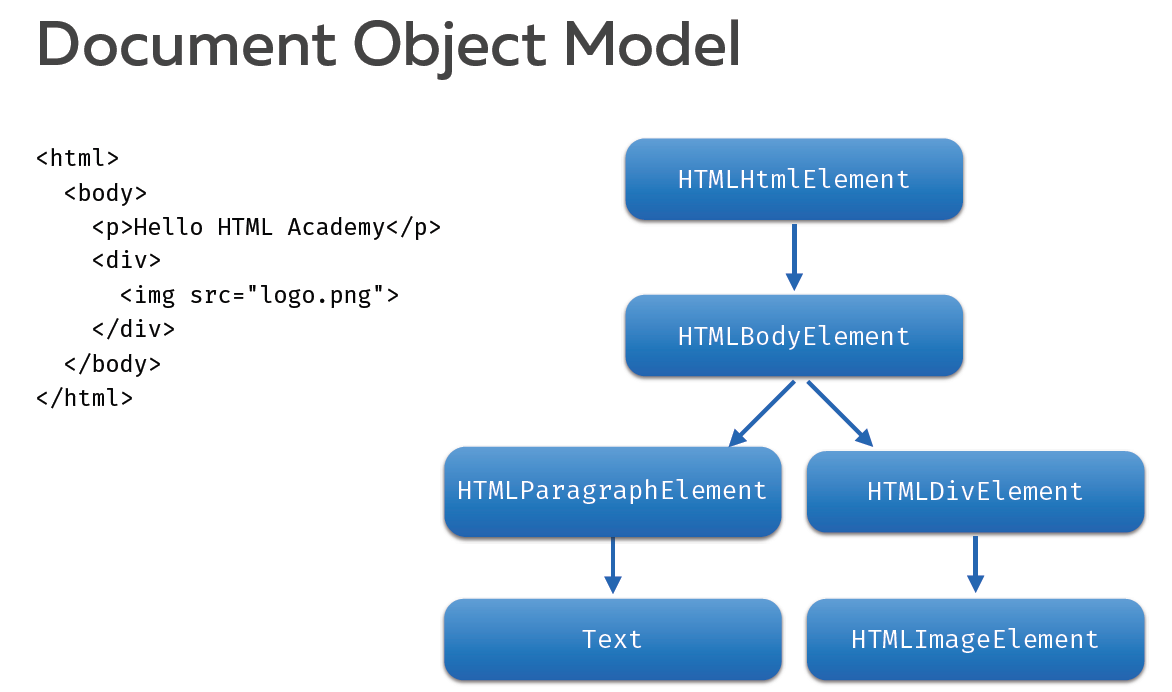
JS

# Из курса HTML Academy

JavaScript-DOM(Объектная модель документа или представление верстки ввиде дерева)- HTML и CSS:



## document

Глобальный объект; является корневым элементом DOM-дерева страницы.

(!) Размещать скрипт в самый конец HTML странички

event.preventDefault(); // preventDefault() отменяет действие по умолчанию для event

## querySelector

Метод для поиска элементов на странице. Возвращает элемент из DOM-дерева,

соответствующий CSS-селектору.

Пример1:

document1.querySelector2(".user-block")3;

1. Где будет искать (по всему DOM-дереву). Можно указать др.верхнюю точку
2. Метод для поиска элемента
3. Передаём методу CSS-селектор, по которому будем искать.

Находит 1-й найденный элемент в дереве.

Пример найти «ссылку вход»:

<html>

<body>

…

<a class="login" href="#">Вход</a>

…

<script>

**var link = document.querySelector(".login");**

link.addEventListener("click", function(event) {

event.preventDefault();

});

</script>

</body>

</html>

Пример2(ищем модальное окно):

…

<div class="modal-content">

…

</div>

<script>

var popup = document.querySelector(".modal-content");

</script>

.querySelectorAllвозвращает все элементы из DOM-дерева, соответствующие CSS-селектору.

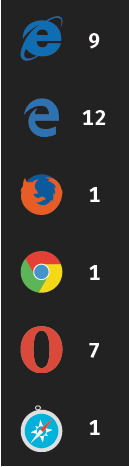
Пример:

document.querySelectorAll("nav li");

## addEventListener

Метод для отлова диких событий. Отлавливает событие элемента и выполняет переданную функцию.

Пример:

link1.addEventListener2("click"3, function(){}4);

1. Элемент, у которого мы будем ловить событие.

2. Метод для отлова событий.

3. Определяем, какое событие мы ловим («клик»).

4. Функция, которая будет выполняться каждый раз,

когда мы ловим событие «клик».

Примеры событий:

Mouse events

Click , dblclick, keydown-нажали, keypress-нажали клавишу и отпустили, keyup-отпустили, mouseover , mouseout

Focus events

blur, focus, focusin, focusout, change

Form events

reset, submit

Другие: <https://developer.mozilla.org/ru/docs/web/events>

## classList

Набор методов для управления классами элемента.

Пример:

element1.classList2.add3 ("modal-content-show"4);

1. Элемент, которому добавляем новый класс.

2. Свойство, содержащее в себе методы для работы

с классами.

3. Метод добавления класса.

* element.classList.add(); Добавляет класс.
* element.classList.remove(); Удаляет класс.
* element.classList.toggle(); Переключает класс (если есть этот класс – удаляет,если нет - добавляет).
* element.classList.contains(); Сообщает, есть ли класс у элемента.

4. Название класса, которое нужно добавить элементу.

Пример:

<style>

.modal-content {

display: none;

**…**

}

.modal-content-show { /\* Создаем новый класс \*/

display: block;

}

</style>

<script>

var popup = document.querySelector(".modal-content");

popup.classList.add("(\*)modal-content-show"); (\*) здесь точка не нужна

</script>

Сообщение в браузере

console.log("Сообщение в консоль браузера");

Скрипт вместе:

<script>

var link = document.querySelector(".login"); найти ссылку .login и сохранить в link

var popup = document.querySelector(".modal-content"); найти .modal-content и сохранить в popup

link.addEventListener("click", function(event) { на ссылке .login отслеживать все события «клика»

event.preventDefault(); при клике по ссылке отменяется значение по умолчанию(href)

popup.classList.add("modal-content-show"); добавить modal-content класс modal-content-show

});

</script>

Закрыть модальное окно:

…

<div class="modal-content">

<button class="modal-content-close" type="button">Закрыть</button>

…

</div>

<script>

…

var close = document.querySelector(".modal-content-close"); // кнопка закрыть

close.addEventListener("click", function(event) {

event.preventDefault(); // отменить действие по умолч.

popup.classList.remove("modal-content-show"); // у popup удалить класс modal-content-show

});

</script>

Промежуточный итог:

<script>

var link = document.querySelector(".login");

var popup = document.querySelector(".modal-content");

var close = document.querySelector(".modal-content-close");

link.addEventListener("click", function(event) {

event.preventDefault();

popup.classList.add("modal-content-show");

});

close.addEventListener("click", function(event) {

event.preventDefault();

popup.classList.remove("modal-content-show");

});

</script>

Фокус на поле ввода

var login = popup.querySelector(“[name=login]”); // внутри popup (div class=”modal-content) c атрибутом [name=login]

login.focus(); // вызвать функцию курсор попадает в поле Логин

Вставляем:

<script>

var link = document.querySelector(".login");

var popup = document.querySelector(".modal-content");

var close = document.querySelector(".modal-content-close");

var login = popup.querySelector(“[name=login]”);

link.addEventListener("click", function(event) {

event.preventDefault();

popup.classList.add("modal-content-show");

login.focus();

});

close.addEventListener("click", function(event) {

event.preventDefault();

popup.classList.remove("modal-content-show");

});

</script>

Проверим форму (на заполненность форм)

<script>

...

var popup = document.querySelector(“.modal-content”);

var form = popup.querySelector(“form”);

form.addEventListener(“submit” , function(event) {

event.preventDefault();

console.log(“Отправляем форму”); // проверка

});

</script> Все работает теперь настраиваем:

>

## Условие

If1 (!3login.value)2 { … } условие

1.Оператор условия: если условие в скобках верное, то запускается код в фигурных скобках.

2. Проверяем существование значение поля login.

3. Знак ! переворачивает условие наоборот: значение остуствует у поля login.

&& Эмперсент логическое «и»

|| Логическое “или”

else действие если if не выполняется (иначе)

if (!login.value || !password.value) { … }

1. Оператор условия: если условие в скобках верное,

то запускается код в фигурных скобках.

2. Значение отсутствует у поля login **или** у поля password (или у обоих сразу).

Промежуточный итог #2

<script>

var link = document.querySelector(".login");

var popup = document.querySelector(".modal-content");

var close = popup.querySelector(".modal-content-close");

var form = popup.querySelector("form");

var login = popup.querySelector("[name=login]");

var password = popup.querySelector("[name=password]");

link.addEventListener("click", function(event) {

event.preventDefault();

popup.classList.add("modal-content-show");

login.focus();

});

close.addEventListener("click", function(event) {

event.preventDefault();

popup.classList.remove("modal-content-show");

});

form.addEventListener("submit", function(event) {

if (!login.value || !password.value) {

event.preventDefault();

console.log("Нужно ввести логин и пароль");

}

});

</script>

## localStorage

Хранение данных в браузере (ключ=значение).

localStorage1.setItem2("name"3, "keks"4);

1.Глобальный объект для работы с хранилищем.

2. Метод создания новой записи в хранилище.

* getItem() получает значение ключа их хранилища.
* setItem() создает новую запись в хранилище.
* removeitem() удаляет запись из хранилища.
* clear() полностью очищает хранилище.

3. Ключ записи.

4. Значение записи.

Запомним логин

<script>

…

form.addEventListener("submit", function(event) {

if (!login.value || !password.value) {

event.preventDefault();

console.log("Нужно ввести логин и пароль");

} else {

**localStorage.setItem("login", login.value);**

});

</script>

<script>

…

var popup = document.querySelector(".modal-content");

var form = popup.querySelector("form");

var login = popup.querySelector("[name=login]");

var password = popup.querySelector("[name=password]");

**var storage = localStorage.getItem("login");** в **localStorage cохраем в “ключе” login текущее значение, которое находится в поле логин**

link.addEventListener("click", function(event) {

event.preventDefault();

popup.classList.add("modal-content-show");

if (storage) { если значение внутри переменной storage существует

login.value = storage; свойству value переменной login устанавливаем значение переменной,которая находится в storage

password.focus(); если в хранилище есть данные фокусировка на поле пароля.

} else { иначе

login.focus(); мы ставим фокус на поле логин

}

});

…

</script>

## window

Глобальный объект; занимается окном, внутри которого находится DOM-дерево.

<script>

…

**window**.addEventListener(**"keydown"**, function(event) { // нажатие любой клавиши

**if (event.keyCode === 27) { //** keyCode индификатор клавиш, 27 индиф. номер клавиши, === строгое сравнение

if (popup.classList.contains(“modal-content-show”)) // если есть у popup класс modal-content-show

popup.classList.remove(“modal-content-show”); // удаляем его

**}**

**}**

…

});

</script>

Клавиши: developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent#Virtual\_key\_codes

keycod.es

Промежуточный итог #3

<script>

var link = document.querySelector(".login");

var popup = document.querySelector(".modal-content");

var close = popup.querySelector(".modal-content-close");

var form = popup.querySelector("form");

var login = popup.querySelector("[name=login]");

var password = popup.querySelector("[name=password]");

var storage = localStorage.getItem("login");

link.addEventListener("click", function(event) {

event.preventDefault();

popup.classList.add("modal-content-show");

if (storage) {

login.value = storage;

password.focus();

} else {

login.focus();

}

});

close.addEventListener("click", function(event) {

event.preventDefault();

popup.classList.remove("modal-content-show");

});

form.addEventListener("submit", function(event) {

if (!login.value || !password.value) {

event.preventDefault();

console.log("Нужно ввести логин и пароль");

} else {

localStorage.setItem("login", login.value);

}

});

window.addEventListener("keydown", function(event) {

if (event.keyCode === 27) {

if (popup.classList.contains("modal-content-show")) {

popup.classList.remove("modal-content-show");

}

}

});

</script>

## Анимация

Плавно спустим форму

<style>

…

**@keyframes bounce** {

0% { transform: translateY(-2000px); }

70% { transform: translateY(30px); }

90% { transform: translateY(-10px); }

100% { transform: translateY(0); }

}

.modal-content-show {

display: block;

**animation: bounce 0.6s;**

}

</style>

Потрясём форму

<style>

…

**@keyframes shake** {

0%, 100% { transform: translateX(0); }

10%, 30%, 50%, 70%, 90% { transform: translateX(-10px); }

20%, 40%, 60%, 80% { transform: translateX(10px); }

}

**.modal-error {**

**animation: shake 0.6s;**

**}**

</style>

form.addEventListener(“submit” , function(event) {

if (!login.value || !password.value) {

event.preventDefault();

popup.classList.add(“modal-error”); Не забыть удалить этот класс при нажатии Esc и т.п.

} else {

localStorage.setItem(“login”, login.value);

}

});

Готовый вариант:

<script>

var link = document.querySelector(".login");

var popup = document.querySelector(".modal-content");

var close = popup.querySelector(".modal-content-close");

var form = popup.querySelector("form");

var login = popup.querySelector("[name=login]");

var password = popup.querySelector("[name=password]");

var storage = localStorage.getItem("login");

link.addEventListener("click", function(event) {

event.preventDefault();

popup.classList.add("modal-content-show");

if (storage) {

login.value = storage;

password.focus();

} else {

login.focus();

}

});

close.addEventListener("click", function(event) {

event.preventDefault();

popup.classList.remove("modal-content-show");

popup.classList.remove("modal-error");

});

form.addEventListener("submit", function(event) {

if (!login.value || !password.value) {

event.preventDefault();

popup.classList.add("modal-error");

} else {

localStorage.setItem("login", login.value);

}

});

window.addEventListener("keydown", function(event) {

if (event.keyCode === 27) {

if (popup.classList.contains("modal-content-show")) {

popup.classList.remove("modal-content-show");

popup.classList.remove("modal-error");

}

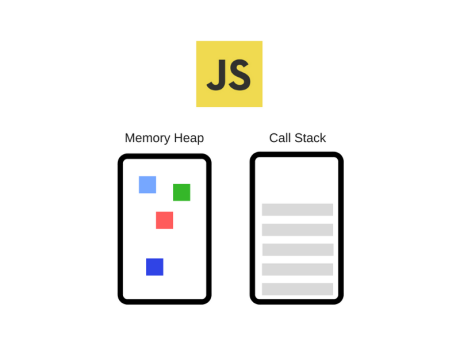
}

});

</script>

JavaScript —язык программирования сценариев *(script)* со слабой типизацией и динамическим приведением типов.

// однопоточный язык, использующий очередь функций обратного вызова. <Движок v8 от Google, он так же используется, например, в браузере Chrome и в Node.js.>



**Динамическая и статическая типизация**

Код конвертируется в другую форму, которую компьютер может запустить. Этот процесс называется компиляцией, а период времени, за который этот процесс происходит — **стадией компиляции** (compile time).

