` отличается от префиксной `++i` тем, что возвращает старое значение, бывшее до увеличения.

В примере ниже в `a` попадёт старое значение `i`, равное `1`:

```
var i = 1;
var a = i++; // (*)

alert(a); // 1
```

- Если результат оператора не используется, а нужно только увеличить/уменьшить переменную — без разницы, какую форму использовать:

```
var i = 0;
i++;
++i;
alert(i); // 2
```

- Если хочется тут же использовать результат, то нужна префиксная форма:

```
var i = 0;
alert( ++i ); // 1
```

- Если нужно увеличить, но нужно значение переменной *до увеличения* — постфиксная форма:

```
var i = 0;
alert( i++ ); // 0
```

Инкремент/декремент можно использовать в любых выражениях.

При этом он имеет более высокий приоритет и выполняется раньше, чем арифметические операции:

```
var i = 1;
alert( 2 * ++i ); // 4
```

```
var i = 1;
alert( 2 * i++ ); // 2, выполнился раньше но значение вернул старое
```

При этом, нужно с осторожностью использовать такую запись, потому что при чтении кода зачастую неочевидно, что переменая увеличивается. Три строки — длиннее, зато нагляднее:

```
var i = 1;
alert( 2 * i );
i++;
```

Побитовые операторы

Побитовые операторы рассматривают аргументы как 32-разрядные целые числа и работают на уровне их внутреннего двоичного представления.

Эти операторы не являются чем-то специфичным для JavaScript, они поддерживаются в большинстве языков программирования.

Поддерживаются следующие побитовые операторы:

- AND(и) (`&`)
- OR(или) (`|`)
- XOR(побитовое исключающее или) (`^`)
- NOT(не) (`~`)
- LEFT SHIFT(левый сдвиг) (`<<`)
- RIGHT SHIFT(правый сдвиг) (`>>`)
- ZERO-FILL RIGHT SHIFT(правый сдвиг с заполнением нулями) (`>>>`)

Вы можете более подробно почитать о них в отдельной статье Побитовые операторы [3].

Вызов операторов с присваиванием

Часто нужно применить оператор к переменной и сохранить результат в ней же, например:

```
var n = 2;
n = n + 5;
n = n * 2;
```

Эту запись можно укоротить при помощи совмещённых операторов: `+=, -=, *=, /=, >>=, <<=, >>>=, &=, |=, ^=`.

Вот так:

```
var n = 2;
n += 5; // теперь n=7 (работает как n = n + 5)
n *= 2; // теперь n=14 (работает как n = n * 2)

alert(n); // 14
```

Все эти операторы имеют в точности такой же приоритет, как обычное присваивание, то есть выполняются после большинства других операций.

Оператор запятая

Запятая тоже является оператором. Ее можно вызвать явным образом, например:

```
a = (5, 6);

alert(a);
```

Запятая позволяет перечислять выражения, разделяя их запятой `','`. Каждое из них — вычисляется и отбрасывается, за исключением последнего, которое возвращается.

Запятая — единственный оператор, приоритет которого ниже присваивания. В выражении `a = (5,6)` для явного задания приоритета использованы скобки, иначе оператор `'='` выполнился бы до запятой `','`, получилось бы `(a=5), 6`.

Зачем же нужен такой странный оператор, который отбрасывает значения всех перечисленных выражений, кроме последнего?

Обычно он используется в составе более сложных конструкций, чтобы сделать несколько действий в одной строке. Например:

```
// три операции в одной строке
for (a = 1, b = 3, c = a*b; a < 10; a++) {
  ...
}
```

Такие трюки используются во многих JavaScript-фреймворках для укорачивания кода.

Операторы сравнения и логические значения

В этом разделе мы познакомимся с операторами сравнения и с логическими значениями, которые такие операторы возвращают.

Многие операторы сравнения знакомы нам со школы:

- Больше/меньше: `a > b`, `a < b`.
- Больше/меньше или равно: `a >= b`, `a <= b`.
- Равно `a == b`.
  Для сравнения используется два символа равенства `'='`. Один символ `a = b` означал бы присваивание.
- «Не равно». В школе он пишется как `≠`, в JavaScript — знак равенства с восклицательным знаком перед ним `!=`.

Логические значения

Как и другие операторы, сравнение возвращает значение. Это значение имеет специальный *логический* тип.

Существует всего два логических значения:

- `true` — имеет смысл «да», «верно», «истина».
- `false` — означает «нет», «неверно», «ложь».

Например:

```
alert( 2 > 1 ); // true, верно
alert( 2 == 1 ); // false, неверно
alert( 2 != 1 ); // true
```

Логические значения можно использовать и напрямую, присваивать переменным, работать с ними как с любыми другими:

```
var a = true; // присвоили явно
var b = 3 > 4; // false

alert( b ); // false

alert( a == b ); // (true == false) неверно, результат false
```

Сравнение строк

Строки сравниваются побуквенно:

```
alert( 'Б' > 'А' ); // true
```

Буквы сравниваются в алфавитном порядке. Какая буква в алфавите позже — та и больше.

Кодировка Unicode

Аналогом «алфавита» во внутреннем представлении строк служит кодировка, у каждого символа — свой номер (код). JavaScript использует кодировку Unicode. При этом сравниваются *численные коды символов*.

В кодировке Unicode обычно код у строчной буквы больше, чем у прописной, поэтому:

```
alert('a' > 'Я'); // true, строчные буквы больше прописных
```

Для корректного сравнения символы должны быть в одинаковом регистре.

Сравнение осуществляется как в телефонной книжке или в словаре. Сначала сравниваются первые буквы, потом вторые, и так далее, пока одна не будет больше другой.

Иными словами, больше — та строка, которая в телефонной книге была бы на большей странице.

Например:

- Если первая буква одной строки больше — значит первая строка больше, независимо от остальных символов:

```
alert( 'Банан' > 'Аят' );
```

➡ Если одинаковы — сравнение идёт дальше. Здесь оно дойдёт до третьей буквы:

```
alert( 'Вася' > 'Ваня' ); // true, т.к. 'с' > 'н'
```

➡ При этом любая буква больше отсутствия буквы:

```
alert( 'Привет' > 'Прив' ); // true, так как 'е' больше чем "ничего".
```

Такое сравнение называется *лексикографическим*.

Обычно мы получаем значения от посетителя в виде строк. Например, `prompt` возвращает *строку*, которую ввел посетитель.

Числа, полученные таким образом, в виде строк сравнивать нельзя, результат будет неверен. Например:

```
alert( "2" > "14" ); // true, неверно, ведь 2 не больше 14
```

В примере выше `2` оказалось больше `14`, потому что строки сравниваются посимвольно, а первый символ `'2'` больше `'1'`.

Правильно было бы преобразовать их к числу явным образом. Например, поставив перед ними `+`:

```
alert( +"2" > +"14" ); // false, теперь правильно
```

Сравнение разных типов

При сравнении значения преобразуются к числам. Исключение: когда оба значения — строки, тогда не преобразуются.

Например:

```
alert( '2' > 1 ); // true
alert( '01' == 1 ); //true
alert( false == 0 ); // true, false становится 0, а true 1.
```

Тема преобразований типов будет продолжена далее, в главе Преобразование объектов: toString и valueOf [4].

Строгое равенство

Обычное равенство не может отличить `0` от `false`:

```
alert(0 == false); // true, т.к. false преобразуется к 0
```

Что же делать, если все же нужно отличить `0` от `false`?

Для проверки равенства без преобразования типов используются операторы строгого равенства `===` (тройное равно) и `!==`.

Они сравнивают без приведения типов. Если тип разный, то такие значения всегда неравны (строго):

```
alert(0 === false); // false, т.к. типы различны
```

Сравнение с null и undefined

Проблемы со специальными значениями возможны, когда к переменной применяется операция сравнения `> < <= >=`, а у неё может быть как численное значение, так и `null/undefined`.

Интуитивно кажется, что `null/undefined` эквивалентны нулю, но это не так! Они ведут себя по-другому.

1. **Значения `null` и `undefined` равны `==` друг другу и не равны чему бы то ни было ещё.**
   Это жёсткое правило буквально прописано в спецификации языка.
2. **При преобразовании в число `null` становится `0`, а `undefined` становится `NaN`.**

Посмотрим забавные следствия.

Некорректный результат сравнения null с 0

Сравним `null` с нулём:

```
alert(null > 0); // false
alert(null == 0); // false
```

Итак, мы получили, что `null` не больше и не равен нулю. А теперь…

```
alert(null >= 0); // true
```

Как такое возможно? Если нечто *«больше или равно нулю»*, то резонно полагать, что оно либо *больше*, либо *равно*. Но здесь это не так.

Дело в том, что алгоритмы проверки равенства `==` и сравнения `>= > < <=` работают по-разному.

Сравнение честно приводит к числу, получается ноль. А при проверке равенства значения `null` и `undefined` обрабатываются особым образом: они равны друг другу, но не равны чему-то ещё.

В результате получается странная с точки зрения здравого смысла ситуация, которую мы видели в примере выше.

Несравнимый undefined

Значение `undefined` вообще нельзя сравнивать:

```
alert(undefined > 0); // false (1)
alert(undefined < 0); // false (2)
alert(undefined == 0); // false (3)
```

- Сравнения `(1)` и `(2)` дают `false` потому, что `undefined` при преобразовании к числу даёт `NaN`. А значение `NaN` по стандарту устроено так, что сравнения `==`, `<`, `>`, `<=`, `>=` и даже `===` с ним возвращают `false`.
- Проверка равенства `(3)` даёт `false`, потому что в стандарте явно прописано, что `undefined` равно лишь `null` и ничему другому.

Вывод: любые сравнения с `undefined/null`, кроме точного `===`, следует делать с осторожностью. Желательно не использовать сравнения `>= > < <=`, во избежание ошибок в коде.

Итого

- В JavaScript есть логические значения `true` (истина) и `false` (ложь). Операторы сравнения возвращают их.
- Строки сравниваются побуквенно.
- Значения разных типов приводятся к числу при сравнении, за исключением строгого равенства `===` (`!==`).
- Значения `null` и `undefined` равны `==` друг другу и не равны ничему другому. В других сравнениях (с участием `>`,`<`) их лучше не использовать, так как они ведут себя не как `0`.

Побитовые операторы

Побитовые операторы интерпретируют операнды как последовательность из 32 битов (нулей и единиц). Они производят операции, используя двоичное представление числа, и возвращают новую последовательность из 32 бит (число) в качестве результата.

Эта глава сложная, требует дополнительных знаний в программировании и не очень важная, вы можете пропустить её.

Формат 32-битного целого числа со знаком

Побитовые операторы в JavaScript работают с 32-битными целыми числами в их двоичном представлении.

Это представление называется «32-битное целое со знаком, старшим битом слева и дополнением до двойки».

Разберём, как устроены числа внутри подробнее, это необходимо знать для битовых операций с ними.

- Что такое двоичная система счисления [5] , вам, надеюсь, уже известно. При разборе побитовых операций мы будем обсуждать именно двоичное представление чисел, из 32 бит.
- *Старший бит слева* — это научное название для самого обычного порядка записи цифр (от большего разряда к меньшему). При этом, если больший разряд отсутствует, то соответствующий бит равен нулю.

  Примеры представления чисел в двоичной системе:

```
a = 0;   // 00000000000000000000000000000000
a = 1;   // 00000000000000000000000000000001
a = 2;   // 00000000000000000000000000000010
a = 3;   // 00000000000000000000000000000011
a = 255;// 00000000000000000000000011111111
```

Обратите внимание, каждое число состоит ровно из 32-битов.

Младший бит слева

Несмотря на то, что нам такой способ записи чисел кажется не совсем обычным, бывают языки и технологии, использующие способ записи «младший бит слева», когда биты пишутся наоборот, от меньшего разряда к большему.

Именно поэтому спецификация EcmaScript явно говорит «старший бит слева».

Дополнение до двойки — это название способа поддержки отрицательных чисел.

Двоичный вид числа, обратного данному (например, `5` и `-5`) получается путём обращения всех битов с прибавлением 1.

То есть, нули заменяются на единицы, единицы — на нули и к числу прибавляется `1`. Получается внутреннее представление того же числа, но со знаком минус.

Например, вот число `314`:

```
00000000000000000000000100111010
```

Чтобы получить `-314`, первый шаг — обратить биты числа: заменить `0` на `1`, а `1` на `0`:

```
11111111111111111111111011000101
```

Второй шаг — к полученному двоичному числу приплюсовать единицу, обычным двоичным сложением:
`11111111111111111111111011000101 + 1 = 11111111111111111111111011000110`.

Итак, мы получили:

```
-314 = 11111111111111111111111011000110
```

Принцип дополнения до двойки делит все двоичные представления на два множества: если крайний-левый бит равен `0` — число положительное, если `1` — число отрицательное. Поэтому этот бит называется *знаковым битом*.

Список операторов

В следующей таблице перечислены все побитовые операторы.
Далее операторы разобраны более подробно.

Оператор	Использование	Описание
Побитовое И (AND)	`a & b`	Ставит 1 на бит результата, для которого соответствующие биты операндов равны 1.
Побитовое ИЛИ (OR)	`a | b`	Ставит 1 на бит результата, для которого хотя бы один из соответствующих битов операндов равен 1.