После того, как компиляция закончена, запускается программа и период, пока она запущена, называется **стадией исполнения** (run time).

**Статически типизированные** языки проверяют типы и ищут ошибки типизации на стадии компиляции.

**Динамически типизированные** языки проверяют типы и ищут ошибки типизации на стадии исполнения.

Иными словами: статическое типизирование значит проверку типов перед запуском программы; динамическое — проверку типов, пока программа запущена.

**Слабая и сильная типизация**

JavaScript часто конвертирует типы автоматически:

4 + '7'; // '47'

4 \* '7'; // 28

2 + true; // 3

false - 3; // -3

JavaScript — это язык со слабой типизацией. У него есть представление о типах, но он расслаблено к ним относится и может оперировать значениями, можно сказать, произвольно. Чем сильнее система типизации, тем строже правила.

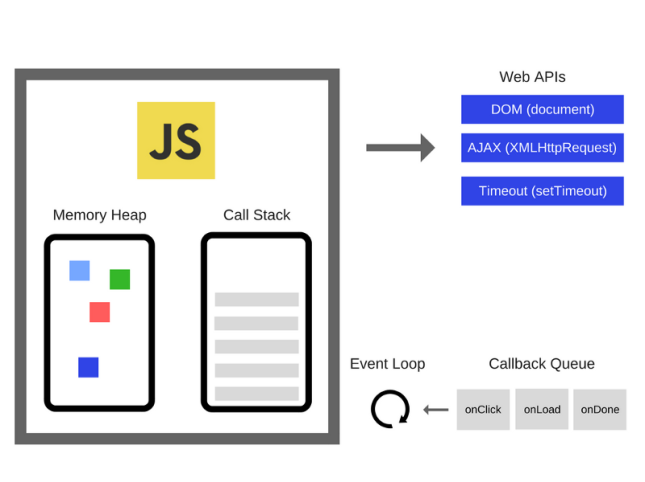
*Упрощённое представление движка V8*  
На нашей схеме движок представлен состоящим из двух основных компонентов:

* Куча (Memory Heap) — то место, где происходит выделение памяти.
* Стек вызовов (Call Stack) — то место, куда в процессе выполнения кода попадают так называемые стековые кадры.

**Механизмы времени выполнения**

Если говорить о применении JavaScript в браузере, то здесь существуют API < (Application Programming Interface) — это интерфейс программирования, интерфейс создания приложений. Если говорить более понятным языком, то API — это готовый код для упрощения жизни программисту. API создавался для того, чтобы программист реально мог облегчить задачу написания того или иного приложения благодаря использованию готового кода (например, функций). Всем известный jQuery, написанный на JavaScript является тоже своего рода API. Если рассматривать конкретно данный пример, то jQuery позволяет намного облегчить написание кода. То что обычными средствами JavaScript можно было сделать за 30 строк, через jQuery пишется через 5-6. Если рассматривать API в общем, то можно найти очень много сервисов, представляющих решения для разработки. Самый известный на сегодняшний день — это сервис code.google.com, предоставляющий около полусотни разнообразных API! Это и интерфейс для создания Android-приложений, и различные API для работы с AJAX, и различные API приложений, которые можно легко подстроить под свой лад.>, например, что-то вроде функции setTimeout, которые использует практически каждый JS-разработчик. Однако, эти API предоставляет не движок.

Откуда же они берутся? Оказывается, что реальность выглядит немного сложнее, чем может показаться на первый взгляд.



*Движок, цикл событий, очередь функций обратного вызова и API, предоставляемые браузером*

Итак, помимо движка у нас есть ещё очень много всего. Скажем — так называемые Web API, которые предоставляет нам браузер — средства для работы с DOM, инструменты для выполнения AJAX-запросов, нечто вроде функции setTimeout, и многое другое.

**Стек вызовов**

JavaScript — однопоточный язык программирования. Это означает, что у него один стек вызовов. Таким образом, в некий момент времени он может выполнять лишь какую-то одну задачу.  
  
Стек вызовов — это структура данных, которая, говоря упрощённо, записывает сведения о месте в программе, где мы находимся. Если мы переходим в функцию, мы помещаем запись о ней в верхнюю часть стека. Когда мы из функции возвращаемся, мы вытаскиваем из стека самый верхний элемент и оказываемся там, откуда вызывали эту функцию. Это — всё, что умеет стек.  
  
Рассмотрим пример. Взгляните на следующий код:

function multiply(x, y) {

   return x \* y;

}

function printSquare(x) {

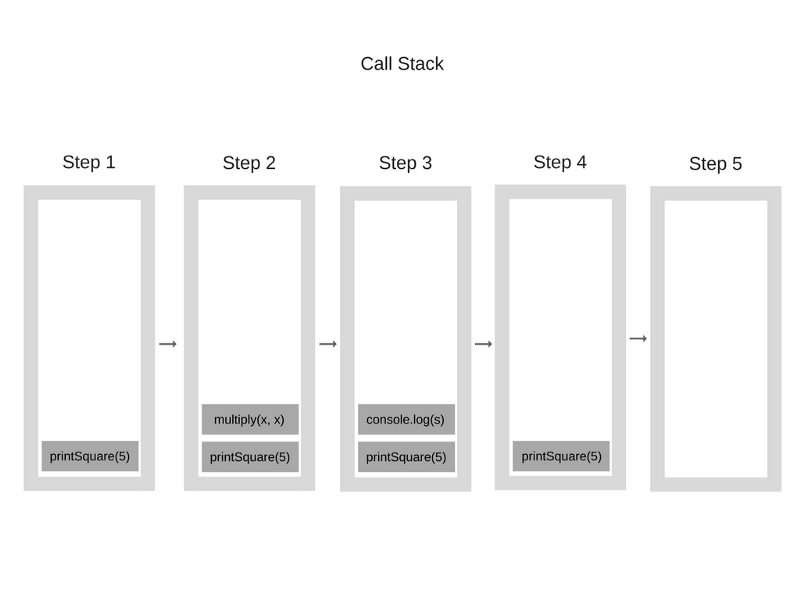
   var s = multiply(x, x);

   console.log(s);

}

printSquare(5);

Когда движок только начинает выполнять этот код, стек вызовов пуст. После этого происходит следующее:



*Стек вызовов в ходе выполнения программы*

Каждая запись в стеке вызовов называется **стековым кадром**.  
  
На механизме анализа стековых кадров основана информация о стеке вызовов, трассировка стека, выдаваемая при возникновении исключения. Трассировка стека представляет собой состояние стека в момент исключения. Взгляните на следующий код:

function foo() {

   throw new Error('SessionStack will help you resolve crashes :)');

}

function bar() {

   foo();

}

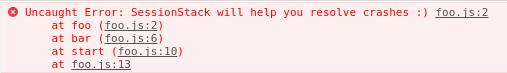
function start() {

   bar();

}

start();

Если выполнить это в Chrome (предполагается, что код находится в файле foo.js), мы увидим следующие сведения о стеке:



*Трассировка стека после возникновения ошибки*

Если будет достигнут максимальный размер стека, возникнет так называемое переполнение стека. Произойти такое может довольно просто, например, при необдуманном использовании рекурсии. Взгляните на этот фрагмент кода:

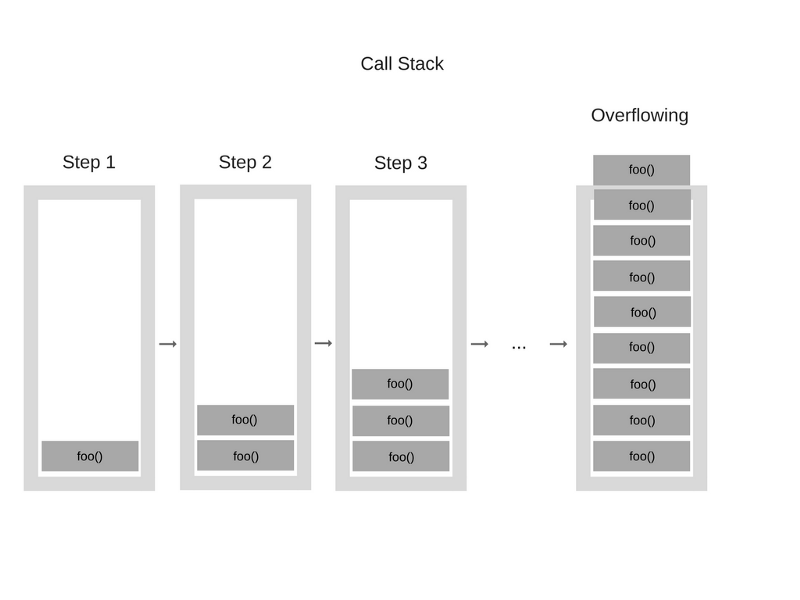
function foo() *{*

*foo();*

*}*

foo();

Когда движок приступает к выполнению этого кода, всё начинается с вызова функции foo. Это — рекурсивная функция, которая не содержит условия прекращения рекурсии. Она бесконтрольно вызывает сама себя. В результате на каждом шаге выполнения в стек вызовов снова и снова добавляется информация об одной и той же функции. Выглядит это примерно так:



*Переполнение стека*  
  
В определённый момент, однако, объём данных о вызовах функции превысит размер стека вызовов и браузер решит вмешаться, выдав ошибку:

https://habrastorage.org/getpro/habr/post_images/7b3/3b2/5c8/7b33b25c8e41536d214bf5a8cc37a137.png

Модель выполнения кода в однопоточном режиме облегчает жизнь разработчика. Ему не нужно принимать во внимание сложные схемы взаимодействия программных механизмов, вроде возможности взаимной блокировки потоков, которые возникают в многопоточных окружениях.  
  
Однако, и у исполнения кода в однопоточном режиме тоже есть определённые ограничения. Учитывая то, что у JavaScript имеется один стек вызовов, поговорим о том, что происходит, когда программа «тормозит».

**Параллельное выполнение кода и цикл событий**

Что происходит, когда в стеке вызовов имеется функция, на выполнение которой нужно очень много времени? Например, представьте, что вам надо выполнить какое-то сложно преобразование изображения с помощью JavaScript в браузере.  
  
«А в чём тут проблема?», — спросите вы. Проблема заключается в том, что до тех пор, пока в стеке вызовов имеется выполняющаяся функция, браузер не может выполнять другие задачи — он оказывается заблокированным. Это означает, что браузер не может выводить ничего на экран, не может выполнять другой код. Он просто останавливается. Подобные эффекты, например, несовместимы с интерактивными интерфейсами.  
  
Однако, это — не единственная проблема. Если браузер начинает заниматься обработкой тяжёлых задач, он может на достаточно долгое время перестать реагировать на какие-либо воздействия. Большинство браузеров в подобной ситуации выдают ошибку, спрашивая пользователя о том, хочет ли он завершить выполнение сценария и закрыть страницу.



*Браузер предлагает завершить выполнение страницы*  
  
Пользователям подобные вещи точно не понравятся.  
  
Итак, как же выполнять тяжёлые вычисления, не блокируя пользовательский интерфейс и не подвешивая браузер? Решение этой проблемы заключается в использовании асинхронных функций обратного вызова

**О JS-движках**

JavaScript-движок — это программа, или, другими словами, интерпретатор, выполняющий код, написанный на JavaScript. Движок может быть реализован с использованием различных подходов: в виде обычного интерпретатора, в виде динамического компилятора (или JIT-компилятора), который, перед выполнением программы, преобразует исходный код на JS в байт-код некоего формата.  
  
Вот список популярных реализаций JavaScript-движков.

* [V8](https://en.wikipedia.org/wiki/V8_%28JavaScript_engine%29) — движок с открытым исходным кодом, написан на C++, его разработкой занимается Google.
* [Rhino](https://en.wikipedia.org/wiki/Rhino_%28JavaScript_engine%29) — этот движок с открытым кодом поддерживает Mozilla Foundation, он полностью написан на Java.
* [SpiderMonkey](https://en.wikipedia.org/wiki/SpiderMonkey_%28JavaScript_engine%29) — это самый первый из появившихся JS-движков, который в прошлом применялся в браузере Netscape Navigator, а сегодня — в Firefox.
* [JavaScriptCore](https://en.wikipedia.org/wiki/JavaScriptCore) — ещё один движок с открытым кодом, известный как Nitro и разрабатываемый Apple для браузера Safari.
* [KJS](https://en.wikipedia.org/wiki/KJS_%28KDE%29) — JS-движок KDE, который разработал Гарри Портен для браузера Konqueror, входящего в проект KDE.
* [Chakra (JScript9)](https://en.wikipedia.org/wiki/JScript) — движок для Internet Explorer.
* [Chakra (JavaScript)](https://en.wikipedia.org/wiki/Chakra_%28JavaScript_engine%29) — движок для Microsoft Edge.
* [Nashorn](https://en.wikipedia.org/wiki/Nashorn_%28JavaScript_engine%29) — движок с открытым кодом, являющийся частью OpenJDK, которым занимается Oracle.
* [JerryScript](https://en.wikipedia.org/wiki/JerryScript) — легковесный движок для интернета вещей.

В этом материале мы остановимся на особенностях V8.

**Почему был создан движок V8?**

Движок с открытым кодом V8 был создан компанией Google, он написан на C++. Движок используется в браузере Google Chrome. Кроме того, что отличает V8 от других движков, он применяется в популярной серверной среде Node.js.

При проектировании V8 разработчики задались целью улучшить производительность JavaScript в браузерах. Для того, чтобы добиться высокой скорости выполнения программ, V8 транслирует JS-код в более эффективный машинный код, не используя интерпретатор. Движок компилирует JavaScript-код в машинные инструкции в ходе исполнения программы, реализуя механизм динамической компиляции, как и многие современные JavaScript-движки, например, SpiderMonkey и Rhino (Mozilla). Основное различие заключается в том, что V8 не использует при исполнении JS-программ байт-код или любой промежуточный код.

**О двух компиляторах, которые использовались в V8**

Внутреннее устройство V8 изменилось с выходом версии 5.9, которая появилась совсем недавно. До этого же он использовал два компилятора:

* full-codegen — простой и очень быстрый компилятор, который выдаёт сравнительно медленный машинный код.
* Crankshaft — более сложный оптимизирующий JIT-компилятор, который генерирует хорошо оптимизированный код.