Побитовое исключающее ИЛИ (XOR)	`a ^ b`	Ставит 1 на бит результата, для которого только один из соответствующих битов операндов равен 1 (но не оба).
Побитовое НЕ (NOT)	`~a`	Заменяет каждый бит операнда на противоположный.
Левый сдвиг	`a << b`	Сдвигает двоичное представление `a` на `b` битов влево, добавляя справа нули.
Правый сдвиг, переносящий знак	`a >> b`	Сдвигает двоичное представление `a` на `b` битов вправо, отбрасывая сдвигаемые биты.
Правый сдвиг с заполнением нулями	`a >>> b`	Сдвигает двоичное представление `a` на `b` битов вправо, отбрасывая сдвигаемые биты и добавляя нули слева.

Описание работы операторов

Побитовые операторы работают следующим образом:

1. Операнды преобразуются в 32-битные целые числа, представленные последовательностью битов. Дробная часть, если она есть, отбрасывается.
2. Для бинарных операторов — каждый бит в первом операнде рассматривается вместе с соответствующим битом второго операнда: первый бит с первым, второй со вторым и т.п. Оператор применяется к каждой паре бит, давая соответствующий бит результата.
3. Получившаяся в результате последовательность бит интерпретируется как обычное число.

Посмотрим, как работают операторы, на примерах.

& (побитовое И)

Выполняет операцию И над каждой парой бит.

Результат `a & b` равен единице только когда оба бита `a` и `b` равны единице.

Таблица истинности для &:

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

Пример:

```
9 (по осн. 10)
  = 00000000000000000000000000001001 (по осн. 2)
14 (по осн. 10)
  = 00000000000000000000000000001110 (по осн. 2)
                  --------------------------------
14 & 9 (по осн. 10)
  = 00000000000000000000000000001000 (по осн. 2)
  = 8 (по осн. 10)
```

| (Побитовое ИЛИ)

Выполняет операцию ИЛИ над каждой парой бит. Результат `a | b` равен 1, если хотя бы один бит из `a,b` равен 1.

Таблица истинности для |:

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

Пример:

```
9 (по осн. 10)
  = 00000000000000000000000000001001 (по осн. 2)
14 (по осн. 10)
  = 00000000000000000000000000001110 (по осн. 2)
                    --------------------------------
14 | 9 (по осн. 10)
  = 00000000000000000000000000001111 (по осн. 2)
  = 15 (по осн. 10)
```

^ (Исключающее ИЛИ)

Выполняет операцию «Исключающее ИЛИ» над каждой парой бит.

a Исключающее ИЛИ b равно 1, если только a=1 или только b=1, но не оба одновременно a=b=1.

Таблица истинности для исключающего ИЛИ:

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

Как видно, оно даёт 1, если ЛИБО слева 1, ЛИБО справа 1, но не одновременно. Поэтому его и называют «исключающее ИЛИ».

Пример:

```
9 (по осн. 10)
  = 00000000000000000000000000001001 (по осн. 2)
14 (по осн. 10)
  = 00000000000000000000000000001110 (по осн. 2)
                    --------------------------------
14 ^ 9 (по осн. 10)
  = 00000000000000000000000000000111 (по осн. 2)
  = 7 (по осн. 10)
```

Исключающее ИЛИ в шифровании

Исключающее или можно использовать для шифрования, так как эта операция полностью обратима. Двойное применение исключающего ИЛИ с тем же аргументом даёт исходное число.

Иначе говоря, верна формула: `a ^ b ^ b == a`.

Пускай Вася хочет передать Пете секретную информацию `data`. Эта информация заранее превращена в число, например строка интерпретируется как последовательность кодов символов.

Вася и Петя заранее договариваются о числовом ключе шифрования `key`.

Алгоритм:

- Вася берёт двоичное представление `data` и делает операцию `data ^ key`. При необходимости `data` бьётся на части, равные по длине `key`, чтобы можно было провести побитовое ИЛИ `^` по всей длине. JavaScript позволяет делать операцию `^` с 32-битными целыми числами, так что `data` нужно разбить на последовательность таких чисел.
- Результат `data ^ key` отправляется Пете, это шифровка.

Например, пусть в `data` очередное число равно `9`, а ключ `key` равен `1220461917`.

```
Данные: 9 в двоичном виде
00000000000000000000000000001001

Ключ: 1220461917 в двоичном виде
01001000101111101100010101011101

Результат операции 9 ^ key:
01001000101111101100010101010100
Результат в 10-ной системе (шифровка):
1220461908
```

- Петя, получив очередное число шифровки `1220461908`, применяет к нему такую же операцию `^ key`.
- Результатом будет исходное число `data`.

В нашем случае:

```
Полученная шифровка в двоичной системе:
9 ^ key = 1220461908
01001000101111101100010101010100

Ключ: 1220461917 в двоичном виде:
01001000101111101100010101011101

Результат операции 1220461917 ^ key:
00000000000000000000000000001001
Результат в 10-ной системе (исходное сообщение):
9
```

Конечно, такое шифрование поддаётся частотному анализу и другим методам дешифровки, поэтому современные алгоритмы используют XOR как одну из важных частей более сложной многоступенчатой схемы.

~ (Побитовое НЕ)

Производит операцию НЕ над каждым битом, заменяя его на обратный ему.

Таблица истинности для НЕ:

a	~a
0	1
1	0

Пример:

```
9 (по осн. 10)
  = 00000000000000000000000000001001 (по осн. 2)
                 --------------------------------
~9 (по осн. 10)
  = 11111111111111111111111111110110 (по осн. 2)
  = -10 (по осн. 10)
```

Из-за внутреннего представления отрицательных чисел получается так, что `~n == -(n+1)`.

Например:

```
alert(~3); // -4
alert(~-1); // 0
```

Операторы битового сдвига

Операторы битового сдвига принимают два операнда. Первый — это число для сдвига, а второй — количество битов, которые нужно сдвинуть в первом операнде.

Направление сдвига — то же, что и направление стрелок в операторе.

<< (Левый сдвиг)

Этот оператор сдвигает первый операнд на указанное число битов влево. Лишние биты отбрасываются, справа добавляются нулевые биты.

Например, `9 << 2` даст `36`:

```
9 (по осн.10)
  = 00000000000000000000000000001001 (по осн.2)
                    --------------------------------
9 << 2 (по осн.10)
  = 00000000000000000000000000100100 (по осн.2)
  = 36 (по осн.10)
```

>> (Правый сдвиг, переносящий знак)

Этот оператор сдвигает биты вправо, отбрасывая лишние. Копии крайнего-левого бита добавляются слева. Так как итоговый крайний-левый бит имеет то же значение, что и исходный, знак числа (представленный крайним-левым битом) не изменяется.

Поэтому он назван «переносящим знак».

Например, `9 >> 2` даст 2:

```
9 (по осн.10)
  = 00000000000000000000000000001001 (по осн.2)
                  --------------------------------
9 >> 2 (по осн.10)
  = 00000000000000000000000000000010 (по осн.2)
  = 2 (по осн.10)
```

Аналогично, `-9 >> 2` даст `-3`, так как знак сохранен:

```
-9 (по осн.10)
  = 11111111111111111111111111110111 (по осн.2)
                   --------------------------------
-9 >> 2 (по осн.10)
  = 11111111111111111111111111111101 (по осн.2) = -3 (по осн.10)
```

`>>>` (Правый сдвиг с заполнением нулями)

Этот оператор сдвигает биты первого операнда вправо. Лишние биты справа отбрасываются. Слева добавляются нулевые биты.

Знаковый бит становится равным 0, поэтому результат всегда положителен.
Для неотрицательных чисел правый сдвиг с заполнением нулями и правый сдвиг с переносом знака дадут одинаковый результат, т.к в обоих случаях слева добавятся нули.

Для отрицательных чисел — результат работы разный. Например, `-9 >>> 2` даст `1073741821`, в отличие от `-9 >> 2` (дает `-3`):

```
-9 (по осн.10)
  = 11111111111111111111111111110111 (по осн.2)
                   --------------------------------
-9 >>> 2 (по осн.10)
  = 00111111111111111111111111111101 (по осн.2)
  = 1073741821 (по осн.10)
```

Применение побитовых операторов

Побитовые операторы используются редко, но всё же используются.

Случаи применения побитовых операторов, которые мы здесь разберём, составляют большинство всех использований в JavaScript.

Осторожно, приоритеты!

В JavaScript побитовые операторы `^`, `&`, `|` выполняются после сравнений `==`.

Например, в сравнении `a == b^0` будет сначала выполнено сравнение `a == b`, а потом уже операция `^0`, как будто стоят скобки `(a == b)^0`.

Обычно это не то, чего мы хотим. Чтобы гарантировать желаемый порядок, нужно ставить скобки: `a == (b^0)`.

Маска

Для этого примера представим, что наш скрипт работает с пользователями:

- `Гость`
- `Редактор`
- `Админ`

У каждого из них есть ряд доступов, которые можно свести в таблицу:

Пользователь	Просмотр статей	Изменение статей	Просмотр товаров	Изменение товаров	Общее администрирование
Гость	Да	Нет	Да	Нет	Нет
Редактор	Да	Да	Да	Да	Нет
Админ	Да	Да	Да	Да	Да

Если вместо «Да» поставить `1`, а вместо «Нет» — `0`, то каждый набор доступов описывается числом:

Пользователь	Просмотр статей	Изменение статей	Просмотр товаров	Изменение товаров	Общее администрирование	В десятичной системе
Гость	1	0	1	0	0	= 20
Редактор	1	1	1	1	0	= 30
Админ	1	1	1	1	1	= 31

Мы «упаковали» много информации в одно число. Это экономит память. Но, кроме этого, по нему очень легко проверить, имеет ли посетитель заданную *комбинацию доступов*.

Для этого посмотрим, как в 2-ной системе представляется каждый доступ в отдельности.

- Доступ, соответствующий только общему администрированию: `00001 (=1)` (все нули кроме 1 на позиции, соответствующей этому доступу).
- Доступ, соответствующий только изменению товаров: `00010 (=2)`.
- Доступ, соответствующий только просмотру товаров: `00100 (=4)`.
- Доступ, соответствующий только изменению статей: `01000 (=8)`.
- Доступ, соответствующий только просмотру статей: `10000 (=16)`.

Например, просматривать и изменять статьи позволит доступ `access = 11000`.

Как правило, доступы задаются в виде констант:

```
var ACCESS_ADMIN = 1;           // 00001
var ACCESS_GOODS_CHANGE = 2;    // 00010
var ACCESS_GOODS_VIEW = 4;      // 00100
var ACCESS_ARTICLE_CHANGE = 8; // 01000
var ACCESS_ARTICLE_VIEW = 16;  // 10000
```

Из этих констант получить нужную комбинацию доступов можно при помощи операции `|`.

```
var access = ACCESS_ARTICLE_VIEW | ACCESS_ARTICLE_CHANGE; // 11000
```

Двоичные числа в JavaScript

Для удобной работы с примерами в этой статье пригодятся две функции.

- `parseInt("11000", 2)` — переводит строку с двоичной записью числа в число.
- `n.toString(2)` — получает для числа `n` запись в 2-ной системе в виде строки.

Например:

```
var access = parseInt("11000", 2);  // получаем число из строки

alert(access); // 24, число с таким 2-ным представлением

var access2 = access.toString(2); // обратно двоичную строку из числа

alert(access2); // 11000
```

Проверка доступов

Для того, чтобы понять, есть ли в доступе `access` нужный доступ, например право администрирования — достаточно применить к нему побитовый оператор И (`&`) с соответствующей маской.

Создадим для примера ряд доступов и проверим их:

```
var access = parseInt("11111", 2); // 31, все 1 означает, что доступ полный

alert(access & ACCESS_ADMIN); // если результат не 0, то есть доступ ACCESS_ADMIN
```

А теперь та же проверка для посетителя с другими правами:

```
var access = parseInt("10100"); // 20, нет 1 в конце

alert(access & ACCESS_ADMIN); // 0, нет доступа к общему администрированию
```

Такая проверка работает, потому что оператор И ставит `1` на те позиции результата, на которых в обоих операндах стоит `1`.

Так что `access & 1` для любого числа `access` поставит все биты в ноль, кроме самого правого. А самый правый станет `1` только если он равен `1` в `access`.

Для полноты картины также проверим, даёт ли доступ `11111` право на изменение товаров. Для этого нужно применить к доступу оператор И (`&`) с `00010` (=2 в 10-ной системе).**

```
var adminAccess = 31; // 11111

alert(adminAccess & ACCESS_GOODS_CHANGE); // не 0, есть доступ к изменению товаров
```

Можно проверить один из нескольких доступов.

Например, проверим, есть ли права на просмотр ИЛИ изменение товаров. Соответствующие права задаются битом `1` на втором и третьем месте с конца, что даёт число `00110` (=6 в 10-ной системе).

```
var check = ACCESS_GOODS_VIEW | ACCESS_GOODS_CHANGE; // 6, 00110

var access = 30; // 11110;

alert(access & check); // не 0, значит есть доступ к просмотру ИЛИ изменению

access = parseInt("11100", 2);

alert(access & check); // не 0, есть доступ к просмотру ИЛИ изменению
```

Как видно из примера выше, если в аргументе `check` стоит ИЛИ из нескольких доступов `ACCESS_*`, то и результат проверки скажет, есть ли хотя бы один из них. А какой — нужно смотреть отдельной проверкой, если это важно.

Итак, маска даёт возможность удобно «паковать» много битовых значений в одно число при помощи ИЛИ `|`, а также, при помощи оператора И (`&`), проверять маску на комбинацию установленных битов.

Маски в функциях

Зачастую маски используют в функциях, чтобы одним параметром передать несколько «флагов», т.е. однобитных значений.

Например:

```
// найти пользователей с правами на изменение товаров или администраторов
findUsers(ACCESS_GOODS_CHANGE | ACCESS_ADMIN);
```

Округление

Так как битовые операции отбрасывают десятичную часть, то их можно использовать для округления. Достаточно взять любую операцию, которая не меняет значение числа.

Например, двойное НЕ (`~`):

```
alert( ~~12.345 ); // 12
```

Подойдёт и Исключающее ИЛИ (`^`) с нулём:

```
alert( 12.345^0 ); // 12
```

Последнее даже более удобно, поскольку отлично читается:

```
alert( 12.3 * 14.5 ^ 0); // (=178) "12.3 умножить на 14.5 и округлить"
```

У побитовых операторов достаточно низкий приоритет, он меньше чем у остальной арифметики:

```
alert( 1.1 + 1.2 ^ 0 ); // 2, сложение выполнится раньше округления
```

Проверка на `-1`

Внутренний формат [6] чисел устроен так, что для смены знака нужно все биты заменить на противоположные («обратить») и прибавить `1`.

Обращение битов — это побитовое НЕ (`~`). То есть, при таком формате представления числа `-n = ~n + 1`. Или, если перенести единицу: `~n = -(n+1)`.

Как видно из последнего равенства, `~n == 0` только если `n == -1`. Поэтому можно легко проверить равенство `n == -1`:

```
var n = 5;

if (~n) {  // сработает, т.к. ~n = -(5+1) = -6
  alert("n не -1"); // выведет!
}
```

```
var n = -1;

if (~n) {  // не сработает, т.к. ~n = -(-1+1) = 0
  alert("...ничего не выведет...");
}
```

Проверка на `-1` пригождается, например, при поиске символа в строке. Вызов `str.indexOf("подстрока")` возвращает позицию подстроки в `str`, или `-1` если не нашёл.

```
var str = "Проверка";

if (~str.indexOf("верка")) { // Сочетание "if (~...indexOf)" читается как "если найдено"
  alert('найдено!');
}
```

Умножение и деление на степени 2

Оператор `a << b`, сдвигая биты, по сути умножает `a` на 2^b.

Например:

```
alert( 1 << 2 ); // 1*(2*2) = 4
alert( 1 << 3 ); // 1*(2*2*2) = 8
alert( 3 << 3 ); // 3*(2*2*2) = 24
```

Оператор `a >> b`, сдвигая биты, производит целочисленное деление `a` на 2^b.

```
alert( 8 >> 2 ); // 2 = 8/4, убрали 2 нуля в двоичном представлении
alert( 11 >> 2 ); // 2, целочисленное деление (менее значимые биты просто отброшены)
```

Итого

- Бинарные побитовые операторы: `& | ^ << >> >>>`.
- Унарный побитовый оператор один: `~`.

Как правило, битовое представление числа используется для:

- Упаковки нескольких битововых значений («флагов») в одно значение. Это экономит память и позволяет проверять наличие комбинации флагов одним оператором &. Кроме того, такое упакованное значение будет для функции всего одним параметром, это тоже удобно.
- Округления числа: `(12.34^0) = 12`.
- Проверки на равенство `-1`: `if (~n) { n не -1 }`.

Взаимодействие с пользователем: alert, prompt, confirm

В этом разделе мы рассмотрим базовые UI операции: `alert`, `prompt` и `confirm`, которые позволяют работать с данными, полученными от пользователя.

alert

Синтаксис:

```
alert(сообщение)
```

`alert` выводит на экран окно с сообщением и приостанавливает выполнение скрипта, пока пользователь не нажмет «ОК».

```
alert("Привет");
```

Окно сообщения, которое выводится, является *модальным окном*. Слово «модальное» означает, что посетитель не может взаимодействовать со страницей, нажимать другие кнопки и т.п., пока не разберется с окном. В данном случае - пока не нажмет на «ОК».

prompt

Функция prompt принимает два аргумента:

```
result = prompt(title, default);
```

Она выводит модальное окно с заголовком `title`, полем для ввода текста, заполненным строкой по умолчанию `default` и кнопками OK/CANCEL.

Пользователь должен либо что-то ввести и нажать OK, либо отменить ввод кликом на CANCEL или нажатием ESC на клавиатуре.

Вызов `prompt` возвращает то, что ввел посетитель - строку или специальное значение `null`, если ввод отменен.

Как и в случае с `alert`, окно `prompt` модальное.

```
var years = prompt('Сколько вам лет?', 100);

alert('Вам ' + years + ' лет!')
```

Всегда указывайте `default`

Вообще, второй `default` может отсутствовать. Однако при этом IE вставит в диалог значение по умолчанию `"undefined"`.

Запустите этот код в IE, чтобы понять о чем речь:

```
var test = prompt("Тест");
```

Поэтому рекомендуется *всегда* указывать второй аргумент:

```
var test = prompt("Тест", ''); // <-- так лучше
```

confirm

Синтаксис:

```
result = confirm(question);
```

`confirm` выводит окно с вопросом `question` с двумя кнопками: OK и CANCEL.

Результатом будет `true` при нажатии OK и `false` - при CANCEL(Esc).

Например:

```
var isAdmin = confirm("Вы - администратор?");

alert(isAdmin);
```

Особенности встроенных функций

Место, где выводится модальное окно с вопросом, и внешний вид окна выбирает браузер. Разработчик не может на это влиять.

С одной стороны — это недостаток, т.к. нельзя вывести окно в своем дизайне.

С другой стороны, преимущество этих функций по сравнению с другими, более сложными методами взаимодействия, которые мы изучим в дальнейшем — как раз в том, что они очень просты.

Это самый простой способ вывести сообщение или получить информацию от посетителя. Поэтому их используют в тех случаях, когда простота важна, а всякие «красивости» особой роли не играют.

Резюме

- `alert` выводит сообщение.
- `prompt` выводит сообщение и ждет, пока пользователь введет текст, а затем возвращает введенное значение или `null`, если ввод отменен (CANCEL/Esc).
- `confirm` выводит сообщение и ждет, пока пользователь нажмет «OK» или «CANCEL» и возвращает `true/false`.

Условные операторы: if, '?'

Иногда, в зависимости от условия, нужно выполнить различные действия. Для этого используется оператор `if`.

Например:

```
var year = prompt('В каком году появилась спецификация ECMA-262 5.1?', '');

if (year != 2011) alert('А вот и неправильно!');
```

Оператор `if`

Оператор `if` («если») получает условие, в примере выше это `year != 2011`. Он вычисляет его, и если результат — `true`, то выполняет команду.

Если нужно выполнить более одной команды — они оформляются блоком кода в фигурных скобках:

```
if (year != 2011) {
  alert('А вот..');
  alert('..и неправильно!');
}
```

Рекомендуется использовать фигурные скобки всегда, даже когда команда одна. Это улучшает читаемость кода.

Преобразование к логическому типу

Оператор `if (...)` вычисляет и преобразует выражение в скобках к логическому типу.

В логическом контексте число `0`, пустая строка `""`, `null` и `undefined`, а также `NaN` являются `false`, остальные значения — `true`.

Например, такое условие никогда не выполнится:

```
if (0) { // 0 преобразуется к false
  ...
}
```

... А такое — выполнится всегда:

```
if (1) { // 1 преобразуется к true
  ...
}
```

Вычисление условия в проверке `if (year != 2011)` может быть вынесено в отдельную переменную:

```
var cond = (year != 2011);  // true/false

if (cond) {
  ...
}
```

Неверное условие, `else`

Необязательный блок `else` («иначе») выполняется, если условие неверно:

```
var year = prompt('Введите год ECMA-262 5.1', '');

if (year == 2011) {
  alert('Да вы знаток!');
} else {
  alert('А вот и неправильно!'); // любое значение, кроме 2011
}
```

Несколько условий, `else if`

Бывает нужно проверить несколько вариантов условия. Для этого используется блок `else if ...`. Например:

```
var year = prompt('В каком году появилась спецификация ECMA-262 5.1?', '');

if (year < 2011) {
  alert('Это слишком рано..');
} else if (year > 2011) {
  alert('Это поздновато..');
} else {
  alert('Да, точно в этом году!');
}
```

В примере выше JavaScript сначала проверит первое условие, если оно ложно — перейдет ко второму — и так далее, до последнего `else`.

Оператор вопросительный знак `'?'`

Иногда нужно в зависимости от условия присвоить переменную. Например:

```
var access;
var age = prompt('Сколько вам лет?', '');

if (age > 14) {
  access = true;
} else {
  access = false;
}

alert(access);
```

Оператор вопросительный знак `'?'` позволяет делать это короче и проще.

Он состоит из трех частей:

```
условие ? значение1 : значение2
```

Проверяется условие, затем если оно верно — возвращается `значение1` , если неверно — `значение2`, например:

```
access = (age > 14) ? true : false;
```

Оператор `'?'` выполняется позже большинства других, в частности — позже сравнений, поэтому скобки можно не ставить:

```
access = age > 14 ? true : false;
```

.. Но когда скобки есть — код лучше читается. Так что рекомендуется их писать.

В данном случае можно было бы обойтись и без оператора `'?'`, т.к. сравнение само по себе уже возвращает `true/false`:

```
access = age > 14;
```

«Тернарный оператор»

Вопросительный знак — единственный оператор, у которого есть аж три аргумента, в то время как у обычных операторов их один-два.
Поэтому его называют *«тернарный оператор»*.

Несколько операторов `'?'`

Несколько операторов `if..else` можно заменить последовательностью операторов `'?'`. Например:

```
var a = prompt('a?', 1);

var res = (a == 1) ? 'значение1' :
  (a == 2) ? 'значение2' :
  (a > 2) ? 'значение3' :
  'значение4';

alert(res);
```

Поначалу может быть сложно понять, что происходит. Однако, внимательно приглядевшись, мы замечаем, что это *обычный `if..else`*!

Вопросительный знак проверяет сначала `a == 1`, если верно — возвращает `значение1`, если нет — идет проверять `a == 2`. Если это верно — возвращает `значение2`, иначе проверка `a > 2` и `значение3`.. Наконец, если ничего не верно, то `значение4`.

Альтернативный вариант с `if..else`:

```
var res;

if (a == 1) {
  res = 'значение1';
} else if (a == 2) {
  res = 'значение2';
} else if (a > 2) {
  res = 'значение3';
} else {
  res = 'значение4';
}
```

Нетрадиционное использование `'?'`

Иногда оператор вопросительный знак `'?'` используют как замену `if`:

```
var company = prompt('Какая компания создала JavaScript?', '');

(company == 'Netscape') ?
   alert('Да, верно') : alert('Неправильно');
```

Работает это так: в зависимости от условия, будет выполнена либо первая, либо вторая часть после `'?'`.

Результат выполнения не присваивается в переменную, так что пропадёт (впрочем, `alert` ничего не возвращает).

Рекомендуется не использовать вопросительный знак таким образом.

Несмотря на то, что с виду такая запись короче `if`, она является существенно менее читаемой.

Вот, для сравнения, то же самое с `if`:

```
var company = prompt('Какая компания создала JavaScript?', '');

if (company == 'Netscape') {
  alert('Да, верно');
} else {
  alert('Неправильно');
}
```

Логические операторы

В JavaScript поддерживаются операторы `||` (ИЛИ), `&&` (И) и `!` (НЕ).

Они называются *«логическими»*, но в JavaScript могут применяться к значениям любого типа и возвращают также значения любого типа.

|| (ИЛИ)

Оператор ИЛИ выглядит как двойной символ вертикальной черты:

```
result = a || b;
```

Логическое ИЛИ в классическом программировании работает следующим образом: «если *хотя бы один* из аргументов `true`, то возвращает `true`, иначе — `false`».

Получается следующая таблица результатов:

```
alert( true  || true ); // true
alert( false || true ); // true
alert( true  || false); // true
alert( false || false); // false
```

При вычислении ИЛИ в JavaScript можно использовать любые значения. В этом случае они будут интерпретироваться как логические.

Например, число `1` будет воспринято как `true`, а `0` — как `false`:

```
if ( 1 || 0 ) { // сработает как if( true || false )
  alert('верно');
}
```

Обычно оператор ИЛИ используется в `if`, чтобы проверить, выполняется ли хотя бы одно из условий, например:

```
var hour = 9;

if (hour < 10 || hour > 18) {
  alert('Офис до 10 или после 18 закрыт');
}
```

Можно передать и больше условий:

```
var hour = 12, isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {
  alert('Офис до 10 или после 18 или в выходной закрыт');
}
```

Короткий цикл вычислений

JavaScript вычисляет несколько ИЛИ слева направо. При этом, чтобы экономить ресурсы, используется так называемый *«короткий цикл вычисления»*.

Допустим, вычисляются несколько ИЛИ подряд: `a || b || c || ...`. Если первый аргумент — `true`, то результат заведомо будет `true` (хотя бы одно из значений — `true`), и остальные значения игнорируются.