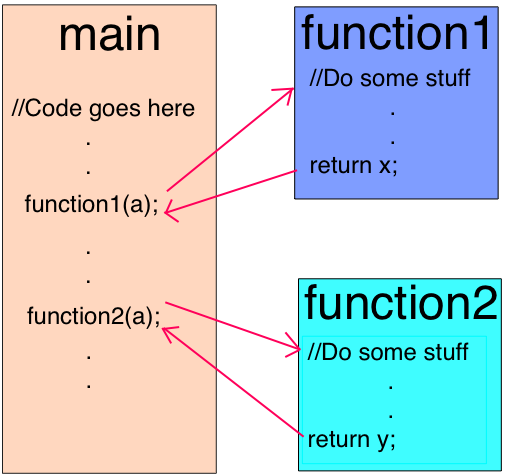
Внутри движка используются несколько потоков:

* Главный поток, который занимается тем, что от него можно ожидать: читает исходный JS-код, компилирует его и выполняет.
* Поток компиляции, который занимается оптимизацией кода в то время, когда выполняется главный поток.
* Поток профилировщика, который сообщает системе о том, в каких методах программа тратит больше всего времени, как результат, Crankshaft может эти методы оптимизировать.
* Несколько потоков, которые поддерживают механизм сборки мусора.

При первом исполнении JS-кода V8 задействует компилятор full-codegen, который напрямую, без каких-либо дополнительных трансформаций, транслирует разобранный им JavaScript-код в машинный код. Это позволяет очень быстро приступить к выполнению машинного кода. Обратите внимание на то, что V8 не использует промежуточное представление программы в виде байт-кода, таким образом, устраняя необходимость в интерпретаторе.  
  
После того, как код какое-то время поработает, поток профилировщика соберёт достаточно данных для того, чтобы система могла понять, какие методы нужно оптимизировать.  
  
Далее, в другом потоке, начинается оптимизация с помощью Crankshaft. Он преобразует абстрактное синтаксическое дерево JavaScript в высокоуровневое представление, использующее модель единственного статического присваивания (static single-assignment, SSA). Это представление называется Hydrogen. Затем Crankshaft пытается оптимизировать граф потока управления Hydrogen. Большинство оптимизаций выполняется на этом уровне.

**Встраивание кода**

Первая оптимизация программы заключается в заблаговременном встраивании в места вызовов как можно большего объёма кода. Встраивание кода — это процесс замены команды вызова функции (строки, где вызывается функция) на её тело. Этот простой шаг позволяет сделать следующие оптимизации более результативными.



*Вызов функции заменяется на её тело*

<https://habrahabr.ru/company/ruvds/blog/337460/>

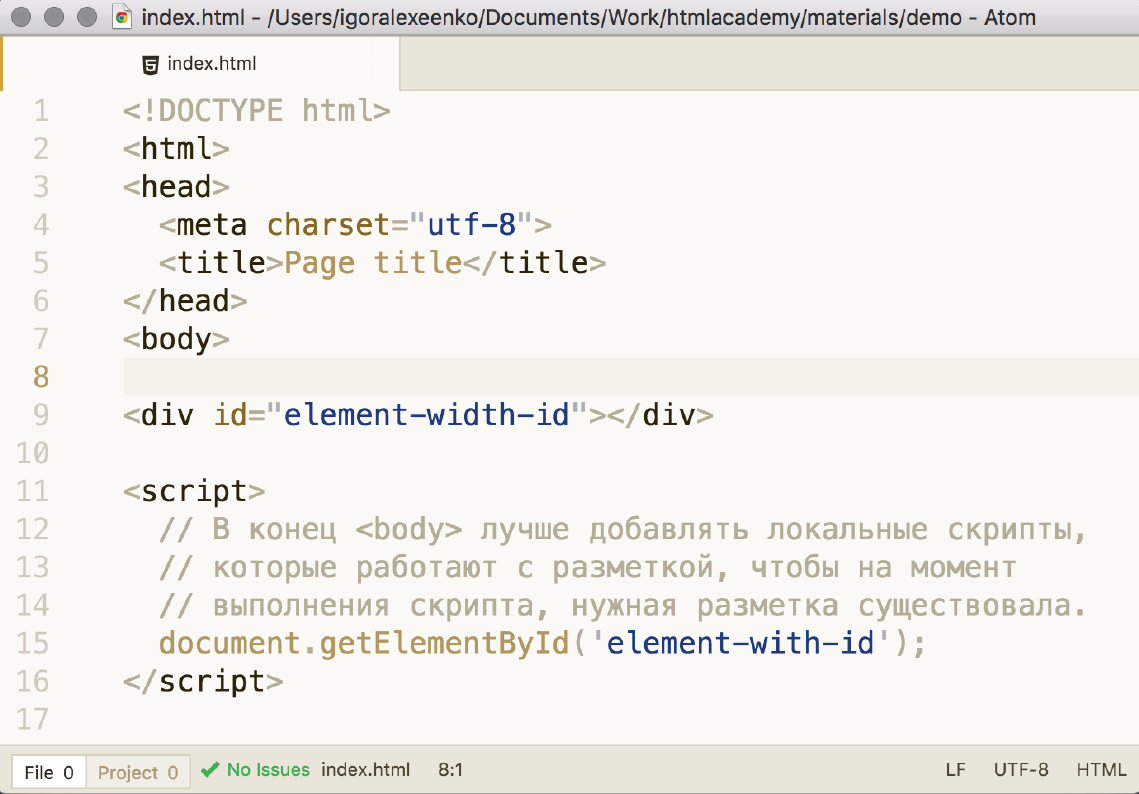
**4 типа ошибок:**

**Syntax error**. Неверное использование языка. Часто лишние или недостающие скобки или кавычки. **Что делать?** Заменить, удалить или добавить символы. Часто проблема в скобках или кавычках: открытые скобки должны быть закрыты, открытые кавычки должны быть закрыты.

**Reference error**. Использование несуществующего названия. **Что делать?** Проверить, существует ли то, на что вы ссылаетесь. Возможно вы использовали ошибочное название или забыли его создать.

**Type error**. Использование неверного типа, например попытка вызвать константу числа, как функцию. **Что делать?** Убедиться, что всё используется верно. Часто проблема в простой путанице: вы создали численную константу и функциональную константу, а потом пытаетесь вызвать число. Вероятно вы собирались вызвать функцию.

**Logic error**. (Логическая ошибка) Ваш код выполняет не то, что требуется, но программа запускается и не выдаёт ошибок трёх перечисленных выше типов. Сломана логика. **Что делать?** Проверить свой код, убедиться, что он выполняет то, что должен.

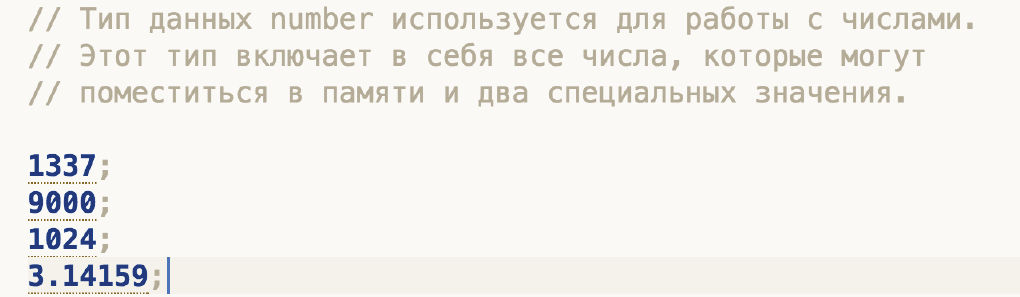


# Типы данных

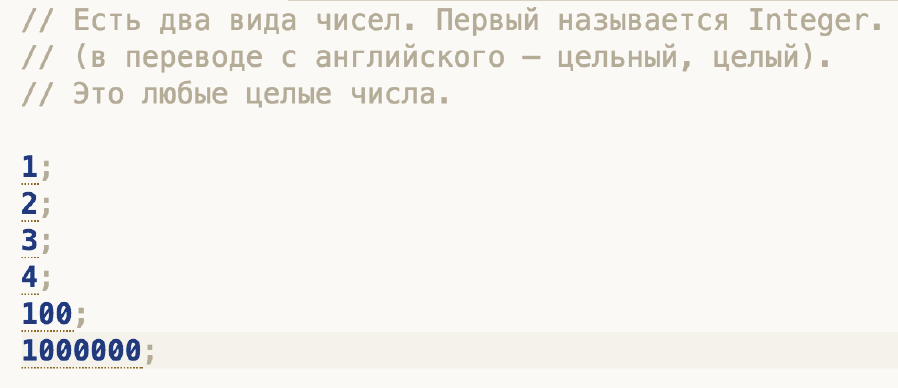
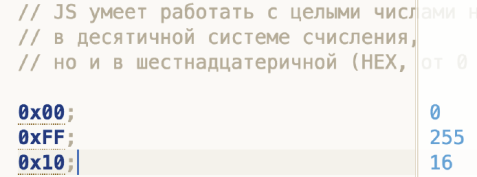
Примивные:

## Числа (number)

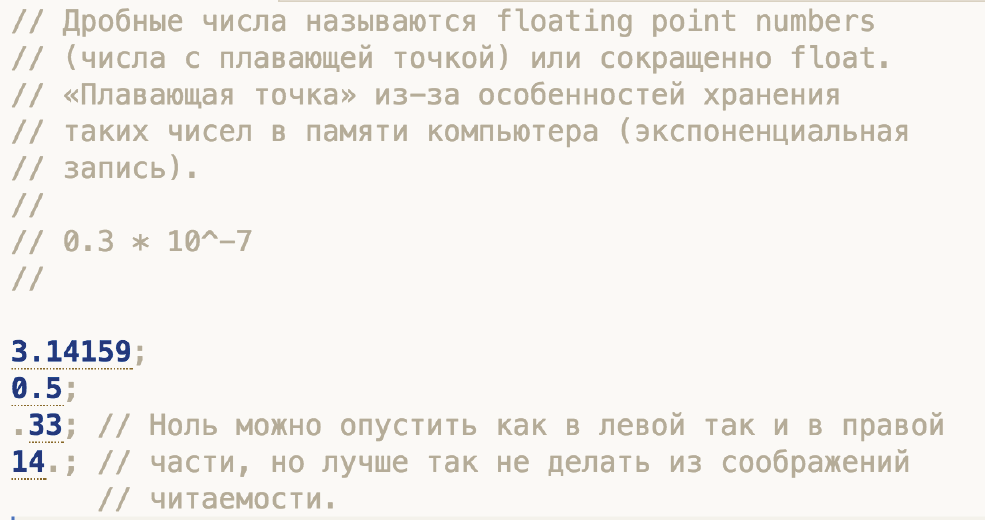
Все числа в JavaScript, как целые так и дробные, имеют тип Number и хранятся в 64-битном формате [IEEE-754](http://en.wikipedia.org/wiki/IEEE_754-1985), также известном как «double precision».



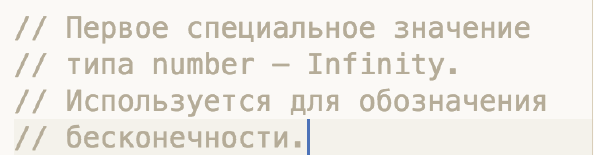
Целые числа (integer numbers):



Дробные числа (floating point numbers):



Бесконечность (Infinity ):



Infinity – особенное численное значение, которое ведет себя в точности как математическая бесконечность ∞.

- Infinity больше любого числа.

- Добавление к бесконечности не меняет её.

alert( Infinity > 1234567890 ); // true

alert( Infinity + 5 == Infinity ); // true

Бесконечность можно присвоить и в явном виде: var x = Infinity.

Бывает и минус бесконечность -Infinity:

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

Бесконечность можно получить также, если сделать ну очень большое число, для которого количество разрядов в двоичном представлении не помещается в соответствующую часть стандартного 64-битного формата, например:

alert( 1e500 ); // Infinity

Ошибка вычислений (Nan):

Если математическая операция не может быть совершена, то возвращается специальное значение NaN (Not-A-Number).

Например, деление 0/0 в математическом смысле неопределено, поэтому его результат NaN:

alert( 0 / 0 ); // NaN

Значение NaN используется для обозначения математической ошибки и обладает следующими свойствами:

Значение NaN – единственное в своем роде, которое *не равно ничему, включая* себя.

Следующий код ничего не выведет:

if (NaN == NaN) alert( "==" ); // Ни один вызов

if (NaN === NaN) alert( "===" ); // не сработает

Значение NaN можно проверить специальной функцией isNaN(n), которая преобразует аргумент к числу и возвращает true, если получилось NaN, и false – для любого другого значения.

var n = 0 / 0;

alert( isNaN(n) ); // true

alert( isNaN("12") ); // false, строка преобразовалась к обычному числу 12

Значение NaN «прилипчиво». Любая операция с NaN возвращает NaN.

alert( NaN + 1 ); // NaN

Если аргумент isNaN – не число, то он автоматически преобразуется к числу.

Забавный способ проверки на NaN

Отсюда вытекает забавный способ проверки значения на NaN: можно проверить значение на равенство самому себе, если не равно – то NaN:

var n = 0 / 0;

if (n !== n) alert( 'n = NaN!' );

Это работает, но для наглядности лучше использовать isNaN(n).

### Способы записи

В JavaScript можно записывать числа не только в десятичной, но и в шестнадцатеричной (начинается с 0x) системе счисления:

alert( 0xFF ); // 255 в шестнадцатиричной системе

Также доступна запись в «научном формате» (ещё говорят «запись с плавающей точкой»), который выглядит как <число>e<количество нулей>.

Например, 1e3 – это 1 с 3 нулями, то есть 1000.

// еще пример научной формы: 3 с 5 нулями

alert( 3e5 ); // 300000

Если количество нулей отрицательно, то число сдвигается вправо за десятичную точку, так что получается десятичная дробь:

// здесь 3 сдвинуто 5 раз вправо, за десятичную точку.

alert( 3e-5 ); // 0.00003 <-- 5 нулей, включая начальный ноль

### isFinite(n)

Итак, в JavaScript есть обычные числа и три специальных числовых значения: NaN, Infinity и -Infinity.

Тот факт, что они, хоть и особые, но числа, демонстрируется работой оператора +:

var value = prompt("Введите Infinity", 'Infinity');

var number = +value;

alert( number ); // Infinity, плюс преобразовал строку "Infinity" к такому "числу"

Обычно если мы хотим от посетителя получить число, то Infinity или NaN нам не подходят. Для того чтобы отличить «обычные» числа от таких специальных значений, существует функция isFinite.

**Функция isFinite(n) преобразует аргумент к числу и возвращает true, если это не NaN/Infinity/-Infinity:**

alert( isFinite(1) ); // true

alert( isFinite(Infinity) ); // false

alert( isFinite(NaN) ); // false

### Преобразование к числу

Большинство арифметических операций и математических функций преобразуют значение в число автоматически.

Для того чтобы сделать это явно, обычно перед значением ставят унарный плюс '+':

var s = "12.34";

alert( +s ); // 12.34

При этом, если строка не является в точности числом, то результат будет NaN:

alert( +"12test" ); // NaN тоже самое будет при +"12 test"

Единственное исключение – пробельные символы в начале и в конце строки, которые игнорируются:

alert( +" -12" ); // -12

alert( +" \n34 \n" ); // 34, перевод строки \n является пробельным символом

alert( +"" ); // 0, пустая строка становится нулем

alert( +"1 2" ); // NaN, пробел посередине числа - ошибка

Аналогичным образом происходит преобразование и в других математических операторах и функциях:

alert( '12.34' / "-2" ); // -6.17

### Мягкое преобразование: parseInt и parseFloat

В мире HTML/CSS многие значения не являются в точности числами. Например метрики CSS: 10pt или -12px.