Это особенно заметно, когда выражение, переданное в качестве второго аргумента, имеет *сторонний эффект* — например, присваивает переменную.

При запуске примера ниже присвоение `x` не произойдёт:

```
var x;

true || (x = 1);  // просто вычислим ИЛИ, без if

alert(x); // undefined, x не присвоен
```

…А в примере ниже первый аргумент — `false`, так что ИЛИ попытается вычислить второй, запустив тем самым присваивание:

```
var x;

false || (x = 1);
alert(x); // 1
```

Значение ИЛИ

Итак, как мы видим, оператор ИЛИ вычисляет ровно столько значений, сколько необходимо — до первого `true`.

Оператор ИЛИ возвращает то значение, на котором остановились вычисления.

Примеры:

```
alert( 1 || 0 ); // 1
alert( true || 'неважно что'); // true

alert( null || 1 ); // 1
alert( undefined || 0 ); // 0
```

Это используют, в частности, чтобы выбрать первое «истинное» значение из списка:

```
var undef; // переменная не присвоена, т.е. равна undefined
var zero = 0;
var emptyStr = "";
var msg = "Привет!";

var result = undef || zero || emptyStr || msg || 0;

alert(result) // выведет "Привет!" - первое значение, которое является true
```

&& (И)

Оператор И пишется как два амперсанда &&:

```
result = a && b;
```

В классическом программировании И возвращает `true`, если оба аргумента истинны, а иначе — `false`

```
alert( true  && true ); // true
alert( false && true ); // false
alert( true  && false); // false
alert( false && false); // false
```

Пример:

```
var hour = 12, minute = 30;

if (hour == 12 && minute == 30) {
  alert('Время 12:30');
}
```

Как и в ИЛИ, допустимы любые значения:

```
if ( 1 && 0 ) { // вычислится как true && false
  alert('не сработает, т.к. условие ложно');
}
```

К И применим тот же принцип «короткого цикла вычислений», но немного по-другому, чем к ИЛИ.

Если левый аргумент — `false`, оператор И возвращает его и заканчивает вычисления. Иначе — вычисляет и возвращает правый аргумент.

Например:

```
// Первый аргумент - true,
// Поэтому возвращается второй аргумент
alert(1 && 0);  // 0
alert(1 && 5);  // 5

// Первый аргумент - false,
// Он и возвращается, а второй аргумент игнорируется
alert(null && 5);  // null
alert(0 && "не важно"); // 0
```

Приоритет оператора И `&&` больше, чем ИЛИ `||`, т.е. он выполняется раньше.

Поэтому в следующем коде сначала будет вычислено правое И: `1 && 0 = 0`, а уже потом — ИЛИ.

```
alert(5 || 1 && 0); // 5
```

Не используйте && вместо if

Оператор `&&` в простых случаях можно использовать вместо `if`, например:

```
var x = 1;

(x > 0) && alert('Больше');
```

Действие в правой части `&&` выполнится только в том случае, если до него дойдут вычисления. То есть, если в левой части будет `true`.

Получился аналог:

```
var x = 1;

if (x > 0) {
  alert('Больше');
}
```

Однако, как правило, `if` лучше читается и воспринимается. Он более очевиден, поэтому лучше использовать его. Это, впрочем, относится и к другим неочевидным применениям возможностей языка.

! (НЕ)

Оператор НЕ — самый простой. Он получает один аргумент. Синтаксис:

```
var result = !value;
```

Действия `!`:

1. Сначала приводит аргумент к логическому типу `true/false`.
2. Затем возвращает противоположное значение.

Например:

```
alert( !true ) // false
alert( !0 )  // true
```

В частности, двойное НЕ используются для преобразования значений к логическому типу:

```
alert( !!"строка" )  // true
alert( !!null ) // false
```

Циклы while, for

При написании скриптов зачастую встает задача сделать однотипное действие много раз.

Например, вывести товары из списка один за другим. Или просто перебрать все числа от 1 до 10 и для каждого выполнить одинаковый код.

Для многократного повторения одного участка кода - предусмотрены *циклы*.

Цикл `while`

Цикл `while` имеет вид:

```
while (условие) {
  // код, тело цикла
}
```

Пока `условие` верно — выполняется код из тела цикла.

Например, цикл ниже выводит `i` пока `i < 3`:

```
1 var i = 0;
2 while (i < 3) {
3   alert(i);
4   i++;
5 }
```

Повторение цикла по-научному называется «итерация». Цикл в примере выше совершает три итерации.

Если бы `i++` в коде выше не было, то цикл выполнялся бы (в теории) вечно. На практике, браузер выведет сообщение о «зависшем» скрипте и посетитель его остановит.

Бесконечный цикл можно сделать и проще:

```
while (true) {
  // ...
}
```

Условие в скобках интерпретируется как логическое значение, поэтому вместо `while (i!=0)` обычно пишут `while (i)`:

```
1 var i = 3;
2 while (i) { // при i=0 значение в скобках будет false и цикл остановится
3   alert(i);
4   i--;
5 }
```

Цикл `do..while`

Проверку условия можно поставить *под* телом цикла, используя специальный синтаксис `do..while`:

```
do {
  // тело цикла
} while (условие);
```

Цикл, описанный, таким образом, сначала выполняет тело, а затем проверяет условие.

Например:

```
var i = 0;
do {
  alert(i);
  i++;
} while (i < 3);
```

Синтаксис `do..while` редко используется, т.к. обычный `while` нагляднее — в нём не приходится искать глазами условие и ломать голову, почему оно проверяется именно в конце.

Цикл `for`

Чаще всего применяется цикл `for`. Выглядит он так:

```
for (начало; условие; шаг) {
 // ... тело цикла ...
}
```

Например, цикл ниже выводит значения от `0` до `3` (не включая `3`):

```
var i;

for (i=0; i<3; i++) {
  alert(i);
}
```

- *Начало* `i=0` выполняется при заходе в цикл.
- *Условие* `i<3` проверяется перед каждой итерацией.
- *Шаг* `i++` выполняется после каждой итерации, но перед проверкой условия.

В цикле также можно определить переменную:

```
for (var i=0; i<3; i++) {
  ...
}
```

Любая часть `for` может быть пропущена.

Например, можно убрать `начало`:

```
var i = 0;
for (; i<3; i++) ...
```

Можно убрать и шаг:

```
var i = 0;
for (; i<3; ) {
 // цикл превратился в аналог while (i<3)
}
```

А можно и вообще убрать все, получив бесконечный цикл:

```
for (;;) {
  // будет выполняться вечно
}
```

При этом сами точки с запятой `;` обязательно должны присутствовать, иначе будет ошибка синтаксиса.

`for..in`

Существует также специальная конструкция `for..in` для перебора свойств объекта.

Мы познакомимся с ней позже, когда будем говорить об объектах [7].

Директивы break и continue

Для более гибкого управления циклом используются директивы `break` и `continue`.

Выход: `break`

Выйти из цикла можно не только при проверке условия но и, вообще, в любой момент. Эту возможность обеспечивает директива `break`.

Например, бесконечный цикл в примере прекратит выполнение при `i==5`:

```
01 var i=0;
02
03 while(1) {
04   i++;
05
06   if (i==5) break;
07
08   alert(i);
09 }
10
11 alert('Последняя i = '+ i ); // 5 (*)
```

Выполнение продолжится со строки `(*)`, следующей за циклом.

Следующая итерация: `continue`

Директива `continue` прекращает выполнение *текущей итерации* цикла. Например, цикл ниже не выводит четные значения:

```
1 for (var i = 0; i < 10; i++) {
2
3   if (i % 2 == 0) continue;
4
5   alert(i);
6 }
```

Для четных `i` срабатывает `continue`, выполнение блока прекращается и управление передается на `for`.

Совет по стилю

Как правило, `continue` и используют, чтобы не обрабатывать определенные значения в цикле.

Цикл, который обрабатывает только часть значений, мог бы выглядеть так:

```
01 for (var i = 0; i < 10; i++) {
02
03   if ( checkValue(i) ) {
04     // функция checkValue проверяет, подходит ли i
05
06     // ...
07     // ... обработка
08     // ... этого
09     // ... значения
10     // ... цикла
11     // ...
12
13   }
14 }
```

Все хорошо, но мы получили *дополнительный уровень вложенности фигурных скобок, без которого можно и нужно обойтись*.

Гораздо лучше здесь использовать `continue`:

```
01 for (var i = 0; i < 10; i++) {
02
03   if ( !checkValue(i) ) continue;
04
05   // здесь мы точно знаем, что i подходит
06
07   // ...
08   // ... обработка
09   // ... этого
10   // ... значения
11   // ... цикла
12   // ...
13
14 }
```

Нельзя использовать break/continue справа от оператора ‘?’

Обычно мы можем заменить `if` на оператор вопросительный знак `'?'`.

То есть, запись:

```
if (условие) {
  a();
} else {
  b();
}
```

..Аналогична записи:

```
условие ? a() : b();
```

В обоих случаях в зависимости от условия выполняется либо `a()` либо `b()`.

Но разница состоит в том, что оператор вопросительный знак `'?'`, использованный во второй записи, возвращает значение.

Синтаксические конструкции, которые не возвращают значений, нельзя использовать в операторе `'?'`. К таким относятся большинство конструкций и, в частности, `break/continue`.

Поэтому такой код приведёт к ошибке:

```
(i > 5) ? alert(i) : continue;
```

Метки

Бывает нужно выйти одновременно из нескольких уровней цикла.

Представим, что нужно ввести значения точек. У каждой точки есть две координаты `(i, j)`. Цикл для ввода значений `i,j = 0..2` может выглядеть так:

```
for (var i = 0; i < 3; i++) {

  for (var j = 0; j < 3; j++) {

    var input = prompt("Значение в координатах " + i + "," + j, "");

    if (input == null) break; // (*)

  }
}
alert('Готово!');
```

Здесь `break` используется, чтобы прервать ввод, если посетитель нажал на `Отмена`. Но обычный вызов `break` в строке `(*)` не может прервать два цикла сразу. Как же прервать ввод полностью? Один из способов — поставить *метку*.

Метка имеет вид `"имя:"`, имя должно быть уникальным. Она ставится перед циклом, вот так:

```
outer: for (var i = 0; i < 3; i++) { ... }
```

Можно также выносить ее на отдельную строку. Вызов `break outer` прерывает управление цикла с такой меткой, вот так:

```
outer:
for (var i = 0; i < 3; i++) {

  for (var j = 0; j < 3; j++) {

    var input = prompt('Значение в координатах '+i+','+j, '');

    if (input == null) break outer; // (*)

  }
}
alert('Готово!');
```

Директива `continue` также может быть использована с меткой. Управление перепрыгнет на следующую итерацию цикла с меткой.

Метки можно ставить в том числе на блок, без цикла:

```
my: {

  for (;;) {
    for (i=0; i<10; i++) {
      if (i>4) break my;
    }
  }

  some_code; // произвольный участок кода

}
alert("После my"); // (*)
```

В примере выше, `break` перепрыгнет через `some_code`, выполнение продолжится сразу после блока `my`, со строки `(*)`. Возможность ставить метку на блоке используется редко. Обычно метки ставятся перед циклом.

Goto?

В некоторых языках программирования есть оператор `goto`, который может передавать управление на любой участок программы.

Операторы `break/continue` более ограниченны. Они работают только внутри циклов, и метка должна быть не где угодно, а выше по уровню вложенности.

В JavaScript нет `goto`.

Конструкция switch

Конструкция `switch` заменяет собой сразу несколько `if`.

Это — более наглядный способ сравнить выражение сразу с несколькими вариантами.

Синтаксис

Выглядит она так:

```
switch(x) {
  case 'value1':  // if (x === 'value1')
    ...
    [break]

  case 'value2':  // if (x === 'value2')
    ...
    [break]

  default:
    ...
    [break]
}
```

- Переменная `x` проверяется на строгое равенство первому значению `value1`, затем второму `value2` и так далее.
- Если соответствие установлено — switch начинает выполняться от соответствующей директивы `case` и далее, *до ближайшего `break`* (или до конца `switch`).

  При этом `case` называют *вариантами `switch`*.
- Если ни один `case` не совпал — выполняется (если есть) вариант `default`.

Пример работы

Пример использования `switch` (сработавший код выделен):

```
var a = 2+2;

switch (a) {
  case 3:
    alert('Маловато');
    break;
  case 4:
    alert('В точку!');
    break;
  case 5:
    alert('Перебор');
    break;
  default:
    alert('Я таких значений не знаю');
}
```

Будет выведено только одно значение, соответствующее `4`. После чего `break` прервёт выполнение.

Если его не прервать — оно пойдёт далее, при этом остальные проверки игнорируются.

Например:

```
var a = 2+2;

switch (a) {
  case 3:
    alert('Маловато');
  case 4:
    alert('В точку!');
  case 5:
    alert('Перебор');
  default:
    alert('Я таких значений не знаю');
}
```

В примере выше последовательно выполнятся три `alert`.

```
alert('В точку!');
alert('Перебор');
alert('Я таких значений не знаю');
```

В `case` могут быть любые выражения, в том числе включающие в себя переменные и функции.

Например:

```
var a = 1;
var b = 0;

switch(a) {
  case b+1:
    alert(1);
    break;

  default:
    alert('нет-нет, выполнится вариант выше')
}
```

Группировка `case`

Несколько значений case можно группировать.

В примере ниже `case 3` и `case 5` выполняют один и тот же код:

```
var a = 2+2;

switch (a) {
  case 4:
    alert('Верно!');
    break;

  case 3:                     // (*)
  case 5:                     // (**)
    alert('Неверно!');
    break;

  default:
    alert('Я таких значений не знаю');
}
```

При `case 3` выполнение идёт со строки `(3)` и идёт вниз до ближайшего `break`, таким образом проходя и то, что предназначено для `case 5`.

Тип имеет значение

Следующий пример принимает значение от посетителя.

```
var arg = prompt("Введите arg?")
switch(arg) {
  case '0':
  case '1':
    alert('Один или ноль');

  case '2':
    alert('Два');
    break;

  case 3:
    alert('Никогда не выполнится');

  case null:
    alert('Отмена');
    break;

  default:
    alert('Неизвестное значение: ' + arg)
}
```

Что оно выведет при вводе чисел 0, 1, 2, 3? Подумайте и *потом* читайте дальше…

- При вводе `0` или `1` выполнится первый `alert`, далее выполнение продолжится вниз до первого `break` и выведет второй `alert('Два')`.
- При вводе `2`, `switch` перейдет к `case '2'` и выведет `Два`.
- **При вводе `3`, `switch` перейдет на `default`.** Это потому, что `prompt` возвращает строку `'3'`, а не число. Типы разные. `Switch` использует строгое равенство `===`, так что совпадения не будет.
- При отмене сработает `case null`.

Функции

Зачастую нам надо повторять одно и то же действие во многих частях программы.

Например, красиво вывести сообщение необходимо при приветствии посетителя, при выходе посетителя с сайта, еще где-нибудь.

Чтобы не повторять один и тот же код во многих местах, придуманы функции. Функции являются основными «строительными блоками» программы.

Примеры встроенных функций вы уже видели — это `alert(message)`, `prompt(message, default)` и `confirm(question)`. Но можно создавать и свои.

Объявление

Пример объявления функции:

```
function showMessage() {
  alert('Привет всем присутствующим!');
}
```

Вначале идет ключевое слово `function`, после него *имя функции*, затем *список параметров* в скобках (в примере выше он пустой) и *тело функции* — код, который выполняется при её вызове.

Объявленная функция доступна по имени, например:

```
function showMessage() {
  alert('Привет всем присутствующим!');
}

showMessage();
showMessage();
```

Этот код выведет сообщение два раза. Уже здесь видна **главная цель создания функций: избавление от дублирования кода**.

Если понадобится поменять сообщение или способ его вывода — достаточно изменить его в одном месте: в функции, которая его выводит.

Локальные переменные

Функция может содержать *локальные* переменные, объявленные через `var`. Такие переменные видны только внутри функции:

```
function showMessage() {
  var message = 'Привет, я - Вася!'; // локальная переменная

  alert(message);
}

showMessage(); // 'Привет, я - Вася!'

alert(message); // <-- будет ошибка, т.к. переменная видна только внутри
```

Блоки `if/else`, `switch`, `for`, `while`, `do..while` не влияют на область видимости переменных.

При объявлении переменной в таких блоках, она всё равно будет видна во всей функции.

Например:

```
function count() {
  for (var i=0; i<3; i++) {
    var j = i * 2;
  }

  alert(i); // i=3, на этом значении цикл остановился
  alert(j); // j=4, последнее значение, на котором цикл сработал, было i=2
}
```

Неважно, где именно в функции и сколько раз объявляется переменная. Любое объявление срабатывает один раз и распространяется на всю функцию.

Объявления переменных в примере выше можно передвинуть вверх, это ни на что не повлияет:

```
function count() {
  var i, j; // передвинули объявления var в начало
  for (i=0; i<3; i++) {
    j = i * 2;
  }

  alert(i); // i=3
  alert(j); // j=4
}
```

Внешние переменные

Функция может обратиться ко внешней переменной, например:

```
var userName = 'Вася';

function showMessage() {
  var message = 'Привет, я ' + userName;
  alert(message);
}

showMessage(); // Привет, я Вася
```

Доступ возможен не только на чтение, но и на запись. При этом, так как переменная внешняя, то изменения будут видны и снаружи функции:

```
var userName = 'Вася';

function showMessage() {
  userName = 'Петя'; // (1) присвоение во внешнюю переменную
  var message = 'Привет, я ' + userName;
  alert(message);
}

showMessage();

alert(userName); // Петя, значение внешней переменной изменено функцией
```

Конечно, если бы внутри функции, в строке `(1)`, была бы объявлена своя локальная переменная `var userName`, то все обращения

использовали бы её, и внешняя переменная осталась бы неизменной.

Переменные, объявленные на уровне всего скрипта, называют *«глобальными переменными»*.

Делайте глобальными только те переменные, которые действительно имеют общее значение для вашего проекта.

Пусть каждая функция работает «в своей песочнице».

Внимание: неявное объявление глобальных переменных!

В старом стандарте JavaScript существовала возможность неявного объявления переменных присвоением значения.

Например:

```
function showMessage() {
  message = 'Привет'; // без var!
}

showMessage();

alert(message); // Привет
```

В коде выше переменная `message` нигде не объявлена, а сразу присваивается. Скорее всего, программист просто забыл поставить `var`.

В современном стандарте JavaScript такое присвоение запрещено, а в старом, который работает в браузерах по умолчанию, переменная будет создана автоматически, причём в примере выше она создаётся не в функции, а на уровне всего скрипта.

Избегайте этого.

Здесь опасность даже не в автоматическом создании переменной, а в том, что глобальные переменные должны использоваться тогда, когда действительно нужны «общескриптовые» параметры.

Забыли `var` в одном месте, потом в другом — в результате две функции неожиданно друг для друга поменяли одну и ту же глобальную переменную.

В будущем, когда мы лучше познакомимся с основами JavaScript, в главе Замыкания, функции изнутри [8], мы более детально рассмотрим внутренние механизмы работы переменных и функций.

Параметры

При вызове функции ей можно передать данные, которые та использует по своему усмотрению.

Например, этот код выводит два сообщения:

```
function showMessage(from, text) { // параметры from, text

  from = "** " + from + " **"; // здесь может быть сложный код оформления
  alert(from + '\n\n' + text);
}

showMessage('Маша', 'Привет!');
showMessage('Маша', 'Как дела?');
```

Параметры копируются в локальные переменные функции.

В примере ниже изменение `from` в строке `(1)` не отразится на значении *внешней* переменной `from (2)`, т.к. изменена была *копия значения*:

```
function showMessage(from, text) {
  from = '**' + from + '**';  // (1), красиво оформили from
  alert(from + '\n\n' + text);
}

var from = 'Маша', msg = 'Привет!'; // (2)

showMessage(from, msg); // значения будут скопированы в параметры
alert(from); // перезапись в строке (1) не повлияет на внешнюю переменную
```

Аргументы по умолчанию

Функцию можно вызвать с любым количеством аргументов.

Например, функцию показа сообщения `showMessage(from, text)` можно вызвать с одним аргументом:

```
showMessage("Маша");
```

Если параметр не передан при вызове — он считается равным `undefined`.

Такую ситуацию можно отловить и назначить значение «по умолчанию»:

```
function showMessage(from, text) {
  if (text === undefined) {
    text = 'текст не передан';
  }

  alert(from + ": " + text);
}

showMessage("Маша", "Привет!"); // Маша: Привет!
showMessage("Маша"); // Маша: текст не передан
```

При объявлении функции необязательные аргументы, как правило, располагают в конце списка.

Для указания значения «по умолчанию», то есть, такого, которое используется, если аргумент не указан, используется два способа:

1. Можно проверить, равен ли аргумент `undefined`, и если да — то записать в него значение по умолчанию. Этот способ продемонстрирован в примере выше.
2. Использовать оператор `||`:

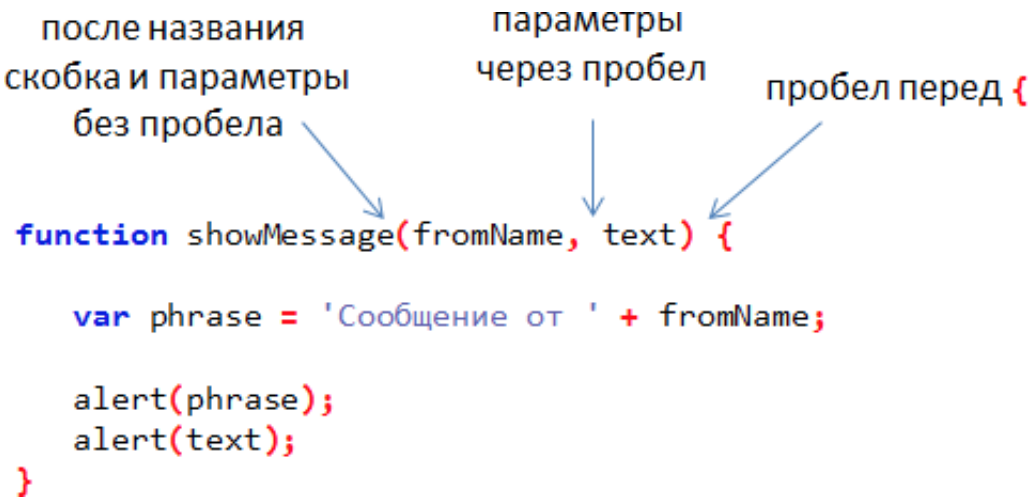
```
function showMessage(from, text) {
  text = text || 'текст не передан';

  ...
}
```

Второй способ считает, что аргумент отсутствует, если передана пустая строка, `0`, или вообще любое значение, которое в булевом виде является `false`.

Если аргументов передано больше, чем надо, например `showMessage("Маша", "привет", 1, 2, 3)`, то ошибки не будет. Но так как для «лишних» аргументов не предусмотрены параметры, то доступ к ним можно будет получить только через специальный объект `arguments`, который мы рассмотрим в главе Псевдо-массив arguments [9].

Стиль объявления функций

В объявлении функции есть правила для расстановки пробелов. Они отмечены стрелочками:



Конечно, вы можете ставить пробелы и по-другому, но эти правила используются в большинстве JavaScript-фреймворков.

Возврат значения

Функция может возвратить результат, который будет передан в вызвавший её код.

Например, создадим функцию `calcD`, которая будет возвращать дискриминант квадратного уравнения по формуле b^2 - 4ac:

```
function calcD(a, b, c) {
   return b*b - 4*a*c;
}

var test = calcD(-4, 2, 1);
alert(test); // 20
```

Для возврата значения используется директива `return`.

Она может находиться в любом месте функции. Как только до нее доходит управление — функция завершается и значение передается обратно.

Вызовов `return` может быть и несколько, например:

```
function checkAge(age) {
  if (age > 18) {
    return true;
  } else {
    return confirm('Родители разрешили?');
  }
}

var age = prompt('Ваш возраст?');

if (checkAge(age)) {
  alert('Доступ разрешен');
} else {
  alert('В доступе отказано');
}
```

Директива `return` может также использоваться без значения, чтобы прекратить выполнение и выйти из функции.

Например:

```
function showMovie(age) {
  if (!checkAge(age)) {
    return;
  }

  alert("Фильм не для всех"); // (*)
  // ...
}
```

В коде выше, если сработал `if`, то строка `(*)` и весь код под ней никогда не выполнится, так как `return` завершает выполнение функции.

Значение функции без `return` и с пустым `return`

В случае, когда функция не вернула значение или `return` был без аргументов, считается что она вернула `undefined`:

```
function doNothing() { /* пусто */ }

alert( doNothing() ); // undefined
```

Обратите внимание, никакой ошибки нет. Просто возвращается `undefined`.

Ещё пример, на этот раз с `return` без аргумента:

```
function doNothing() {
  return;
}

alert( doNothing() === undefined ); // true
```

Выбор имени

Имя функции следует тем же правилам, что и имя переменной. Основное отличие — оно должно быть глаголом, т.к. функция — это действие.

Как правило, используются глагольные префиксы, обозначающие общий характер действия, после которых следует уточнение.

Функции, которые начинаются с `"show"` — что-то показывают:

```
showMessage(..)     // префикс show, "показать" сообщение
```

Функции, начинающиеся с `"get"` — получают, и т.п.:

```
getAge(..)          // get, "получает" возраст
calcD(..)           // calc, "вычисляет" дискриминант
createForm(..)      // create, "создает" форму
checkPermission(..) // check, "проверяет" разрешение, возвращает true/false
```

Это очень удобно, поскольку взглянув на функцию — мы уже примерно представляем, что она делает, даже если функцию написал совсем другой человек, а в отдельных случаях — и какого вида значение она возвращает.

Одна функция — одно действие

Функция должна делать только то, что явно подразумевается её названием. И это должно быть одно действие.

Если оно сложное и подразумевает поддействия — может быть имеет смысл выделить их в отдельные функции? Зачастую это имеет смысл, чтобы лучше структурировать код.

...Но самое главное — в функции не должно быть ничего, кроме самого действия и поддействий, неразрывно связанных с ним.

Например, функция проверки данных (скажем, `"validate"`) не должна показывать сообщение об ошибке. Её действие — проверить.