Оператор '+' для таких значений возвратит NaN:

alert(+"12px") // NaN

Для удобного чтения таких значений существует функция parseInt:

alert( parseInt('12px') ); // 12

**Функция parseInt и ее аналог parseFloat преобразуют строку символ за символом, пока это возможно.**

При возникновении ошибки возвращается число, которое получилось. Функция parseInt читает из строки целое число, а parseFloat – дробное.

alert( parseInt(' 12px') ) // 12, ошибка на символе 'p'

alert( parseFloat('12.3.4') ) // 12.3, ошибка на второй точке

Конечно, существуют ситуации, когда parseInt/parseFloat возвращают NaN. Это происходит при ошибке на первом же символе:

alert( parseInt('a123') ); // NaN

Функция parseInt также позволяет указать систему счисления, то есть считывать числа, заданные в шестнадцатиричной и других системах счисления:

parseInt("ff"); //NaN

parseInt('FF', 16); // 255

parseInt(“0x10”); // 16

parseInt(“0x10”,”10”); // 0

### Экспоненциальное преобразование

var x = 123456789; // undefined

x.toExponential(); //'1.23456789e+8'

x.toExponential(1); //'1.2e+8'

x.toExponential(2); //'1.23e+8'

x.toExponential(3); //'1.235e+8'

### Проверка на число

Для проверки строки на число можно использовать функцию isNaN(str).

Она преобразует строку в число аналогично +, а затем вернёт true, если это NaN, то есть если преобразование не удалось:

var x = prompt("Введите значение", "-11.5");

if (isNaN(x)) {

alert( "Строка преобразовалась в NaN. Не число" );

} else {

alert( "Число" );

}

Однако, у такой проверки есть две особенности:

1. Пустая строка и строка из пробельных символов преобразуются к 0, поэтому считаются числами.
2. Если применить такую проверку не к строке, то могут быть сюрпризы, в частности isNaN посчитает числами значения false, true, null, так как они хотя и не числа, но преобразуются к ним.

alert( isNaN(null) ); // false - не NaN, т.е. "число"

alert( isNaN("\n \n") ); // false - не NaN, т.е. "число"

Если такое поведение допустимо, то isNaN – приемлемый вариант.

Если же нужна действительно точная проверка на число, которая не считает числом строку из пробелов, логические и специальные значения, а также отсекает Infinity – используйте следующую функцию isNumeric:

function isNumeric(n) {

return !isNaN(parseFloat(n)) && isFinite(n);

}

Разберёмся, как она работает. Начнём справа.

* Функция isFinite(n) преобразует аргумент к числу и возвращает true, если это не Infinity/-Infinity/NaN.

Таким образом, правая часть отсеет заведомо не-числа, но оставит такие значения как true/false/null и пустую строку '', так как они корректно преобразуются в числа.

* Для их проверки нужна левая часть. Вызов parseFloat(true/false/null/'') вернёт NaN для этих значений.

Так устроена функция parseFloat: она преобразует аргумент к строке, то есть true/false/null становятся "true"/"false"/"null", а затем считывает из неё число, при этом пустая строка даёт NaN.

В результате отсеивается всё, кроме строк-чисел и обычных чисел.

### toString(система счисления)

Как показано выше, числа можно записывать не только в 10-ричной, но и в 16-ричной системе. Но бывает и противоположная задача: получить 16-ричное представление числа. Для этого используется метод toString(основание системы), например:

var n = 255;

alert( n.toString(16) ); // ff

В частности, это используют для работы с цветовыми значениями в браузере, вида #AABBCC.

Основание может быть любым от 2 до 36.

* Основание 2 бывает полезно для отладки побитовых операций:

var n = 4;

n.toString(); // '4'

n.toString(2); // 100

Основание 36 (по количеству букв в английском алфавите – 26, вместе с цифрами, которых 10) используется для того, чтобы «кодировать» число в виде буквенно-цифровой строки. В этой системе счисления сначала используются цифры, а затем буквы от a до z:

var n = 1234567890;

alert( n.toString(36) ); // kf12oi

При помощи такого кодирования можно «укоротить» длинный цифровой идентификатор, например чтобы выдать его в качестве URL

JavaScript берет целую часть числа и конвертирует в необходимое количество битов, например:

0 – два бита (знак + 0)

1 – два бита

2 – три бита

### Округление

Одна из самых частых операций с числом – округление. В JavaScript существуют целых 3 функции для этого.

Math.floor

Округляет вниз

Math.ceil

Округляет вверх

Math.round

Округляет до ближайшего целого

alert( Math.floor(3.1) ); // 3 если 3.5 -> 3

alert( Math.ceil(3.1) ); // 4 если 3.5 -> 4

alert( Math.round(3.1) ); // 3 если 3.5 -> 4

**Округление битовыми операторами**

Битовые операторы делают любое число 32-битным целым, обрезая десятичную часть.

В результате побитовая операция, которая не изменяет число, например, двойное битовое НЕ – округляет его:

alert( ~~12.3 ); // 12

Любая побитовая операция такого рода подойдет, например XOR (исключающее ИЛИ, "^") с нулем:

alert( 12.3 ^ 0 ); // 12

alert( 1.2 + 1.3 ^ 0 ); // 2, приоритет ^ меньше, чем +

Это удобно в первую очередь тем, что легко читается и не заставляет ставить дополнительные скобки как Math.floor(...):

var x = a \* b / c ^ 0; // читается как "a \* b / c и округлить"

### Округление до заданной точности

Для округления до нужной цифры после запятой можно умножить и поделить на 10 с нужным количеством нулей. Например, округлим 3.456 до 2-го знака после запятой:

var n = 3.456;

alert( Math.round(n \* 100) / 100 ); // 3.456 -> 345.6 -> 346 -> 3.46

Таким образом можно округлять число и вверх и вниз.

### num.toFixed(precision)

Существует также специальный метод num.toFixed(precision), который округляет число num до точности precision и возвращает результат *в виде строки*:

var n = 12.34;

alert( n.toFixed(1) ); // "12.3"

Округление идёт до ближайшего значения, аналогично Math.round:

var n = 12.36;

alert( n.toFixed(1) ); // "12.4"

Итоговая строка, при необходимости, дополняется нулями до нужной точности:

var n = 12.34;

alert( n.toFixed(5) ); // **"**12.34000**"**, добавлены нули до 5 знаков после запятой

Если нам нужно именно число, то мы можем получить его, применив '+' к результату n.toFixed(..):

var n = 12.34;

alert( **+**n.toFixed(5) ); // 12.34

Метод toFixed не эквивалентен Math.round!

Например, произведём округление до одного знака после запятой с использованием двух способов: toFixed и Math.round с умножением и делением:

var price = 6.35;

alert( price.toFixed(1) ); // 6.3

alert( Math.round(price \* 10) / 10 ); // 6.4

Как видно результат разный! Вариант округления через Math.round получился более корректным, так как по общепринятым правилам 5 округляется вверх. А toFixed может округлить его как вверх, так и вниз.

### Неточные вычисления

Запустите этот пример:

alert( 0.1 + 0.2 == 0.3 );

Запустили? Если нет – все же сделайте это.

Ок, вы запустили его. Он вывел false. Результат несколько странный, не так ли? Возможно, ошибка в браузере? Поменяйте браузер, запустите еще раз.

Хорошо, теперь мы можем быть уверены: 0.1 + 0.2 это не 0.3. Но тогда что же это?

alert( 0.1 + 0.2 ); // 0.30000000000000004

Как видите, произошла небольшая вычислительная ошибка, результат сложения 0.1 + 0.2 немного больше, чем 0.3.

alert( 0.1 + 0.2 > 0.3 ); // true

Всё дело в том, что в стандарте IEEE 754 на число выделяется ровно 8 байт(=64 бита), не больше и не меньше.

Число 0.1 (одна десятая) записывается просто в десятичном формате. Но в двоичной системе счисления это бесконечная дробь, так как единица на десять в двоичной системе так просто не делится. Также бесконечной дробью является 0.2 (=2/10).

Двоичное значение бесконечных дробей хранится только до определенного знака, поэтому возникает неточность. Её даже можно увидеть:

alert( 0.1.toFixed(20) ); // 0.10000000000000000555

Когда мы складываем 0.1 и 0.2, то две неточности складываются, получаем незначительную, но всё же ошибку в вычислениях.

Конечно, это не означает, что точные вычисления для таких чисел невозможны. Они возможны. И даже необходимы.

Например, есть два способа сложить 0.1 и 0.2:

1. Сделать их целыми, сложить, а потом поделить:

alert( (0.1 \* 10 + 0.2 \* 10) / 10 ); // 0.3

 Это работает, так как числа 0.1\*10 = 1 и 0.2\*10 = 2 могут быть точно представлены в двоичной системе.

 Сложить, а затем округлить до разумного знака после запятой. Округления до 10-го знака обычно бывает достаточно, чтобы отсечь ошибку вычислений:

var result = 0.1 + 0.2;

alert( +result.toFixed(10) ); // 0.3

Забавный пример

Привет! Я – число, растущее само по себе!

alert( 9999999999999999 ); // выведет 10000000000000000

Причина та же – потеря точности.

Из 64 бит, отведённых на число, сами цифры числа занимают до 52 бит, остальные 11 бит хранят позицию десятичной точки и один бит – знак. Так что если 52 бит не хватает на цифры, то при записи пропадут младшие разряды.

Интерпретатор не выдаст ошибку, но в результате получится «не совсем то число», что мы и видим в примере выше. Как говорится: «как смог, так записал».

**Т.о. когда код компилируется или интерпретируется, «0,1» уже округляется до ближайшего номера в этом формате, что приводит к небольшой ошибке округления даже до того, как произойдет расчет.**

Ради справедливости заметим, что в точности то же самое происходит в любом другом языке, где используется формат IEEE 754, включая Java, C, PHP, Ruby, Perl.

Конвертер Числа с плавающей точкой => двоичный код https://www.h-schmidt.net/FloatConverter/IEEE754.html

Прочитать: http://docs.oracle.com/cd/E19957-01/806-3568/ncg\_goldberg.html

Другие математические методы

JavaScript предоставляет базовые тригонометрические и некоторые другие функции для работы с числами.

### Тригонометрия

Встроенные функции для тригонометрических вычислений:

Math.acos(x)

Возвращает арккосинус x (в радианах)

Math.asin(x)

Возвращает арксинус x (в радианах)

Math.atan(x)

Возвращает арктангенс x (в радианах)

Math.atan2(y, x)

Возвращает угол до точки (y, x). Описание функции: [Atan2](http://en.wikipedia.org/wiki/Atan2).

Math.sin(x)

Вычисляет синус x

Math.cos(x)

Вычисляет косинус x

Math.tan(x)

Возвращает тангенс x

Тригонометрические функции в JavaScript принимают аргументы только в радианах!

function testMath(a) {

var radian = a \* Math.PI / 180;

return Math.sin( radian );

}

### Функции общего назначения

Разные полезные функции:

Math.sqrt(x) // Возвращает квадратный корень из x.

Math.log(x) // Возвращает натуральный (по основанию e) логарифм x.

Math.pow(x, exp) // Возводит число в степень, возвращает xexp, например Math.pow(2,3) = 8. Работает в том числе с дробными и отрицательными степенями, например: Math.pow(4, -1/2) = 0.5. Или 2 \*\* 3 = 8

Math.abs(x) // Возвращает абсолютное значение числа ( модуль )

Math.exp(x) // Возвращает ex, где e – основание натуральных логарифмов.

Math.max(a, b, c...) // Возвращает наибольший из списка аргументов

Math.min(a, b, c...) // Возвращает наименьший из списка аргументов

Math.random() // Возвращает псевдослучайное число в интервале [0,1) – то есть между 0 (включительно) и 1 (не включая). Генератор случайных чисел инициализуется текущим временем.

### Константы

**Math.E** - число е, основание натурального логарифма, константа Эйлера (Непера), приблизительно 2.718...  
**Math.PI** - число Пи, приблизительно равно, как известно, 3.1415926...  
**Math.SQRT2** -квадратный корень из 2, приблизительное значение 1.414  
**Math.SQRT1\_2** -квадратный корень из 1/2, приблизительное значение 0.707  
**Math.LN2** - натуральный логарифм 2, приблизительное значение 0.693  
**Math.LN10** - натуральный логарифм 10, приблизительное значение 2.302  
**Math.LOG2E** - логарифм Е по основанию 2, приблизительное значение 1.442  
**Math.LOG10E** -  логарифм Е по основанию 10, приблизительное значение 0.434

В качестве примера рассмотрим функцию, которая получает радиус окружности и возвращает ее длину:

function circumference(radius) {  
  return radius \* Math.PI \* 2;  
}

Новейшая версия JavaScript на момент написания этих строк (обычно называемая "ES6") включает в себя новый способ объявления *констант*, используя const вместо var:

// согласно ES6:

const TAX\_RATE = 0.08;

// ..

Если вы попытаетесь присвоить любое значение в TAX\_RATE после её объявления, ваша программа отвергнет это изменение (а в строгом (strict) режиме, прервется с ошибкой.

### Форматирование

Для красивого вывода чисел в стандарте [ECMA 402](http://www.ecma-international.org/ecma-402/1.0/ECMA-402.pdf) есть метод toLocaleString():

var number = 123456789;

alert( number.toLocaleString() ); // 123 456 789

Его поддерживают все современные браузеры, кроме IE10- (для которых нужно подключить библиотеку [Intl.JS](https://github.com/andyearnshaw/Intl.js/)). Он также умеет форматировать валюту и проценты. Более подробно про устройство этого метода можно будет узнать в статье [Intl: интернационализация в JavaScript](http://learn.javascript.ru/intl), когда это вам понадобится.

**Итого**

Числа могут быть записаны в десятеричной, шестнадцатиричной системах, а также «научным» способом.

В JavaScript существует числовое значение бесконечность Infinity.

Ошибка вычислений дает NaN.

Арифметические и математические функции преобразуют строку в точности в число, игнорируя начальные и конечные пробелы.

Функции parseInt/parseFloat делают числа из строк, которые начинаются с числа.

Есть четыре способа округления: Math.floor, Math.round, Math.ceil и битовый оператор. Для округления до нужного знака используйте +n.toFixed(p) или трюк с умножением и делением на 10p.

Дробные числа дают ошибку вычислений. При необходимости ее можно отсечь округлением до нужного знака.

Случайные числа от 0 до 1 генерируются с помощью Math.random(), остальные – преобразованием из них.

Существуют и другие математические функции. Вы можете ознакомиться с ними в справочнике в разделах [Number](http://javascript.ru/Number) и [Math](http://javascript.ru/Math).