Сверхкороткие имена функций

Имена функций, которые используются *очень часто*, иногда делают сверхкороткими.

Например, во фреймворке jQuery [10] есть функция `$`, во фреймворке Prototype [11] — функция `$$`, а в библиотеке Underscore очень активно используется функция с названием из одного символа подчеркивания `_`.

Итого

Объявление функции:

```
function имя(параметры, через, запятую) {
  код функции
}
```

- Передаваемые значения копируются в параметры функции и становятся локальными переменными.
- Параметры функции являются её локальными переменными.

- Можно объявить новые локальные переменые при помощи `var`.
- Значение возвращается оператором `return ...`.
- Вызов `return` тут же прекращает функцию.
- Если `return;` вызван без значения, или функция завершилась без `return`, то её результат равен `undefined`.

При обращении к необъявленной переменной функция будет искать внешнюю переменную с таким именем.

По возможности, рекомендуется использовать локальные переменные и параметры:

- Это делает очевидным общий поток выполнения — что передаётся в функцию и какой получаем результат.
- Это предотвращает возможные конфликты доступа, когда две функции, возможно написанные в разное время или разными людьми, неожиданно используют одну и ту же внешнюю переменную.

Именование функций:

- Имя функции должно понятно и чётко отражать, что она делает. Увидев её вызов в коде, вы должны тут же понимать, что она делает.
- Функция — это действие, поэтому для имён функций, как правило, используются глаголы.

Функции являются основными строительными блоками скриптов. Мы будем неоднократно возвращаться к ним и изучать все более и более глубоко.

Рекурсия, стек

В коде функции могут вызывать другие функции для выполнения подзадач. Частный случай подвызова — когда функция вызывает сама себя. Это называется *рекурсией*.

В этой главе мы рассмотрим, как рекурсия устроена изнутри, и как её можно использовать.

Реализация `pow(x, n)` через рекурсию

Чтобы возвести `x` в натуральную степень `n` — можно умножить его на себя `n` раз в цикле:

```
function pow(x, n) {
  var result = x;
  for(var i=1; i<n; i++) {
    result *= x;
  }

  return result;
}
```

А можно поступить проще.

Ведь $x^n = x * x^{n-1}$, т.е. можно вынести один `x` из-под степени. Таким образом, значение функции `pow(x,n)` получается из `pow(x, n-1)` умножением на `x`.

Этот процесс можно продолжить. Например, вычислим `pow(2, 4)`:

```
pow(2, 4) = 2 * pow(2, 3);
pow(2, 3) = 2 * pow(2, 2);
pow(2, 2) = 2 * pow(2, 1);
pow(2, 1) = 2;
```

…То есть, для степени `pow(2, n)` мы получаем результат как `2 * pow(2, n-1)`, затем уменьшаем `n` ещё на единицу и так далее. Этот процесс останавливается на `n==1`, так как очевидно, что `pow(x,1) == x`.

Код для такого вычисления:

```
function pow(x, n) {
  // пока n!=1, сводить вычисление pow(..,n) к pow(..,n-1)
  return (n != 1) ? x*pow(x, n-1) : x;
}

alert( pow(2, 3) ); // 8
```

Говорят, что «функция `pow` *рекурсивно вызывает сама себя*».

Значение, на котором рекурсия заканчивается называют *базисом рекурсии*. В примере выше базисом является `1`.

Общее количество вложенных вызовов называют *глубиной рекурсии*. В случае со степенью, всего будет `n` вызовов. Максимальная глубина рекурсии ограничена и составляет около `10000`, но это число зависит от браузера и может быть в 10 раз меньше.

Рекурсию используют, когда вычисление функции можно свести к её более простому вызову, а его — еще к более простому, и так далее, пока значение не станет очевидно.

Контекст выполнения, стек

Теперь мы посмотрим, как работают рекурсивные вызовы.

У каждого вызова функции есть свой «контекст выполнения» (execution context).

Контекст выполнения включает в себя локальные переменные функции и другую служебную информацию, необходимую для её текущего выполнения.

При любом вызове функции интерпретатор переключает контекст на новый. Этот процесс состоит из нескольких шагов:

1. Текущая функция приостанавливается.
2. Информация о её выполнении, то есть текущий контекст заносится в специальную внутреннюю структуру данных: «стек контекстов».
3. Запускается новая функция, для нее создаётся свой контекст.
4. По завершении подвызова предыдущий контекст достаётся из стека, выполнение в нём возобновляется.

Например, для вызова:

```
function pow(x, n) {
  return (n != 1) ? x*pow(x, n-1) : x;
}

alert( pow(2, 3) );
```

Просходит следующее:

`pow(2, 3)`

Запускается функция `pow`, с аргументами `x=2`, `n=3`. Эти переменные хранятся в контексте выполнения, схематично изображённом ниже:

- Контекст: { x: 2, n: 3 }

`pow(2, 2)`

В строке `2` происходит вызов `pow`, с аргументами `x=2`, `n=2`. Для этой функции создаётся новый текущий контекст (выделен красным), а предыдущий сохраняется в «стеке»:

- Контекст: { x: 2, n: 3 }
- Контекст: { x: 2, n: 2 }

`pow(2, 1)`

Опять вложенный вызов в строке `2`, на этот раз — с аргументами `x=2`, `n=1`. Создаётся новый текущий контекст, предыдущий добавляется в стек:

- Контекст: { x: 2, n: 3 }
- Контекст: { x: 2, n: 2 }
- Контекст: { x: 2, n: 1 }

Выход из `pow(2, 1)`.

При вызове `pow(2, 1)` вложенных вызовов нет. Функция для `n==1` тут же заканчивает свою работу, возвращая `2`. Текущий контекст больше не нужен и удаляется из памяти, из стека восстанавливается предыдущий:

- Контекст: { x: 2, n: 3 }
- Контекст: { x: 2, n: 2 }

Возобновляется обработка внешнего вызова `pow(2, 2)`.

Выход из `pow(2, 2)`.

…И теперь уже `pow(2, 2)` может закончить свою работу, вернув `4`. Восстанавливается контекст предыдущего вызова:

- Контекст: { x: 2, n: 3 }

Возобновляется обработка внешнего вызова `pow(2, 3)`.

Выход из `pow(2, 3)`.

Самый внешний вызов заканчивает свою работу, его результат: `pow(2, 3) = 8`.

Глубина рекурсии в данном случае составила: **3**.

Как видно из иллюстраций выше, глубина рекурсии равна максимальному числу контекстов, одновременно хранимых в стеке.

В самом конце, как и в самом начале, выполнение попадает во внешний код, который находится вне любых функций. У такого кода тоже есть контекст. Его называют «глобальный контекст», и он является начальной и конечной точкой любых вложенных подвызовов.

Обратим внимание на требования к памяти. Фактически, рекурсия приводит к хранению всех данных для внешних вызовов в стеке. То есть, возведение в степень `n` хранит в памяти `n` различных контекстов.

Реализация степени через цикл гораздо более экономна:

```
function pow(x, n) {
  var result = x;
  for(var i=1; i<n; i++) {
    result *= x;
  }
  return result;
}
```

У такой функции `pow` будет один контекст, в котором будут последовательно меняться значения `i` и `result`.

Любая рекурсия может быть переделана в цикл. Как правило, вариант с циклом будет эффективнее.

…Но зачем тогда нужна рекурсия? Да просто затем, что рекурсивный код может быть гораздо проще и понятнее! Переделка в цикл может быть нетривиальной, особенно когда в функции, в зависимости от условий, используются разные рекурсивные подвызовы.

В программировании мы в первую очередь стремимся сделать сложное простым, а повышенная производительность нужна… Лишь там, где она действительно нужна. Поэтому красивое рекурсивное решение во многих случаях лучше.

Недостатки и преимущества рекурсии:

- Требования к памяти.
- Ограничена максимальная глубина стека.
- Краткость и простота кода.

Задачи на рекурсию

Существуют много областей применения рекурсивных вызовов, в частности — работа со структурами данных, такими как дерево документа. Мы обязательно встретимся с ними.

Методы и свойства

Все значения в JavaScript, за исключением `null` и `undefined`, содержат набор вспомогательных функций и значений, доступных «через точку».

Такие функции называют «методами», а значения — «свойствами». Посмотрим на примеры.

Пример: `str.length`, `str.toUpperCase()`

У строки есть *свойство* `length`, содержащее длину:

```
alert( "Привет, мир!".length ); // 12
```

Еще у строк есть *метод* `toUpperCase()`, который возвращает строку в верхнем регистре:

```
var hello = "Привет, мир!";

alert( hello.toUpperCase() ); // "ПРИВЕТ, МИР!"
```

Если через точку вызывается функция (`toUpperCase()`) — это называют «вызов метода», если просто читаем значение (`length`) — «получение свойства».

Пример: `num.toFixed`

У чисел есть метод `num.toFixed(n)`. Он округляет число `num` до `n` знаков после запятой, при необходимости добивает нулями до данной длины и возвращает в виде строки (удобно для форматированного вывода):

```
var n = 12.345;

alert( n.toFixed(2) ); // "12.35"
alert( n.toFixed(0) ); // "12"
alert( n.toFixed(5) ); // "12.34500"
```

Детали работы `toFixed` разобраны в главе Числа [12].

Обращение к методам чисел

К методу числа можно обратиться и напрямую:

```
alert( 12.34.toFixed(1) ); // 12.3
```

...Но если число целое, то будет проблема:

```
alert( 12.toFixed(1) ); // ошибка!
```

Ошибка произойдёт потому, что JavaScript ожидает десятичную дробь после точки.

Это — особенность синтаксиса JavaScript. Вот так — будет работать:

```
alert( 12..toFixed(1) ); // 12.0
```

Вызов метода — через круглые скобки!

Обратите внимание, для вызова метода после его названия идут скобки: `hello.toUpperCase()`. Без скобок метод вызван не будет.

Посмотрим, к примеру, результат обращения к `toUpperCase` без скобок:

```
1 var hello = "Привет";
2
3 alert( hello.toUpperCase ); // function...
```

Этот код выводит значение свойства `toUpperCase`, которое является встроенной в язык функцией. Как правило браузер выводит его как-то так: `"function toUpperCase() { [native code] }"`.

Для получения результата эту функцию необходимо вызвать, и как раз для этого в JavaScript обязательны скобки:

```
1 var hello = "Привет";
2
3 alert( hello.toUpperCase() ); // "ПРИВЕТ" (результат вызова)
```

Мы еще встретимся со строками [13] и числами [14] в последующих главах и глубже познакомимся со средствами для работы с ними.

Всё вместе: особенности JavaScript

В этой главе приводятся основные особенности JavaScript, на уровне базовых конструкций, типов, синтаксиса.

Она будет особенно полезна, если ранее вы программировали на другом языке, ну или как повторение важных моментов раздела.

Всё очень компактно, со ссылками на развёрнутые описания.

Структура кода

Операторы разделяются точкой с запятой:

```
1 alert('Привет'); alert('Мир');
```

Как правило, перевод строки тоже подразумевает точку с запятой. Так тоже будет работать:

```
1 alert('Привет')
2 alert('Мир')
```

..Однако, иногда JavaScript не вставляет точку с запятой. Например:

```
1 var a = 2
2 +3
3
4 alert(a); // 5
```

Бывают случаи, когда это ведёт к ошибкам, которые достаточно трудно найти и исправить.

Поэтому рекомендуется точки с запятой ставить. Сейчас это, фактически, стандарт.

Поддерживаются однострочные комментарии `// ...` и многострочные `/* ... */`:

Подробнее: Структура кода [15].

Переменные и типы

- Объявляются директивой `var`. Могут хранить любое значение:

```
var x = 5;
x = "Петя";
```

- Есть 5 «примитивных» типов и объекты:

```
x = 1;              // число
x = "Тест";         // строка, кавычки могут быть одинарные или двойные
x = true;           // булево значение true/false
x = null;           // спец. значение (само себе тип)
x = undefined;      // спец. значение (само себе тип)
```

  Также есть специальные числовые значения `Infinity` (бесконечность) и `NaN`.

  Значение `NaN` обозначает ошибку и является результатом числовой операции, если она некорректна.

  Про объекты мы поговорим в главе Объекты как ассоциативные массивы [16], они в JavaScript сильно отличаются от большинства других языков.

- **Значение `null` не является «ссылкой на нулевой адрес/объект» или чем-то подобным. Это просто специальное значение.**

  Оно присваивается, если мы хотим указать, что значение переменной неизвестно.

  Например:

```
var age = null; // возраст неизвестен
```

- **Значение `undefined` означает «переменная не присвоена».**

  Например:

```
var x;
alert( x ); // undefined
```

  Можно присвоить его и явным образом: `x = undefined`, но так делать не рекомендуется.

- В имени переменной могут быть использованы любые буквы или цифры, но цифра не может быть первой. Символы доллар `$` и подчёркивание `_` допускаются наравне с буквами.

Подробнее: Переменные [17], Введение в типы данных [18].

Взаимодействие с посетителем

Простейшие функции для взаимодействия с посетителем в браузере:

prompt(вопрос[, по_умолчанию]) [19]

Задать `вопрос` и возвратить введённую строку, либо `null`, если посетитель нажал «Отмена».

confirm(вопрос) [20]
Задать `вопрос` и предложить кнопки «Ок», «Отмена». Возвращает, соответственно, `true/false`.

alert(сообщение) [21]
Вывести сообщение на экран.