### Задачи

**Создайте страницу, которая предлагает ввести два числа и выводит их сумму.**

var a = +prompt("Введите первое число", "");

var b = +prompt("Введите второе число", "");

alert( a + b );

Обратите внимание на оператор + перед prompt, он сразу приводит вводимое значение к числу. Если бы его не было, то a и b были бы строками и складывались бы как строки, то есть "1" + "2" = "12".

**Почему 6.35.toFixed(1) == 6.3?**

В математике принято, что 5 округляется вверх, например:

alert( 1.5.toFixed(0) ); // 2

alert( 1.35.toFixed(1) ); // 1.4

Но почему в примере ниже 6.35 округляется до 6.3?

alert( 6.35.toFixed(1) ); // 6.3

Во внутреннем двоичном представлении 6.35 является бесконечной двоичной дробью. Хранится она с потерей точности… А впрочем, посмотрим сами:

alert( 6.35.toFixed(20) ); // 6.34999999999999964473

Интерпретатор видит число как 6.34..., поэтому и округляет вниз.

**Сложение цен**

Представьте себе электронный магазин. Цены даны с точностью до копейки(цента, евроцента и т.п.).

Вы пишете интерфейс для него. Основная работа происходит на сервере, но и на клиенте все должно быть хорошо. Сложение цен на купленные товары и умножение их на количество является обычной операцией.

Получится глупо, если при заказе двух товаров с ценами 0.10$ и 0.20$ человек получит общую стоимость 0.30000000000000004$:

alert( 0.1 + 0.2 + '$' );

Что можно сделать, чтобы избежать проблем с ошибками округления?

Можно хранить сами цены в «копейках» (центах и т.п.). Тогда они всегда будут целые и проблема исчезнет. Но при показе и при обмене данными нужно будет это учитывать и не забывать делить на 100.

При операциях, когда необходимо получить окончательный результат – округлять до 2-го знака после запятой. Все, что дальше – ошибка округления:

var price1 = 0.1, price2 = 0.2;

alert( +(price1 + price2).toFixed(2) );

**Этот цикл – бесконечный. Почему?**

var i = 0;

while (i != 10) {

i += 0.2;

}

Потому что i никогда не станет равным 10.

Запустите, чтобы увидеть *реальные* значения i:

var i = 0;

while (i < 11) {

i += 0.2;

if (i > 9.8 && i < 10.2) alert( i );

} // 9.999999999999996 и 10.199999999999996

Ни одно из них в точности не равно 10.

**Как получить дробную часть числа?**

Напишите функцию getDecimal(num), которая возвращает десятичную часть числа:

alert( getDecimal(12.345) ); // 0.345

alert( getDecimal(1.2) ); // 0.2

alert( getDecimal(-1.2) ); // 0.2

P.S. Постарайтесь не использовать toFixed

function getDecimal(num) {

return num - Math.floor(num);

}

alert( getDecimal(12.5) ); // 0.5

alert( getDecimal(-1.2) ); // 0.8, неверно!

Как видно из примера выше, для отрицательных чисел она не работает.

Это потому, что округление Math.floor происходит всегда к ближайшему меньшему целому, то есть Math.floor(-1.2) = -2, а нам бы хотелось убрать целую часть, т.е. получить -1.

Можно попытаться решить проблему так:

function getDecimal(num) {

return num > 0 ? num - Math.floor(num) : Math.ceil(num) - num;

}

alert( getDecimal(12.5) ); // 0.5

alert( getDecimal(-1.2) ); // 0.19999999999999996, неверно!

alert( getDecimal(1.2) ); // 0.19999999999999996

Проблема с отрицательными числами решена, но результат, увы, не совсем тот.

Внутреннее неточное представление чисел приводит к ошибке в вычислениях, которая проявляется при работе и с положительными и с отрицательными числами.

Давайте попробуем ещё вариант – получим остаток при делении на 1. При таком делении от любого числа в остатке окажется именно дробная часть:

function getDecimal(num) {

return num > 0 ? (num % 1) : (-num % 1);

}

alert( getDecimal(12.5) ); // 0.5

alert( getDecimal(1.2) ); // 0.19999999999999996, неверно!

В общем-то, работает, функция стала короче, но, увы, ошибка сохранилась.

Что делать?

Увы, операции с десятичными дробями подразумевают некоторую потерю точности.

Зависит от ситуации.

* Если внешний вид числа неважен и ошибка в вычислениях допустима – она ведь очень мала, то можно оставить как есть.
* Перейти на промежуточные целочисленные вычисления там, где это возможно.
* Если мы знаем, что десятичная часть жёстко ограничена, к примеру, может содержать не более 2 знаков то можно округлить число, то есть вернуть +num.toFixed(2).

Если эти варианты не подходят, то можно работать с числом как со строкой:

function getDecimal(num) {

var str = "" + num; // преобразуем в строку

var zeroPos = str.indexOf("."); // возвращает индекс “.”

if (zeroPos == -1) return num; // число целое

str = str.slice(zeroPos); // обрезаем строку после “.” до конца

return +str;

}

alert( getDecimal(12.5) ); // 0.5

alert( getDecimal(1.2) ); // 0.2

**Формула Бине**

Последовательность [чисел Фибоначчи](http://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%B0_%D0%A4%D0%B8%D0%B1%D0%BE%D0%BD%D0%B0%D1%87%D1%87%D0%B8) имеет формулу Fn = Fn-1 + Fn-2. То есть, следующее число получается как сумма двух предыдущих.

Первые два числа равны 1, затем 2(1+1), затем 3(1+2), 5(2+3) и так далее: 1, 1, 2, 3, 5, 8, 13, 21....

Код для их вычисления (из задачи [Числа Фибоначчи](http://learn.javascript.ru/task/fibonacci-numbers)):

function fib(n) {

var a = 1,

b = 0,

x;

for (i = 0; i < n; i++) {

x = a + b;

a = b

b = x;

}

return b;

}

Существует [формула Бине](http://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%B0_%D0%A4%D0%B8%D0%B1%D0%BE%D0%BD%D0%B0%D1%87%D1%87%D0%B8#.D0.A4.D0.BE.D1.80.D0.BC.D1.83.D0.BB.D0.B0_.D0.91.D0.B8.D0.BD.D0.B5), согласно которой Fn равно ближайшему целому для ϕn/√5, где ϕ=(1+√5)/2 – золотое сечение.

Напишите функцию fibBinet(n), которая будет вычислять Fn, используя эту формулу. Проверьте её для значения F77 (должно получиться fibBinet(77) = 5527939700884757).

**Одинаковы ли результаты, полученные при помощи кода fib(n) выше и по формуле Бине? Если нет, то почему и какой из них верный?**

function fibBinet(n) {

var phi = (1 + Math.sqrt(5)) / 2;

// используем Math.round для округления до ближайшего целого

return Math.round(Math.pow(phi, n) / Math.sqrt(5));

}

function fib(n) {

var a = 1,

b = 0,

x;

for (i = 0; i < n; i++) {

x = a + b;

a = b

b = x;

}

return b;

}

alert( fibBinet(2) ); // 1, равно fib(2)

alert( fibBinet(8) ); // 21, равно fib(8)

alert( fibBinet(77) ); // 5527939700884755

alert( fib(77) ); // 5527939700884757, не совпадает!

**Результат вычисления F77 получился неверным!**

Причина – в ошибках округления, ведь √5 – бесконечная дробь.

Ошибки округления при вычислениях множатся и, в итоге, дают расхождение.

**Случайное из интервала (0, max)**

Напишите код для генерации случайного значения в диапазоне от 0 до max, не включая max.

Сгенерируем значение в диапазоне 0..1 и умножим на max:

var max = 10;

alert( Math.random() \* max );

**Случайное из интервала (min, max)**

Напишите код для генерации случайного числа от min до max, не включая max.

Сгенерируем значение из интервала 0..max-min, а затем сдвинем на min:

var min = 5,

max = 10;

alert( min + Math.random() \* (max - min) );

**Случайное целое от min до max**

Напишите функцию randomInteger(min, max) для генерации случайного **целого** числа между min и max, включая min,max как возможные значения.

Любое число из интервала min..max должно иметь одинаковую вероятность.

**Очевидное неверное решение (round)**

Самый простой, но неверный способ – это сгенерировать значение в интервале min..max и округлить его Math.round, вот так:

function randomInteger(min, max) {

var rand = min + Math.random() \* (max - min)

rand = Math.round(rand);

return rand;

}

alert( randomInteger(1, 3) );

Эта функция работает. Но при этом она некорректна: вероятность получить крайние значения min и max будет в два раза меньше, чем любые другие.

При многократном запуске этого кода вы легко заметите, что 2 выпадает чаще всех.

Это происходит из-за того, что Math.round() получает разнообразные случайные числа из интервала от 1 до 3, но при округлении до ближайшего целого получится, что:

значения из диапазона 1 ... 1.49999.. станут 1

значения из диапазона 1.5 ... 2.49999.. станут 2

значения из диапазона 2.5 ... 2.99999.. станут 3

Отсюда явно видно, что в 1 (как и 3) попадает диапазон значений в два раза меньший, чем в 2. Из-за этого такой перекос.

**Верное решение с round**

Правильный способ: Math.round(случайное от min-0.5 до max+0.5)

function randomInteger(min, max) {

var rand = min - 0.5 + Math.random() \* (max - min + 1)

rand = Math.round(rand);

return rand;

}

alert( randomInteger(5, 10) );

В этом случае диапазон будет тот же (max-min+1), но учтена механика округления round.

**Решение с floor**

Альтернативный путь – применить округление Math.floor() к случайному числу от min до max+1.

Например, для генерации целого числа от 1 до 3, создадим вспомогательное случайное значение от 1 до 4 (не включая 4).

Тогда Math.floor() округлит их так:

1 ... 1.999+ станет 1

2 ... 2.999+ станет 2

3 ... 3.999+ станет 3

Все диапазоны одинаковы. Итак, код:

function randomInteger(min, max) {

var rand = min + Math.random() \* (max + 1 - min);

rand = Math.floor(rand);

return rand;

}

lert( randomInteger(5, 10) );

Напишите функцию для проверки на отрицательные числа:

const isNegative = (num) => {

const res = num < 0;

return res;

};

Или:

const isNegative = num => num < 0;

## Строка (string)



**Строки**

В JavaScript любые текстовые данные являются строками. Не существует отдельного типа «символ», который есть в ряде других языков.

Внутренним форматом строк, вне зависимости от кодировки страницы, является [Юникод (Unicode)](http://ru.wikipedia.org/wiki/%D0%AE%D0%BD%D0%B8%D0%BA%D0%BE%D0%B4).

### Создание строк

Создание объекта String можно выполнить тремя способами:

var myString = 'my string' или “my string” //создание с помощью строкового литерала

var myString = new String(object)   //создание объекта, сейчас такой синтаксис не используют

var myString = String(object)

Как правило такие объекты создают неявно, строковым литералом: *var myString = 'Some text;'  
(строковым литералом называют любую последовательность символов, заключенную в любые кавычки)*

В строковом литерале вы можете использовать любые символы а также *escape-последовательности*, если необходимо указать символ, который, например, нельзя набрать на клавиатуре. Выглядит это следующим образом:  var myString = '\u1234'; где 1234 - номер знака в таблице юникода.

**Важный момент!** При создании объекта с помощью литерала (текста в кавычках) вы получаете тип объекта "string", а при создании с помощью ключевого слова "new" вы получите тип "Object", этому объекту можно будет сразу напрямую назначать дополнительные свойства и методы.

### Специальные символы

Строки могут содержать специальные символы. Самый часто используемый из таких символов – это «перевод строки».

Он обозначается как \n, например:

alert( 'Привет\nМир' ); // выведет "Мир" на новой строке

Есть и более редкие символы, вот их список:

| Специальные символы | |
| --- | --- |
| **Символ** | **Описание** |
| \b | Backspace |
| \f | Form feed |
| \n | New line |
| \r | Carriage return |
| \t | Tab |
| \uNNNN | Символ в кодировке Юникод с шестнадцатеричным кодом `NNNN`. Например, `\u00A9` -- юникодное представление символа копирайт © |

### Экранирование специальных символов

Если строка в одинарных кавычках, то внутренние одинарные кавычки внутри должны быть *экранированы*, то есть снабжены обратным слешем \', вот так:

var str = 'I\'m a JavaScript programmer';

В двойных кавычках – экранируются внутренние двойные:

var str = "I'm a JavaScript \"programmer\" ";

alert( str ); // I'm a JavaScript "programmer"

Экранирование служит исключительно для правильного восприятия строки JavaScript. В памяти строка будет содержать сам символ без '\'. Вы можете увидеть это, запустив пример выше.

Сам символ обратного слэша '\' является служебным, поэтому всегда экранируется, т.е пишется как \\:

var str = ' символ \\ ';

alert( str ); // символ \

Заэкранировать можно любой символ. Если он не специальный, то ничего не произойдёт:

alert( "\a" ); // a

// идентично alert( "a" );

### Методы и свойства

Здесь мы рассмотрим методы и свойства строк,

### Длина length

Одно из самых частых действий со строкой – это получение ее длины:

var str = "My\n"; // 3 символа. Третий - перевод строки

alert( str.length ); // 3

Значение длины пустой строки, естественно, будет равно 0.

### Доступ к символам

Чтобы получить символ, используйте вызов **charAt(позиция)**. Первый символ имеет позицию 0:

var str = "jQuery";

alert( str.charAt(0) ); // "j"

В JavaScript **нет отдельного типа «символ»**, так что charAt возвращает строку, состоящую из выбранного символа.

Также для доступа к символу можно использовать квадратные скобки:

var str = "Я - современный браузер!";

alert( str[0] ); // "Я"

Разница между этим способом и charAt заключается в том, что если символа нет – charAt выдает пустую строку, а скобки – undefined:

alert( "".charAt(0) ); // пустая строка

alert( "" [0] ); // undefined

Вообще же метод charAt существует по историческим причинам, ведь квадратные скобки – проще и короче.

Вызов метода – всегда со скобками

Обратите внимание, str.length – это *свойство* строки, а str.charAt(pos) – *метод*, т.е. функция.

Обращение к методу всегда идет со скобками, а к свойству – без скобок.

### Изменения строк

Содержимое строки в JavaScript нельзя изменять. Нельзя взять символ посередине и заменить его. Как только строка создана – она такая навсегда.

Можно лишь создать целиком новую строку и присвоить в переменную вместо старой, например:

var str = "строка";

str = str[3] + str[4] + str[5];

alert( str ); // ока

### Смена регистра

Методы **toLowerCase()** и **toUpperCase()** меняют регистр строки на нижний/верхний:

alert( "Интерфейс".toUpperCase() ); // ИНТЕРФЕЙС

Пример ниже получает первый символ и приводит его к нижнему регистру:

alert( "Интерфейс" [0].toLowerCase() ); // 'и'

### Поиск подстроки

**indexOf**(searchValue[, fromIndex]) - ищет подстроку searchValue начиная с позиции fromIndex

Он возвращает позицию, на которой находится подстрока или -1, если ничего не найдено. Например:

var str = "Widget with id";

alert( str.indexOf("Widget") ); // 0, т.к. "Widget" найден прямо в начале str

alert( str.indexOf("id") ); // 1, т.к. "id" найден, начиная с позиции 1 (Widget)

alert( str.indexOf("widget") ); // -1, не найдено, так как поиск учитывает регистр

Для начала разберем работу с методом **indexOf**(searchValue[, fromIndex])

**var** myString = "Это просто проверочная строка!";  
console.log(myString.**indexOf**("про", 0));

В данном случае в консоль будет выведен номер позиции с которой начинается первая найденная подстрока  "про" - **4**.  
"Это просто проверочная строка!";

Однако, если мы укажем методу искать подстроку не с самого начала, а, например, с 5-й позиции:

console.log(myString.indexOf("про", 5));

То тогда результат будет "11" - позиция, с которой начинается второе вхождение подстроки "про".

Также существует аналогичный метод **lastIndexOf**, который ищет не с начала, а с конца строки.

**lastIndexOf**(searchValue[, fromIndex]) - ищет последнюю подстроку  searchValue начиная с позиции fromIndex

Метод  **lastIndexOf**(searchValue[, **fromIndex**]) действует аналогично, но возвращает номер, с которого начинается последнее вхождение искомой подстроки. **fromIndex** обозначает место, с которого нужно начинать поиск по направлению к началу строки, его значение по-умолчанию - длина строки. В примере:

**var** myString = "Это просто проверочная строка!";  
**console.log**(myString.**lastIndexOf**("про"));

в консоль будет выведена цифра 11 - начало последнего вхождения подстроки "про".

**search**(regExp) - проверяет - есть ли совпадение с аргументом, результат true или false  
**match**(regExp) - выдает массив всех на совпадение с аргументом

**Преобразование типов**

**toString**() - возвращает элементарную строку вместо объекта String   
**valueOf**() - возвращает элементарную строку вместо объекта String, эквивалентно toString()   
В результате выполнения приведенного ниже кода в консоль будет выведено элементарное значение.

myString = new String("Hello world");  
console.log(myString.valueOf());

На заметку:

**Для красивого вызова indexOf применяется побитовый оператор НЕ '~'.**

Дело в том, что вызов ~n эквивалентен выражению -(n+1), например:

alert( ~2 ); // -(2+1) = -3

alert( ~1 ); // -(1+1) = -2

alert( ~0 ); // -(0+1) = -1

alert( ~-1 ); // -(-1+1) = 0

Как видно, ~n – ноль только в случае, когда n == -1.

То есть, проверка if ( ~str.indexOf(...) ) означает, что результат indexOf отличен от -1, т.е. совпадение есть.

Вот так:

var str = "Widget";

if (~str.indexOf("hello userget")) {

alert( 'совпадение есть!' );

}

Вообще, использовать возможности языка неочевидным образом не рекомендуется, поскольку ухудшает читаемость кода.

Однако, в данном случае, все в порядке. Просто запомните: '~' читается как «не минус один», а "if ~str.indexOf" читается как "если найдено".

**Поиск всех вхождений**

Чтобы найти все вхождения подстроки, нужно запустить indexOf в цикле. Как только получаем очередную позицию – начинаем следующий поиск со следующей.

Пример такого цикла:

var str = "Ослик Иа-Иа посмотрел на виадук"; // ищем в этой строке

var target = "Иа"; // цель поиска

var pos = 0;

while (true) {

var foundPos = str.indexOf(target, pos);

if (foundPos == -1) break;

alert( foundPos ); // нашли на этой позиции

pos = foundPos + 1; // продолжить поиск со следующей

}

Такой цикл начинает поиск с позиции 0, затем найдя подстроку на позиции foundPos, следующий поиск продолжит с позиции pos = foundPos+1, и так далее, пока что-то находит.

Впрочем, тот же алгоритм можно записать и короче:

var str = "Ослик Иа-Иа посмотрел на виадук"; // ищем в этой строке

var target = "Иа"; // цель поиска

var pos = -1;

while ((pos = str.indexOf(target, pos + 1)) != -1) {

alert( pos );

}

### Взятие подстроки: substr, substring, slice.

В JavaScript существуют целых 3 (!) метода для взятия подстроки, с небольшими отличиями между ними.

**substring**(start [, end])

Метод substring(start, end) возвращает подстроку с позиции start до, но не включая end.

var str = "stringify";

alert(str.substring(0,1)); // "s", символы с позиции 0 по 1 не включая 1.

Если аргумент end отсутствует, то идет до конца строки:

var str = "stringify";

alert(str.substring(2)); // ringify, символы с позиции 2 до конца

**substr**(start [, length])

Первый аргумент имеет такой же смысл, как и в substring, а второй содержит не конечную позицию, а количество символов.

var str = "stringify";

str = str.substr(2,4); // ring, со 2-й позиции 4 символа

alert(str)

Если второго аргумента нет – подразумевается «до конца строки».

**slice**(start [, end])

Возвращает часть строки от позиции start до, но не включая, позиции end. Смысл параметров – такой же как в substring.

**replace**(regexp, newSubString|function)

возвращает новую строку после замен, указанных в regexp, или функцию, которая ее возвращает.

### Отрицательные аргументы

Различие между substring и slice – в том, как они работают с отрицательными и выходящими за границу строки аргументами:

substring(start, end)

Отрицательные аргументы интерпретируются как равные нулю. Слишком большие значения усекаются до длины строки:

alert( "testme".substring(-2) ); // "testme", -2 становится 0

Кроме того, если start > end, то аргументы меняются местами, т.е. возвращается участок строки *между* start и end:

alert( "testme".substring(4, -1) ); // "test"

// -1 становится 0 -> получили substring(4, 0)

// 4 > 0, так что аргументы меняются местами -> substring(0, 4) = "test"

slice отрицательные значения отсчитываются от конца строки:

alert( "testme".slice(-2) ); // "me", от 2 позиции с конца

alert( "testme".slice(1, -1) ); // "estm", от 1 позиции до первой с конца.

Это гораздо более удобно, чем странная логика substring.

Отрицательное значение первого параметра поддерживается в substr во всех браузерах, кроме IE8-.

Если выбирать из этих трёх методов один, для использования в большинстве ситуаций – то это будет slice: он и отрицательные аргументы поддерживает и работает наиболее очевидно.

### Интерполяция

Кроме одиночных '' и двойных кавычек "", современный JavaScript содержит обратные тики (backticks):

``

С обратными тиками вы можете использовать **интерполяцию**, вместо конкатенации. Вот, смотрите:

const name = "Alex;"

const a = 10;

const b = 12;

console.log(`His name was ${name} and his age was ${a + b}`);

Такой код выведет на экран His name was Alex; and his age was 22.

Внутрь ${} вы можете поместить любое выражение.

### Кодировка Юникод

Как мы знаем, символы сравниваются в алфавитном порядке 'А' < 'Б' < 'В' < ... < 'Я'.

Но есть несколько странностей…

1. Почему буква 'а' маленькая больше буквы 'Я' большой?

alert( 'а' > 'Я' ); // true

Буква 'ё' находится в алфавите между е и ж: абвгде**ё**жз…. Но почему тогда 'ё' больше 'я'?

alert( 'ё' > 'я' ); // true

Чтобы разобраться с этим, обратимся к внутреннему представлению строк в JavaScript.

### Все строки имеют внутреннюю кодировку [Юникод](http://ru.wikipedia.org/wiki/%D0%AE%D0%BD%D0%B8%D0%BA%D0%BE%D0%B4).

Неважно, на каком языке написана страница, находится ли она в windows-1251 или utf-8. Внутри JavaScript-интерпретатора все строки приводятся к единому «юникодному» виду. Каждому символу соответствует свой код.

Есть метод для получения символа по его коду:

String.fromCharCode(code)

Возвращает символ по коду code:

alert( String.fromCharCode(1072) ); // 'а'

…И метод для получения цифрового кода из символа:

str.charCodeAt(pos)

Возвращает код символа на позиции pos. Отсчет позиции начинается с нуля.

alert( "абрикос".charCodeAt(0) ); // 1072, код 'а'

Теперь вернемся к примерам выше. Почему сравнения 'ё' > 'я' и 'а' > 'Я' дают такой странный результат?

Дело в том, что **символы сравниваются не по алфавиту, а по коду**. У кого код больше – тот и больше. В юникоде есть много разных символов. Кириллическим буквам соответствует только небольшая часть из них, подробнее – [Кириллица в Юникоде](http://ru.wikipedia.org/wiki/%D0%9A%D0%B8%D1%80%D0%B8%D0%BB%D0%BB%D0%B8%D1%86%D0%B0_%D0%B2_%D0%AE%D0%BD%D0%B8%D0%BA%D0%BE%D0%B4%D0%B5).

Выведем отрезок символов юникода с кодами от 1034 до 1113:

var str = '';

for (var i = 1034; i <= 1113; i++) {

str += String.fromCharCode(i);

}

alert( str );

Результат:

ЊЋЌЍЎЏАБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзийклмнопрстуфхцчшщъыьэюяѐёђѓєѕіїјљ

Мы можем увидеть из этого отрезка две важных вещи:

1. **Строчные буквы идут после заглавных, поэтому они всегда больше.**

В частности, 'а'(код 1072) > 'Я'(код 1071).

То же самое происходит и в английском алфавите, там 'a' > 'Z'.

1. **Ряд букв, например ё, находятся вне основного алфавита.**

В частности, маленькая буква ё имеет код, больший чем я, поэтому **'ё'(код 1105) > 'я'(код 1103)**.

Кстати, большая буква Ё располагается в Unicode до А, поэтому **'Ё'(код 1025) < 'А'(код 1040)**. Удивительно: есть буква меньше чем А :)

**Буква ё не уникальна, точки над буквой используются и в других языках, приводя к тому же результату.**

Например, при работе с немецкими названиями:

alert( "ö" > "z" ); // true

### Юникод в HTML

Кстати, если мы знаем код символа в кодировке юникод, то можем добавить его в HTML, используя «числовую ссылку» (numeric character reference).

Для этого нужно написать сначала &#, затем код, и завершить точкой с запятой ';'. Например, символ 'а' в виде числовой ссылки: &#1072;.

Если код хотят дать в 16-ричной системе счисления, то начинают с &#x.

В юникоде есть много забавных и полезных символов, например, символ ножниц: ✂ (&#x2702;), дроби: ½ (&#xBD;) ¾ (&#xBE;) и другие. Их можно использовать вместо картинок в дизайне.

### Посимвольное сравнение

Сравнение строк работает *лексикографически*, иначе говоря, посимвольно.

Сравнение строк s1 и s2 обрабатывается по следующему алгоритму:

1. Сравниваются первые символы: s1[0] и s2[0]. Если они разные, то сравниваем их и, в зависимости от результата их сравнения, возвратить true или false. Если же они одинаковые, то…
2. Сравниваются вторые символы s1[1] и s2[1]
3. Затем третьи s1[2] и s2[2] и так далее, пока символы не будут наконец разными, и тогда какой символ больше – та строка и больше. Если же в какой-либо строке закончились символы, то считаем, что она меньше, а если закончились в обеих – они равны.

Спецификация языка определяет этот алгоритм более детально. Если же говорить простыми словами, смысл алгоритма в точности соответствует порядку, по которому имена заносятся в орфографический словарь.

"Вася" > "Ваня" // true, т.к. начальные символы совпадают, а потом 'с' > 'н'

"Дома" > "До" // true, т.к. начало совпадает, но в 1-й строке больше символов

Числа в виде строк сравниваются как строки

Бывает, что числа приходят в скрипт в виде строк, например как результат prompt. В этом случае результат их сравнения будет неверным:

alert( "2" > "14" ); // true, так как это строки, и для первых символов верно "2" > "1"

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

Если хотя бы один аргумент – не строка, то другой будет преобразован к числу:

alert( 2 > "14" ); // false

### Правильное сравнение

Все современные браузеры, кроме IE10- (для которых нужно подключить библиотеку [Intl.JS](https://github.com/andyearnshaw/Intl.js/)) поддерживают стандарт [ECMA 402](http://www.ecma-international.org/ecma-402/1.0/ECMA-402.pdf), поддерживающий сравнение строк на разных языках, с учётом их правил.

Способ использования:

var str = "Ёлки";

alert( str.localeCompare("Яблони") ); // -1

Метод str1.localeCompare(str2) возвращает -1, если str1 < str2, 1, если str1 > str2 и 0, если они равны.

Более подробно про устройство этого метода можно будет узнать в статье [Intl: интернационализация в JavaScript](http://learn.javascript.ru/intl), когда это вам понадобится.

**Итого**

Строки в JavaScript имеют внутреннюю кодировку Юникод. При написании строки можно использовать специальные символы, например \n и вставлять юникодные символы по коду.

Мы познакомились со свойством length и методами charAt, toLowerCase/toUpperCase, substring/substr/slice (предпочтителен slice). Есть и другие методы, например [trim](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/Trim) обрезает пробелы с начала и конца строки.

Строки сравниваются побуквенно. Поэтому если число получено в виде строки, то такие числа могут сравниваться некорректно, нужно преобразовать его к типу *number*.

При сравнении строк следует иметь в виду, что буквы сравниваются по их кодам. Поэтому большая буква меньше маленькой, а буква ё вообще вне основного алфавита.

Для правильного сравнения существует целый стандарт ECMA 402. Это не такое простое дело, много языков и много правил. Он поддерживается во всех современных браузерах, кроме IE10-, в которых нужна библиотека <https://github.com/andyearnshaw/Intl.js/>. Такое сравнение работает через вызов str1.localeCompare(str2).

Больше информации о методах для строк можно получить в справочнике: <http://javascript.ru/String>.

### Задачи

**Сделать первый символ заглавным**

Напишите функцию ucFirst(str), которая возвращает строку str с заглавным первым символом, например:

ucFirst("вася") == "Вася";

ucFirst("") == ""; // нет ошибок при пустой строке

Создайте функцию, используя toUpperCase() и charAt().

Мы не можем просто заменить первый символ, т.к. строки в JavaScript неизменяемы.

Но можно пересоздать строку на основе существующей, с заглавным первым символом:

var newStr = str[0].toUpperCase() + str.slice(1);

Однако, есть небольшая проблемка – в случае, когда строка пуста, будет ошибка.

Ведь str[0] == undefined, а у undefined нет метода toUpperCase().

Выхода два. Первый – использовать str.charAt(0), он всегда возвращает строку, для пустой строки – пустую, но не undefined. Второй – отдельно проверить на пустую строку, вот так:

function ucFirst(str) {

// только пустая строка в логическом контексте даст false

if (!str) return str; //! приводит операнд к true или false и возвращает противоположное значение

return str[0].to UpperCase() + str.slice(1);

}

alert( ucFirst("вася") );

**Проверьте спам**

Напишите функцию checkSpam(str), которая возвращает true, если строка str содержит „viagra“ или „XXX“, а иначе false.