Все эти функции являются *модальными*, т.е. не позволяют посетителю взаимодействовать со страницей до ответа.

Например:

```
var userName = prompt("Введите имя?", "Василий");
var smokes = confirm("Вы хотите чаю?");

alert( "Посетитель: " + userName);
alert( "Чай: " + smokes);
```

Подробнее: Взаимодействие с пользователем: alert, prompt, confirm [22].

Особенности операторов

- **Для сложения строк используется оператор `+`.**

  Если хоть один аргумент — строка, то другой тоже приводится к строке:

  ```
  alert( 1 + 2 );   // 3, число
  alert( '1' + 2 ); // '12', строка
  alert( 1 + '2' ); // '12', строка
  ```

- **Сравнение `===` проверяет точное равенство, включая одинаковый тип.** Это самый очевидный и надёжный способ сравнения.

  Остальные сравнения `== < <= > >=` осуществляют числовое приведение типа:

  ```
  alert( 0 == false ); // true
  alert( true > 0 );   // true
  ```

  Приведения не будет при сравнении двух строк (см. далее).

  Исключение: значения `null` и `undefined` ведут себя в сравнениях не как ноль.

  - Они равны друг другу `null == undefined` и не равны ничему ещё. В частности, не равны нулю.
  - В других сравнениях с приведением (не `===`) значение `null` преобразуется к нулю, а `undefined` — становится `NaN` («ошибка»).

  Такое поведение может привести к неочевидным результатам, поэтому лучше всего использовать для сравнения с ними `===`.

  Например, забавное следствие этих правил для `null`:

  ```
  alert( null > 0 );  // false, т.к. null преобразовано к 0
  alert( null == 0 ); // false, в стандарте явно указано, что null равен лишь undefined

  // но при этом (!)
  alert( null >= 0 ); // true, т.к. null преобразовано к 0
  ```

  С точки зрения здравого смысла такое невозможно. Значение `null` не равно нулю и не больше, но при этом `null >= 0` возвращает `true`!

- **Сравнение строк — лексикографическое, символы сравниваются по своим unicode-кодам.**

  Поэтому получается, что строчные буквы всегда больше, чем прописные:

```
alert('a' > 'Я'); // true
```

Подробнее: Основные операторы [23], Операторы сравнения и логические значения [24].

Логические операторы

В JavaScript есть логические операторы: И (обозначается `&&`), ИЛИ (обозначается `||`) и НЕ (обозначается `!`). Они интерпретируют любое значение как логическое.

Не стоит путать их с побитовыми операторами [25] И, ИЛИ, НЕ, которые тоже есть в JavaScript и работают с числами на уровне битов.

Как и в большинстве других языков, в логических операторах используется «короткий цикл» вычислений. Например, вычисление выражения `1 && 0 && 2` остановится после первого И `&&`, т.к. понятно что результат будет ложным (ноль интерпретируется как `false`).

Результатом логического оператора служит последнее значение в коротком цикле вычислений.

Можно сказать и по-другому: значения хоть и интерпретируются как логические, но то, которое в итоге определяет результат, возвращается без преобразования.

Например:

```
alert( 0 && 1 ); // 0
alert( 1 && 2 && 3 ); // 3
alert( null || 1 || 2 ); // 1
```

Подробнее: Логические операторы [26].

Циклы

- Поддерживаются три вида циклов:

```
// 1
while (условие) {
  ...
}

// 2
do {
  ...
} while(условие);

// 3
for (var i = 0; i < 10; i++) {
  ...
}
```

- Переменную можно объявлять прямо в цикле, но видна она будет и за его пределами.
- Поддерживаются директивы `break/continue` для выхода из цикла/перехода на следующую итерацию.

  Для выхода одновременно из нескольких уровней цикла можно задать метку.

Синтаксис: «`имя_метки:`», ставится она только перед циклами и блоками, например:

```
outer:
for(;;) {
    ...
  for(;;) {
    ...
    break outer;
  }
}
```

Переход на метку возможен только изнутри цикла, и только на внешний блок по отношению к данному циклу. В произвольное место программы перейти нельзя.

Подробнее: Директивы break и continue [27].

Конструкция `switch`

При сравнениях в конструкции `switch` используется оператор `===`.

Например:

```
var age = prompt('Ваш возраст', 18);

switch (age) {
  case 18:
    alert('Никогда не сработает'); // результат prompt - строка, а не число

  case "18":  // вот так - сработает!
    alert('Вам 18 лет!');
    break;

  default:
    alert('Любое значение, не совпавшее с case');
}
```

Подробнее: Конструкция switch [28].

Функции

Синтаксис функций в JavaScript:

```
// function имя(список параметров) { тело }
function sum(a, b) {
  var result = a + b;

  return result;
}
```

- `sum` — имя функции, ограничения на имя функции — те же, что и на имя переменной.
- Переменные, объявленные через `var` внутри функции, видны везде внутри этой функции, блоки `if`, `for` и т.п. на видимость не влияют.
- Параметры передаются «по значению», т.е. копируются в локальные переменные `a`, `b`, за исключением объектов, которые

передаются «по ссылке», их мы подробно обсудим в главе Объекты как ассоциативные массивы [29].

- Функция без `return` считается возвращающей `undefined`. Вызов `return` без значения также возвращает `undefined`:

```
function f() { }
alert( f() ); // undefined
```

Подробнее: Функции [30].

Методы и свойства

Все значения в JavaScript, за исключением `null` и `undefined`, содержат набор вспомогательных функций и значений, доступных «через точку».

Такие функции называют «методами», а значения — «свойствами».

Например:

```
alert( "Привет, мир!".length ); // 12
```

Еще у строк есть *метод* `toUpperCase()`, который возвращает строку в верхнем регистре:

```
var hello = "Привет, мир!";

alert( hello.toUpperCase() ); // "ПРИВЕТ, МИР!"
```

Подробнее: Методы и свойства [31].

Итого

В этой главе мы повторили основные особенности JavaScript, знание которых необходимо для обхода большинства «граблей», да и просто для написания хорошего кода.

Решения задач

Решение задачи: Работа с переменными

Каждая строчка решения соответствует одному шагу задачи:

```
var admin, name; // две переменных через запятую

name = "Василий";

admin = name;

alert(admin); // "Василий"
```

Решение задачи: Объявление переменных

Каждая строчка решения соответствует одному шагу задачи:

```
var ourPlanetName = "Земля"; // буквально "название нашей планеты"

var visitorName = "Петя"; // "имя посетителя"
```

Названия переменных можно бы сократить, например, до `planet` и `name`, но тогда станет менее понятно, о чем речь. Насколько это допустимо - зависит от скрипта, его размера и сложности.

Решение задачи: Инкремент и декремент, примеры

```
var a = 1, b = 1, c, d;

// префиксная форма сначала увеличивает a до 2, а потом возвращает
c = ++a; alert(c); // 2

// постфиксная форма увеличивает, но возвращает старое значение
d = b++; alert(d); // 1

// сначала увеличили a до 3, потом использовали в арифметике
c = (2+ ++a); alert(c); // 5

// увеличили b до 3, но в этом выражении оставили старое значение
d = (2+ b++); alert(d); // 4

// каждую переменную увеличили по 2 раза
alert(a); // 3
alert(b); // 3
```

Решение задачи: Результат присваивания

Ответ: `x = 5`.

Оператор присваивания возвращает значение, которое будет записано в переменную, например:

```
var a = 2;
alert( a *= 2 ); // 4
```

Отсюда `x = 1 + 4 = 5`.

Решение задачи: Побитовый оператор и значение

1. Операция `a^b` ставит бит результата в `1`, если на соответствующей битовой позиции в `a` или `b` (но не одновременно) стоит `1`.

   Так как в `0` везде стоят нули, то биты берутся в точности как во втором аргументе.

2. Первое побитовое НЕ `~` превращает `0` в `1`, а `1` в `0`. А второе НЕ превращает ещё раз, в итоге получается как было.

Решение задачи: Проверка, целое ли число

Один из вариантов такой функции:

```
function isInteger(num) {
  return (num ^ 0) === num;
}

alert( isInteger(1) ); // true
alert( isInteger(1.5) ); // false
alert( isInteger(-0.5) ); // false
```

Обратите внимание: `num^0` — в скобках! Это потому, что приоритет операции `^` очень низкий. Если не поставить скобку, то `===` сработает раньше. Получится `num ^ (0 === num)`, а это уже совсем другое дело.

Решение задачи: Симметричны ли операции ^, |, &?

Операция над числами, в конечном итоге, сводится к битам.

Посмотрим, можно ли поменять местами биты слева и справа.

Например, таблица истинности для `^`:

a	b	результат
0	0	0
0	1	1
1	0	1
1	1	0

Случаи `0^0` и `1^1` заведомо не изменятся при перемене мест, поэтому нас не интересуют. А вот `0^1` и `1^0` эквивалентны и равны `1`.

Аналогично можно увидеть, что и другие операторы симметричны.

Ответ: **да**.

Решение задачи: Почему результат разный?

Результат разный, потому что обычно число в JavaScript имеет 64-битный формат с плавающей точкой. При этом часть битов (52) отведены под цифры, часть (11) отведены под хранение номера позиции, на которой стоит десятичная точка, и один бит — знак числа.

Это означает, что максимальное целое, которое можно хранить, занимает 52 бита.

Побитовые операции преобразуют число в 32-битовое целое. При этом старшие из этих 52 битов будут отброшены. Если число изначально занимало больше чем 31 бита (еще один бит хранит не цифру, а знак) — оно изменится.

Вот ещё пример:

```
// в двоичном виде 10000000000000000000000000000000
alert( Math.pow(2, 30) ); // 1073741824
alert( Math.pow(2, 30) ^ 0 ); // 1073741824, всё ок, длины хватает

// в двоичном виде 100000000000000000000000000000000
alert( Math.pow(2, 32) ); // 4294967296
alert( Math.pow(2, 32) ^ 0 ); // 0, отброшены старшие биты!

// пограничный случай
// в двоичном виде 10000000000000000000000000000000
alert( Math.pow(2, 31) ); // 2147483648
alert( Math.pow(2, 31) ^ 0 ); // -2147483648, старший бит стал знаковым
```

Решение задачи: Простая страница

Решение: http://learn.javascript.ru/play/tutorial/intro/basic.html.

Решение задачи: if (строка с нулём)

Да, выведется, т.к. внутри `if` стоит строка `"0"`.

Любая строка, кроме пустой (а здесь она не пустая), в логическом контексте является `true`.

Можно запустить и проверить:

```
if ("0") {
  alert('Привет');
}
```

Решение задачи: Проверка стандарта

Решение: http://learn.javascript.ru/play/tutorial/intro/ifelse_task2.html.

Решение задачи: Получить знак числа

http://learn.javascript.ru/play/tutorial/intro/if_sign.html

Решение задачи: Проверка логина

Решение: http://learn.javascript.ru/play/tutorial/intro/ifelse_task.html.

Обратите внимание на дополнительные вертикальные отступы внутри `if`. Они полезны для лучшей читаемости кода.

Решение задачи: Перепишите 'if' в '?'

```
result = (a + b < 4) ? 'Мало' : 'Много';
```

Решение задачи: Перепишите 'if..else' в '?'

```
var message = (login == 'Вася') ? 'Привет' :
  (login == 'Директор') ? 'Здравствуйте' :
  (login == '') ? 'Нет логина' :
  '';
```

Решение задачи: Что выведет alert (ИЛИ)?

Ответ: сначала `1`, затем `2`.

```
alert( alert(1) || 2 || alert(3) );
```

Вызов `alert` не возвращает значения, или, иначе говоря, возвращает `undefined`.

1. Первый оператор ИЛИ `||` выполнит первый `alert(1)`, получит `undefined` и пойдёт дальше, ко второму операнду.
2. Так как второй операнд `2` является истинным, то вычисления завершатся, результатом `undefined || 2` будет `2`, которое будет выведено внешним `alert( .... )`.

Второй оператор `||` не будет выполнен, выполнение до `alert(3)` не дойдёт, поэтому `3` выведено не будет.

Решение задачи: Что выведет alert (И)?

Ответ: `1`, `undefined`.

```
alert( alert(1) && alert(2) );
```

Вызов `alert` не возвращает значения, или, иначе говоря, возвращает `undefined`.

Поэтому до правого `alert` дело не дойдёт, вычисления закончатся на левом.

Решение задачи: Проверка if внутри диапазона

```
if (age >= 14 && age <= 90)
```

Решение задачи: Проверка if вне диапазона

Первый вариант:

```
if ( !(age >= 14 && age <= 90) )
```

Второй вариант:

```
if (age < 14 || age > 90)
```

Решение задачи: Вопрос про "if"

Ответ: первое и третье выполнятся. Детали:

```
// Да, выполнится, т.к. -1 в логическом контексте true
// -1 || 0 = -1
if (-1 || 0) alert('первое');

// Не выполнится, т.к. -1 интерпретируется как true, а 0 как false
// -1 && 0 = 0
if (-1 && 0) alert('второе');

// Да, выполнится
// оператор && имеет больший приоритет, чем ||
// так что -1 && 1 выполнится раньше, будет null || 1 = 1
// получится null || -1 && 1 -> null || 1 -> 1
if (null || -1 && 1) alert('третье');
```

Решение задачи: Последнее значение цикла

Ответ: `1`.

```
var i = 3;

while(i) {
  alert(i--);
}
```

Каждое выполнение цикла уменьшает `i`. Проверка `while(i)` даст сигнал «стоп» при `i = 0`.

Соответственно, шаги цикла:

```
var i = 3
alert(i--); // выведет 3, затем уменьшит i до 2

alert(i--) // выведет 2, затем уменьшит i до 1

alert(i--) // выведет 1, затем уменьшит i до 0

// все, проверка while(i) не даст выполняться циклу дальше
```

Решение задачи: Какие значения i выведет цикл while?

1. **От 1 до 4**

```
var i = 0;
while (++i < 5) alert(i);
```

Первое значение: `i=1`, так как операция `++i` сначала увеличит `i`, а потом уже произойдёт сравнение и выполнение `alert`.

Далее `2,3,4..` Значения выводятся одно за другим. Для каждого значения сначала происходит увеличение, а потом — сравнение, так как `++` стоит перед переменной.

При `i=4` произойдет увеличение `i` до `5`, а потом сравнение `while(5 < 5)` — это неверно. Поэтому на этом цикл остановится, и значение `5` выведено не будет.

2. **От 1 до 5**

```
var i = 0;
while (i++ < 5) alert(i);
```

Первое значение: `i=1`. Остановимся на нём подробнее. Оператор `i++` увеличивает `i`, возвращая старое значение, так что в сравнении `i++ < 5` будет участвовать старое `i=0`.

Но последующий вызов `alert` уже не относится к этому выражению, так что получит новый `i=1`.

Далее `2,3,4..` Для каждого значения сначала происходит сравнение, а потом — увеличение, и затем срабатывание `alert`.

Окончание цикла: при `i=4` произойдет сравнение `while(4 < 5)` — верно, после этого сработает `i++`, увеличив `i` до `5`, так что значение `5` будет выведено. Оно станет последним.

Решение задачи: Какие значения i выведет цикл for?

Ответ: от 0 до 4 в обоих случаях.

```
for(var i=0; i<5; ++i) alert(i);

for(var i=0; i<5; i++) alert(i);
```

Такой результат обусловлен алгоритмом работы `for`:

1. Выполнить присвоение `i=0`
2. Проверить `i<5`
3. Если верно - выполнить тело цикла `alert(i)`, затем выполнить `i++`

Увеличение `i++` выполняется отдельно от проверки условия (2), значение `i` при этом не используется, поэтому нет никакой разницы между `i++` и `++i`.

Решение задачи: Замените for на while

http://learn.javascript.ru/play/tutorial/intro/loop.html.

Решение задачи: Повторять цикл, пока ввод неверен

Решение:

```
var num;

do {
  num = prompt("Введите число больше 100?", 0);
} while(num <= 100 && num != null);
```

В действии: http://learn.javascript.ru/play/tutorial/intro/endless_loop.html

Цикл `do..while` повторяется, пока верны две проверки:

1. Проверка `num <= 100` — то есть, введённое число всё еще меньше `100`.
2. Проверка `num != null` — значение `null` означает, что посетитель нажал «Отмена», в этом случае цикл тоже нужно прекратить.

Кстати, сравнение `num <= 100` при вводе `null` даст `true`, так что вторая проверка необходима.

Решение задачи: Вывести простые числа

Схема решения

```
Для всех i от 1 до 10 {
 проверить, делится ли число i на какое-либо из чисел до него
   если делится, то это i не подходит, берем следующее
   если не делится, то i - простое число
}
```

Решение

Решение с использованием метки:

```
nextPrime:
for(var i=2; i<10; i++) {

  for(var j=2; j<i; j++) {
    if ( i % j == 0) continue nextPrime;
  }

  alert(i);  // простое
}
```

Конечно же, его можно оптимизировать с точки зрения производительности. Например, проверять все `j` не от 2 до `i`, а от 2 до квадратного корня из `i`. А для очень больших чисел — существуют более эффективные специализированные алгоритмы проверки простоты числа, например квадратичное решето [32] и решето числового поля [33] .

Решение задачи: Напишите "if", аналогичный "switch"

Если совсем точно следовать условию, то сравнение должно быть строгим `'==='`.

В реальном случае, скорее всего, подойдёт обычное сравнение `'=='`.

```
if(browser == 'IE') {
  alert('О, да у вас IE!');
} else if (browser == 'Chrome' || browser == 'Firefox'
  || browser == 'Safari' || browser == 'Opera') {
  alert('Да, и эти браузеры мы поддерживаем');
} else {
  alert('Мы надеемся, что и в вашем браузере все ок!');
}
```

Как видно, эта запись существенно хуже читается, чем конструкция `switch`.

Решение задачи: Переписать if'ы в switch

Первые две проверки — обычный `case`, третья разделена на два `case`:

```
01 var a = +prompt('a?', '');
02
03 switch(a) {
04   case 0:
05     alert(0);
06     break;
07
08   case 1:
09     alert(1);
10     break;
11
12   case 2:
13   case 3:
14     alert('2,3');
15     break;
16 }
```

Обратите внимание: `break` внизу не обязателен, но ставится по «правилам хорошего тона».

Если он не стоит, то при дописывании в конец нового `case`, к примеру `case 4`, мы, скорее всего, забудем его поставить. В результате выполнение `case 2`/`case 3` продолжится на `case 4` и будет ошибка.

Решение задачи: Обязателен ли "else"?

Оба варианта функции работают одинаково, отличий нет.

Решение задачи: Перепишите функцию, используя оператор '?' или '||'

Используя оператор `'?'`:

```
function checkAge(age) {
  return (age > 18) ? true : confirm('Родители разрешили?');
}
```

Используя оператор `||` (самый короткий вариант):

```
function checkAge(age) {
  return (age > 18) || confirm('Родители разрешили?');
}
```

Решение задачи: Функция min

Код с `if`:

```
function min(a, b) {
  if (a < b) {
    return a;
  } else {
    return b;
  }
}
```

Код с оператором `'?'`:

```
function min(a, b) {
  return a < b ? a : b;
}
```

P.S. Случай равенства `a == b` отдельно не рассматривается, так как при этом неважно, что возвращать.

Решение задачи: Функция pow(x,n)

Решение: http://learn.javascript.ru/play/tutorial/intro/pow.html

Решение задачи: Вычислить сумму чисел до данного

Решение **с использованием цикла**:

```
function sumTo(n) {
  var sum = 0;
  for(var i=1; i<=n; i++) {
    sum += i;
  }
  return sum;
}

alert( sumTo(100) );
```

Решение через **рекурсию**:

```
function sumTo(n) {
  if (n == 1) return 1;
  return n + sumTo(n-1);
}

alert( sumTo(100) );
```

Решение **по формуле**: `sumTo(n) = n*(n+1)/2`:

```
function sumTo(n) {
  return n*(n+1)/2;
}

alert( sumTo(100) );
```

P.S. Надо ли говорить, что решение по формуле работает быстрее всех? Оно использует всего три операции для любого `n`, а цикл и рекурсия требуют как минимум `n` операций сложения.

Вариант с циклом — второй по скорости. Он быстрее рекурсии, так как операции по сути те же, но нет накладных расходов на вложенный вызов функции.

Рекурсия в данном случае работает медленнее всех.

P.P.S. Существует ограничение глубины вложенных вызовов, поэтому рекурсивный вызов `sumTo(100000)` выдаст ошибку.

Решение задачи: Вычислить факториал

По свойствам факториала, как описано в условии, `n!` можно записать как `n * (n-1)!`.

То есть, результат функции для `n` можно получить как `n`, умноженное на результат функции для `n-1`, и так далее до `1!`:

```
function factorial(n) {
  return (n!=1) ? n*factorial(n-1) : 1;
}

alert( factorial(5) ); // 120
```

Базисом рекурсии является значение `1`. А можно было бы сделать базисом и `0`. Тогда код станет чуть короче:

```
function factorial(n) {
  return n ? n*factorial(n-1) : 1;
}

alert( factorial(5) ); // 120
```

В этом случае вызов `factorial(1)` сведется к `1*factorial(0)`, будет дополнительный шаг рекурсии.

Решение задачи: Числа Фибоначчи

Вычисление рекурсией (медленное)

Решение по формуле, используя рекурсию:

```
function fib(n) {
   return n <= 1 ? n : fib(n-1) + fib(n-2);
}

alert( fib(3) ); // 2
alert( fib(7) ); // 13
// fib(77); // не запускаем, подвесит браузер
```

При больших значениях `n` оно будет работать очень медленно. Например, `fib(77)` уже будет вычисляться очень долго.

Это потому, что функция порождает обширное дерево вложенных вызовов. При этом ряд значений вычисляются много раз. Например, `fib(n-2)` будет вычислено как для `fib(n)`, так и для `fib(n-1)`.

Можно это оптимизировать, запоминая уже вычисленные значения.. А можно просто отказаться от рекурсии, а вместо этого пойти по формуле слева-направо в цикле, вычисляя 1е, 2е, 3е и так далее числа до нужного.

Попробуйте. Не получится — откройте решение ниже.

Последовательное вычисление (идея)

Будем идти по формуле слева-направо:

```
var a = 1, b = 1; // начальные значения
var c = a + b; // 2

/* переменные на начальном шаге:
a  b  c
1, 1, 2
*/
```

Теперь следующий шаг, присвоим `a` и `b` текущие 2 числа и получим новое следующее в `c`:

```
a = b, b = c;
c = a + b;

/* стало так (еще число):
   a  b  c
1, 1, 2, 3
*/
```

Следующий шаг даст нам еще одно число последовательности:

```
a = b, b = c;
c = a + b;

/* стало так (еще число):
      a  b  c
1, 1, 2, 3, 5
*/
```

Повторять в цикле до тех пор, пока не получим нужное значение. Это гораздо быстрее, чем рекурсия, хотя бы потому что ни одно из чисел не вычисляется дважды.

P.S. Этот подход к вычислению называется динамическое программирование снизу-вверх [34] .

Последовательное вычисление (код)

```
function fib(n) {
  var a = 1, b = 1;
  for (var i = 3; i <= n; i++) {
    var c = a + b;
    a = b;
    b = c;
  }
  return b;
}

alert( fib(3) ); // 2
alert( fib(7) ); // 13
alert( fib(77)); // 5527939700884757
```

Цикл здесь начинается с `i=3`, так как первое и второе числа Фибоначчи заранее записаны в переменные `a=1`, `b=1`.

Ссылки

1. Функциональные http://ru.wikipedia.org/wiki/Язык_функционального_программирования
2. Таблица приоритетов https://developer.mozilla.org/en/JavaScript/Reference/operators/operator_precedence
3. Побитовые операторы http://learn.javascript.ru/bitwise-operators
4. Преобразование объектов: toString и valueOf http://learn.javascript.ru/object-conversion
5. Двоичная система счисления http://ru.wikipedia.org/wiki/%C4%E2%EE%E8%F7%ED%E0%FF_%F1%E8%F1%F2%E5%EC%E0_%F1%F7%E8%F1%EB%E5%ED%E8%FF
6. Внутренний формат http://learn.javascript.ru/bitwise-operators#signed-format
7. Говорить об объектах http://learn.javascript.ru/object#for..in
8. Замыкания, функции изнутри http://learn.javascript.ru/closures
9. Псевдо-массив arguments http://learn.javascript.ru/arguments-pseudoarray
10. JQuery http://jquery.com
11. Prototype http://prototypejs.com
12. Числа http://learn.javascript.ru/number
13. Строками http://learn.javascript.ru/string
14. Числами http://learn.javascript.ru/number
15. Структура кода http://learn.javascript.ru/structure
16. Объекты как ассоциативные массивы http://learn.javascript.ru/object
17. Переменные http://learn.javascript.ru/variables
18. Введение в типы данных http://learn.javascript.ru/types-intro
19. Prompt(вопрос[, по_умолчанию]) https://developer.mozilla.org/en/DOM/window.prompt
20. Confirm(вопрос) https://developer.mozilla.org/en/DOM/window.confirm
21. Alert(сообщение) https://developer.mozilla.org/en/DOM/window.alert
22. Взаимодействие с пользователем: alert, prompt, confirm http://learn.javascript.ru/uibasic
23. Основные операторы http://learn.javascript.ru/operators
24. Операторы сравнения и логические значения http://learn.javascript.ru/comparison
25. Побитовыми операторами http://learn.javascript.ru/bitwise-operators
26. Логические операторы http://learn.javascript.ru/logical-ops
27. Директивы break и continue http://learn.javascript.ru/break-continue
28. Конструкция switch http://learn.javascript.ru/switch
29. Объекты как ассоциативные массивы http://learn.javascript.ru/object
30. Функции http://learn.javascript.ru/function-basics
31. Методы и свойства http://learn.javascript.ru/properties-and-methods
32. Квадратичное решето http://ru.wikipedia.org/wiki/Метод_квадратичного_решета
33. Решето числового поля http://ru.wikipedia.org/wiki/Общий_метод_решета_числового_поля
34. Динамическое программирование снизу-вверх http://ru.wikipedia.org/wiki/Динамическое_программирование