Функция должна быть нечувствительна к регистру:

checkSpam('buy ViAgRA now') == true

checkSpam('free xxxxx') == true

checkSpam("innocent rabbit") == false

Метод indexOf ищет совпадение с учетом регистра. То есть, в строке 'xXx' он не найдет 'XXX'.

Для проверки, сначала приведем строку str к нижнему регистру, а затем уже будем искать.

function checkSpam(mail) {

var lowerStr = mail.toLowerCase();

return !!(~lowerStr.indexOf('viagra') || ~lowerStr.indexOf('xxx')); // !! приведение к логическому типу

}

**Усечение строки**

Создайте функцию truncate(str, maxlength), которая проверяет длину строки str, и если она превосходит maxlength – заменяет конец str на "...", так чтобы ее длина стала равна maxlength.

Результатом функции должна быть (при необходимости) усечённая строка.

Например:

truncate("Вот, что мне хотелось бы сказать на эту тему:", 20) = "Вот, что мне хоте..."

truncate("Всем привет!", 20) = "Всем привет!"

Так как окончательная длина строки должна быть maxlength, то нужно её обрезать немного короче, чтобы дать место для троеточия.

function truncate(str, maxlength) {

if (str.length > maxlength) {

return str.slice(0, maxlength - 3) + '...';

// итоговая длина равна maxlength

}

return str;

}

alert( truncate("Вот, что мне хотелось бы сказать на эту тему:", 20) );

alert( truncate("Всем привет!", 20) );

Можно было бы написать этот код ещё короче:

function truncate(str, maxlength) {

return (str.length > maxlength) ?

str.slice(0, maxlength - 3) + '...' : str;

}

P.S. Кстати, в кодировке Unicode существует специальный символ «троеточие»: … (HTML: &hellip;), который можно использовать вместо трёх точек. Если его использовать, то можно отрезать только один символ.

Создайте фуннкцию reverse, которая переворачивает сроку.

const reverse = (str) => {

let i = str.length - 1;

let result = '';

while (i >= 0) {

result += str[i];

i--; // i -= 1;

}

return result;

};

**Реализуйте и экспортируйте по умолчанию функцию addDigits, которая работает следующим образом:**

**Дано неотрицательное целое число num. Складывать все входящие в него цифры до тех пор, пока не останется одна цифра.**

**Для числа 38 процесс будет выглядеть так: 3 + 8 = 11,а для 1259 => 1+2+5 = 8**

const sum = (str) => {

let result = 0;

for (let i = 0; i < length(str); i += 1) {

result += Number(str[i]);

}

return result;

};

const addDigits = (num) => {

let result = num;

while (result >= 10) {

result = sum(String(result));

}

return result;

};

const addDigits = (num) => {

let result = 0;

let str = String(num);

for(let i = 0; i < str.length; i++) {

result += +str[i];

};

if (result < 10) return result;

else return addDigits(result);

}

**Реализуйте и экспортируйте по умолчанию функцию, которая делает заглавной первую букву каждого слова в предложении.**

solution('hello, world!'); // Hello, World!

export default (str) => {

let result = '';

for (let i = 0; i < length(str); i++) {

const shouldBeBig = str[i] !== ' ' && (i === 0 || str[i - 1] === ' ');

result += shouldBeBig ? toUpperCase(str[i]) : str[i];

}

return result;

};

Или:

let getBigFirstLatter = (str) => {

let newStr = str;

if(str[0] !== ' ') newStr = str[0].toUpperCase() + str.slice(1);

for(let i = 0; i < newStr.length; i++) {

if(newStr[i] == ' ') {

newStr = newStr.slice(0, i+1) + newStr[i + 1].toUpperCase() + newStr.slice(i+2);

}

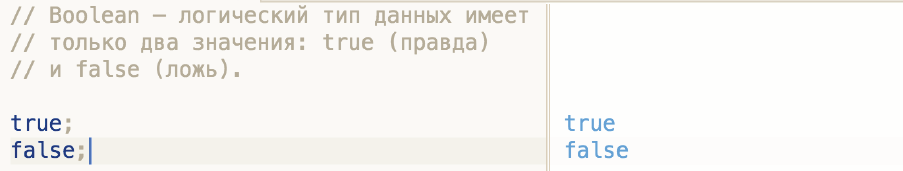
}

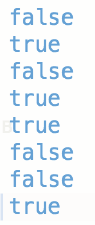
return newStr;

};

export default getBigFirstLatter;

## boolean (логическое значение)





var a = true; // присваивать явно

var b = 3 > 4; // или как результат сравнения

alert( b ); // false

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

**Особый список "ложных" значений в JavaScript таков:**

"" (пустая строка)

0, -0, NaN (некорректное число)

null, undefined

false

Любое значение, не входящее в этот "ложный" список — "истинно."

Вот несколько примеров:

"hello"

42

"0"

" "

true

[ ], [ 1, "2", 3 ] (массивы)

{ }, { a: 42 } (объекты)

function foo() { .. } (функции)

**Сравнение разных типов**

При сравнении значений разных типов, используется числовое преобразование. Оно применяется к обоим значениям.

Например:

alert( '2' > 1 ); // true, сравнивается как 2 > 1

alert( '01' == 1 ); // true, сравнивается как 1 == 1

alert( false == 0 ); // true, false становится числом 0

alert( true == 1 ); // true, так как true становится числом 1.

new Array(3) == ",,"; //true, пустой массив на 3 элемента

new Array(3) === ",,"; //false

**Что использовать  == или === в различных ситуациях правила:**

* Если одно из значений (т.е. сторона) в сравнении может быть значением true или false, избегайте == и используйте ===.
* Если одно из значений в сравнении может быть одним из этих особых значений (0, "" или [] — пустой массив), избегайте == и используйте ===.
* Во *всех* остальных случаях, вы можете безопасно использовать ==. Это не только безопасно, но во многих случаях это упрощает ваш код путем повышения читаемости.

Форма не-равно != идет в паре с ==, а форма !== — в паре с ===. Все правила и утверждения, которые мы только что обсудили также применимы для этих сравнений на не равно.

Следует особо обратить внимение на правила сравнения == и ===, если вы сравниваете два непримитивных значения, таких как object (включая function и array). Так как эти значения на самом деле хранятся по ссылке, оба сравнения == и === просто проверят равны ли ссылки, но ничего не сделают касаемо самих значений.

Например, массив по умолчанию приводится к строке простым присоединением всех значений с запятыми (,) между ними. Можно было бы подумать, что эти два массива с одинаковым содержимым будут равны по ==, но это не так:

var a = [1,2,3];

var b = [1,2,3];

var c = "1,2,3";

a == c; // true

b == c; // true

a == b; // false

Самое большое затруднение, в которое вы можете попасть со сравнениями между потенциально разными типами значений (помните, что нет формы "строгого неравенства"?) — это когда одно из значений не может быть превращено в корректное число, например:

var a = 42;

var b = "foo";

a < b; // false

a > b; // false

a == b; // false

**Строгое равенство**

В обычном операторе == есть «проблема» – он не может отличить 0 от false

alert( 0 == false ); // true

Та же ситуация с пустой строкой:

alert( '' == false ); // true

Это естественное следствие того, что операнды разных типов преобразовались к числу. Пустая строка, как и false, при преобразовании к числу дают 0.

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

Если тип разный, то они всегда возвращают false:

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

Строгое сравнение предпочтительно, если мы хотим быть уверены, что «сюрпризов» не будет.

**Булева логика**

1 && 2 && 3 && 0; логическое и. 0

x < 10 && y > 1 x = 6; y = 3; true

0 || 1 || 2; логическое или. 1

x == 5 || y == 5 при x=6, y= 3; false

!false – логическое отрицание. true

!(x == y) при x=6, y= 3; true

### || (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". В JavaScript, как мы увидим далее, это не совсем так, но для начала рассмотрим только логические значения.

Получается следующая «таблица результатов»:

alert( true || true ); // true

alert( false || true ); // true

alert( true || false ); // true

alert( false || false ); // false

Если значение не логического типа – то оно к нему приводится в целях вычислений. 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);

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

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

При этом оператор ИЛИ возвращает то значение, на котором остановились вычисления. Причём, не преобразованное к логическому типу.

Например:

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

Если все значения «ложные», то || возвратит последнее из них:

alert( undefined || '' || false || 0 ); // 0

Итак, оператор || вычисляет операнды слева направо до первого «истинного» и возвращает его, а если все ложные – то последнее значение. Иначе можно сказать, что "|| запинается на правде".

Пример:

alert( alert(1) || 2 || alert(3) );

Ответ: сначала 1, затем 2.

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.

### && (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 с if:

var hour = 12,

minute = 30;

if (hour == 12 && minute == 30) {

alert( 'Время 12:30' );

}

Как и в ИЛИ, в И 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

Можно передать и несколько значений подряд, при этом возвратится первое «ложное» (на котором остановились вычисления), а если его нет – то последнее:

alert( 1 && 2 && null && 3 ); // null

alert( 1 && 2 && 3 ); // 3

Итак, оператор && вычисляет операнды слева направо до первого «ложного» и возвращает его, а если все истинные – то последнее значение. Иначе можно сказать, что "&& запинается на лжи".

Приоритет оператора И && больше, чем ИЛИ ||, так что 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, alert сработает, если в левой части будет true.

Получился аналог:

var x = 1;

if (x > 0) {

alert( 'Больше' );

}

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

Пример:

alert( alert(1) && alert(2) );

Ответ: 1, а затем undefined.

Вызов alert не возвращает значения, или, иначе говоря, возвращает undefined.

Поэтому до правого alert дело не дойдёт, вычисления закончатся на левом.

alert( null || 2 && 3 || 4 );

Приоритет оператора && выше, чем ||, поэтому он выполнится первым.

Последовательность вычислений:

null || 2 && 3 || 4

null || 3 || 4

3

### ! (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 используют для преобразования значений к логическому типу:

alert( !!"строка" ); // true

alert( !!null ); // false

Оператор не равно != (x != 8 будет true), при x=5;

Оператор строго не равно !==(возвращает **true** если операнды строго не равны или имеют разный тип.)

console.log( 0 !== false ); // true

Пример:

Напишите условие if для проверки того факта, что age НЕ находится между 14 и 90 включительно.

Сделайте два варианта условия: первый с использованием оператора НЕ !, второй – без этого оператора.

Первый вариант:

if (!(age >= 14 && age <= 90))

Второй вариант:

if (age < 14 || age > 90)

## Специальное значение «null»

Значение null не относится ни к одному из типов выше, а образует свой отдельный тип, состоящий из единственного значения null:

var age = null;

Это просто специальное значение, которое имеет смысл «ничего» или «значение неизвестно».

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

Значения null и undefined равны == друг другу и не равны чему бы то ни было ещё. Это жёсткое правило буквально прописано в спецификации языка.

При преобразовании в число null становится 0.

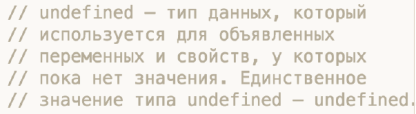
alert( null > 0 ); // false

alert( null == 0 ); // false

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

alert(null >= 0); // true

## Специальное значение undefined:







При преобразовании в число undefined становится NaN.

alert( undefined > 0 ); // false

alert( undefined < 0 ); // false

alert( undefined == 0 ); // false

Вывод: любые сравнения с undefined/null, кроме точного ===, следует делать с осторожностью.

Желательно не использовать сравнения >= > < <= с ними, во избежание ошибок в коде.

## symbol (символ, новое в ES6)

## Объекты «object»

Используется для коллекций данных и для объявления более сложных сущностей.

Объект - это особый вид данных, обладающий свойствами, описывающими его состояние и методами, позволяющими работать с объектом. Объект – контейнер со свойствами. Всё – объекты (кроме чисел, строк, true/false-переменных, null и undefined)

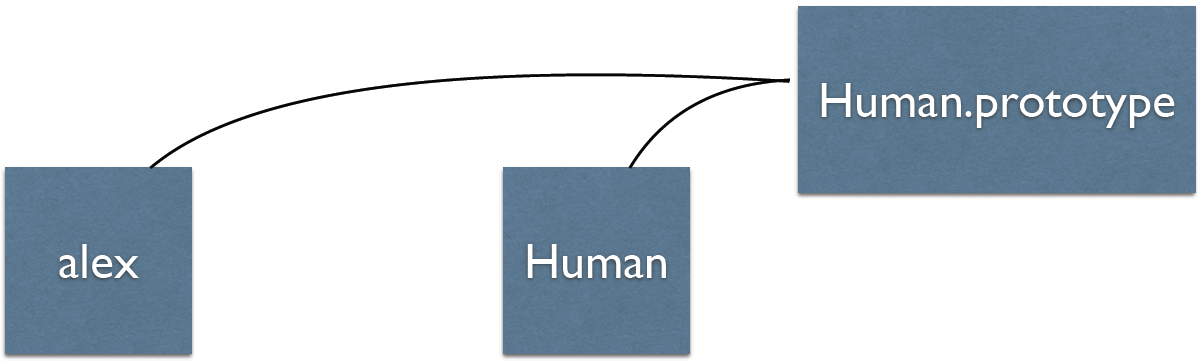
Функция – объект, массив – объект, объект – объект.

Если вдаваться в подробности, то объект в JavaScript представляет собой ассоциативный массив, то есть фактически является набором пар "ключ: значение". Например у строкового объекта *String*, коим является любая строковая переменная, есть свойство *length* - длина. В частности у объекта textString, созданного командой var textString = "Hello World!"; будет свойство *length*, ключом в паре будет имя "length" а его значение будет число 12 - фактическая длина строки.

Свойства объекта - это некоторые значения, связанные с этим объектом.

Синтаксис доступа к свойству объекта выглядит следующим образом: имяОбъекта.имяСвойства

var alex = new Human();

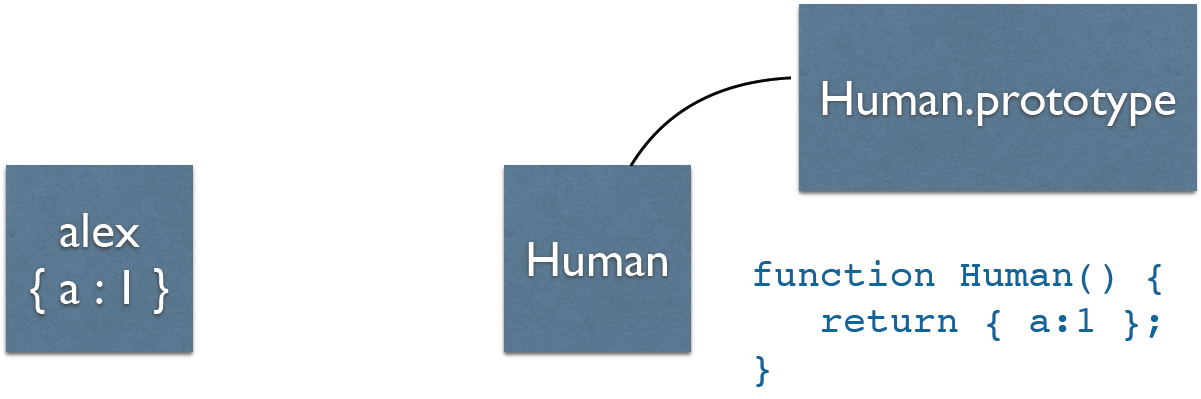


Это работает если функция Human ничего не возвращает

или

возвращает примитивный тип (не объект)

var alex = new Human();



Если функция Human **возвращает объект**, то он и будет возвращен без указания нового прототипа

**Наследование в JS**

• Объект содержит свойства

• Объект содержит специальное свойство, указывающее на прототип объекта

• Объект может переопределять любое свойство прототипа

• Конструктор создает объект. Прототипом этого объекта будет прототип конструктора

Конструктор Human

function Human(name) { // когда работаем с конструктором название с большой буквы

this.name = name; // указывает на новосозданный объект

}

Прототип этого конструктора:

Human.prototype.say = function(what) {

console.log(this.name + " : " + what);

}

var alex = new Human("Alex");

alex; // {name : "Alex" }

alex.say("hello!"); // Alex : hello!

Human("Galex”); //здесь не создается новый объект,тк нет new, при этом создается глобальный объект name = "Galex"

function Human(name) {

if (! (this instanceof Human)) { // проверяет есть ли в цепочке прототипов левого аргумента правый аргумент или др. return new Human(name); // словами: наследует ли this что-то от human, если нет то это был простой вызов как this.name = name; // Human("Galex”);

}

Human("Galex”); // {name: "Galex"}

function Human(name) {

this.name = name;

}

Human.prototype.say = function(what) {

console.log(this.name + ":" + what);

}

var alex = new Human("Alex");

alex.say("hi"); // "Alex: hi"; здесь this ==== jack>

var jack = new Human("Jack");

jack.say("hi") // "Jack: hi"; здесь this ==== jack>

functionName.apply( thisArg, [ params ])

thisArg – объект, который будет this в этом вызове.

params параметры вызова функции functionName

можно использовать arguments

alex.say.apply(jack,["hi"]) //"Jack: hi" apply специально заменить this на объект jack

function speaksTo(someone) {

console.log(this.name + " speaks to " + someone.name);

}

speaksTo(alex); // Galex speaks to Alex; this здесь global name == Galex, без него было бы undefined;

speaksTo.apply(alex, [jack]); // Alex speaks to Jack;

**Создание нового дочернего объекта (Дуглас Крокфорд):**

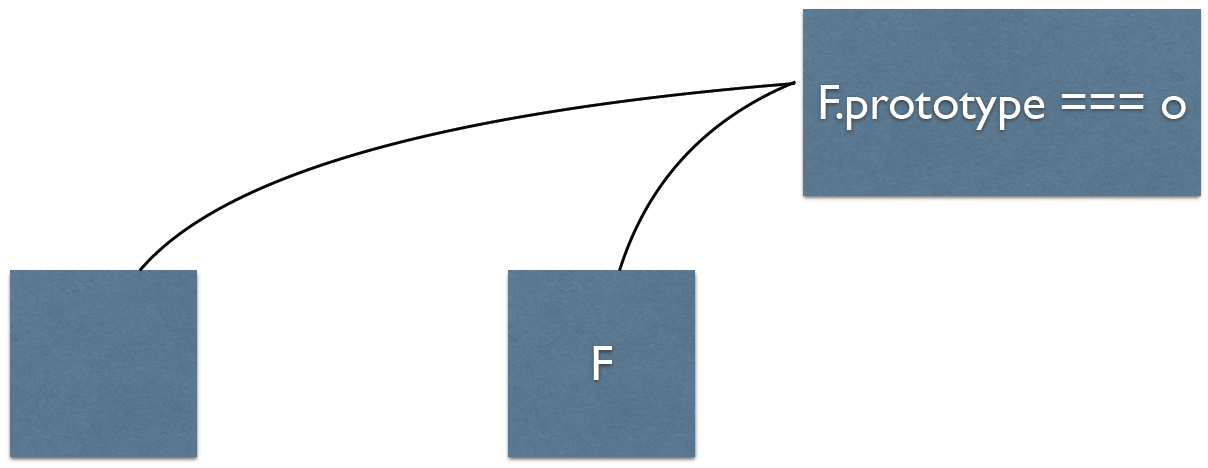
function object(o) {

function F() {} //Пустая функция

F.prototype = o; // прототипом делаем, то что передано в качестве параметра (o);

return new F();

}

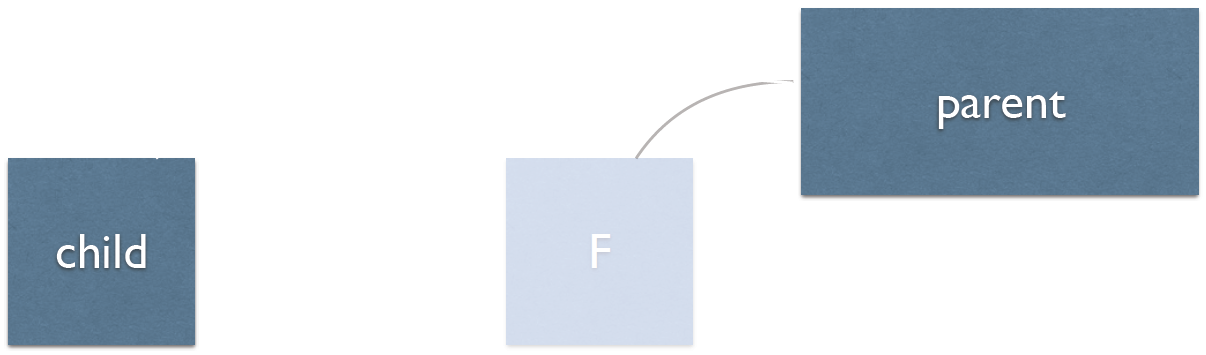


/новый объект/

var parent = { a : 1 };

var child = object(parent);

child.a; //



Тоже самое начиная с ECMAScript 5 JavaScript 1.8.5:

Object.create()

var parent = { name : "Alex" }; // прототип

var child = Object.create(parent); //дочерний объект

child.name //"Alex"

Добавление метода в объект осуществляется с помощью следующего синтаксиса:

var person = {} //Объявляем объект person

person.sayAge = function(n) { //Объявляем метод sayAge для объекта person

  console.log("Person is " + n + " years old"); //Тело метода sayAge - вывод текста

};

В данном примере при вызове функции

person.sayAge(22);

Произойдет вывод в документ текста "Person is 22 years old".

Добавление метода в объект - это фактически присвоение функции некоторому свойству объекта. В предыдущем примере мы присвоили функцию function(n) свойству sayAge объекта person .

### Объекты как ассоциативные массивы

Массив представляет из себя набор значений, доступ к которым осуществляется по их порядковому номеру. Значение в массиве называют "элементом массива" и обращаются к ним по их порядковому номеру, называемому "индексом".

Массивы являются нетипизированными, т.е. позволяют внутри одного массива хранить элементы разных типов. Это могут  быть не только элементарные типы - строки, числа или символы, но также объекты и массивы, и даже массивы массивов.  
 Массивы **начинают отсчет индексов с нуля**, для индексов используются 32-битные значения.

Массивы **являются динамическими**, т.е.они могут увеличиваться или уменьшаться в размерах. Объявлять заранее фиксированные размеры при их создании, а также перераспределять память при изменении размеров, как это делается в некоторых других языках - не нужно. **Массивы являются объектами**!

Объекты в JavaScript сочетают в себе два важных функционала.

Первый – это ассоциативный массив: структура, пригодная для хранения любых данных. Второй – языковые возможности для объектно-ориентированного программирования.

### Ассоциативные массивы

Ассоциативный массив – структура данных, в которой можно хранить любые данные в формате ключ-значение.

Её можно легко представить как шкаф с подписанными ящиками. Все данные хранятся в ящичках. По имени можно легко найти ящик и взять то значение, которое в нём лежит.



В отличие от реальных шкафов, в ассоциативный массив можно в любой момент добавить новые именованные «ящики» или удалить существующие. Далее мы увидим примеры, как это делается.

Кстати, в других языках программирования такую структуру данных также называют «словарь» и «хэш».

### Создание объектов

Пустой объект («пустой шкаф») может быть создан одним из двух синтаксисов:

1. o = new Object();

2. o = {}; // пустые фигурные скобки

Обычно все пользуются синтаксисом (2), т.к. он короче.

### Операции с объектом

Объект может содержать в себе любые значения, которые называются свойствами объекта. Доступ к свойствам осуществляется по имени свойства (иногда говорят «по ключу»).

Например, создадим объект person для хранения информации о человеке:

var person = {}; // пока пустой



Основные операции с объектами – это создание, получение и удаление свойств.

Для обращения к свойствам используется запись «через точку», вида объект.свойство, например:

// при присвоении свойства в объекте автоматически создаётся "ящик"

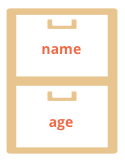
// с именем "name" и в него записывается содержимое 'Вася'

person.name = 'Вася';

person.age = 25; // запишем ещё одно свойство: с именем 'age' и значением 25

Так же можно создавать объект внутри объекта: например:

person.wife = { name: 'Eve', merried: true }



Значения хранятся «внутри» ящиков. Обратим внимание – любые значения, любых типов: число, строка – не важно.

Чтобы прочитать их – также обратимся через точку:

alert( person.name + ': ' + person.age ); // "Вася: 25"

Удаление осуществляется оператором delete:

delete person.wife.merried;

delete person.age;

Осталось только свойство name:



Иногда бывает нужно проверить, есть ли в объекте свойство с определенным ключом.

Для этого есть особый оператор: "in".

Его синтаксис: "prop" in obj, причем имя свойства – в виде строки, например:

if ("name" in person) {

alert( "Свойство name существует!" );

}

Впрочем, чаще используется другой способ – сравнение значения с undefined.

Дело в том, что в JavaScript можно обратиться к любому свойству объекта, даже если его нет.

Ошибки не будет.

Но если свойство не существует, то вернется специальное значение undefined:

var person = {};

alert( person.lalala ); // undefined, нет свойства с ключом lalala

Таким образом мы можем легко проверить существование свойства – получив его и сравнив с undefined:

var person = {

name: "Василий"

};

alert( person.lalala === undefined ); // true, свойства нет

alert( person.name === undefined ); // false, свойство есть.

**Разница между проверками in и === undefined**

Есть два средства для проверки наличия свойства в объекте: первое – оператор in, второе – получить его и сравнить с undefined.

Они почти идентичны, но есть одна небольшая разница.

Дело в том, что технически возможно, что свойство есть, а его значением является undefined:

var obj = {};

obj.test = undefined; // добавили свойство со значением undefined

// проверим наличие свойств test и заведомо отсутствующего blabla

alert( obj.test === undefined ); // true

alert( obj.blabla === undefined ); // true

…При этом, как видно из кода, при простом сравнении наличие такого свойства будет неотличимо от его отсутствия.

Но оператор in гарантирует правильный результат:

var obj = {};

obj.test = undefined;

alert( "test" in obj ); // true

alert( "blabla" in obj ); // false

Как правило, в коде мы не будем присваивать undefined, чтобы корректно работали обе проверки. А в качестве значения, обозначающего неизвестность и неопределенность, будем использовать null.

### Доступ через квадратные скобки

Существует альтернативный синтаксис работы со свойствами, использующий квадратные скобки объект['свойство']:

var person = {};

person['name'] = 'Вася'; // то же что и person.name = 'Вася'

Записи person['name'] и person.name идентичны, но квадратные скобки позволяют использовать в качестве имени свойства любую строку:

var person = {};

person['любимый стиль музыки'] = 'Джаз';

Такое присвоение было бы невозможно «через точку», так интерпретатор после первого пробела подумает, что свойство закончилось, и далее выдаст ошибку:

person.любимый стиль музыки = 'Джаз'; // ??? ошибка

В обоих случаях, **имя свойства обязано быть строкой**. Если использовано значение другого типа – JavaScript приведет его к строке автоматически.

### Доступ к свойству через переменную

Квадратные скобки также позволяют обратиться к свойству, имя которого хранится в переменной:

var person = {};

person.age = 25;

var key = 'age';

alert( person[key] ); // выведет person['age']

Вообще, если имя свойства хранится в переменной (var key = "age"), то единственный способ к нему обратиться – это квадратные скобки person[key].

Доступ через точку используется, если мы на этапе написания программы уже знаем название свойства. А если оно будет определено по ходу выполнения, например, введено посетителем и записано в переменную, то единственный выбор – квадратные скобки.

### Объявление со свойствами

Объект можно заполнить значениями при создании, указав их в фигурных скобках: { ключ1: значение1, ключ2: значение2, ... }.

Такой синтаксис называется литеральным (англ. literal).

Следующие два фрагмента кода создают одинаковый объект:

var menuSetup = {

width: 300,

height: 200,

title: "Menu"

};

// то же самое, что:

var menuSetup = {};

menuSetup.width = 300;

menuSetup.height = 200;

menuSetup.title = 'Menu';

Названия свойств можно перечислять как в кавычках, так и без, если они удовлетворяют ограничениям для имён переменных.

Например:

var menuSetup = {

width: 300,

'height': 200,

"мама мыла раму": true

};

В качестве значения можно тут же указать и другой объект:

var user = {

name: "Таня",

age: 25,

size: {

top: 90,

middle: 60,

bottom: 90

}

}

alert(user.name) // "Таня"

alert(user.size.top) // 90

Здесь значением свойства size является объект {top: 90, middle: 60, bottom: 90 }.

### Компактное представление объектов

Hardcore coders only

Эта секция относится ко внутреннему устройству структуры данных. Она не обязательна к прочтению.

Браузер использует специальное «компактное» представление объектов, чтобы сэкономить память в том случае, когда однотипных объектов много.

Например, посмотрим на такой объект:

var user = {

name: "Vasya",

age: 25

};

Здесь содержится информация о свойстве name и его строковом значении, а также о свойстве age и его численном значении. Представим, что таких объектов много.

Получится, что информация об именах свойств name и age дублируется в каждом объекте. Чтобы этого избежать, браузер применяет оптимизацию.

**При создании множества объектов одного и того же вида (с одинаковыми полями) интерпретатор выносит описание полей в отдельную структуру. А сам объект остаётся в виде непрерывной области памяти с данными.**

Например, есть много объектов с полями name и age:

{name: "Вася", age: 25}

{name: "Петя", age: 22}

{name: "Маша", age: 19}

...

Для их эффективного хранения будет создана структура, которая описывает данный вид объектов. Выглядеть она будет примерно так: <string name, number age>. А сами объекты будут представлены в памяти только данными:

<структура: string name, number age>

Вася 25

Петя 22

Маша 19

При добавлении нового объекта такой структуры достаточно хранить значения полей, но не их имена. Экономия памяти – налицо.

А что происходит, если к объекту добавляется новое свойство? Например, к одному из них добавили свойство isAdmin:

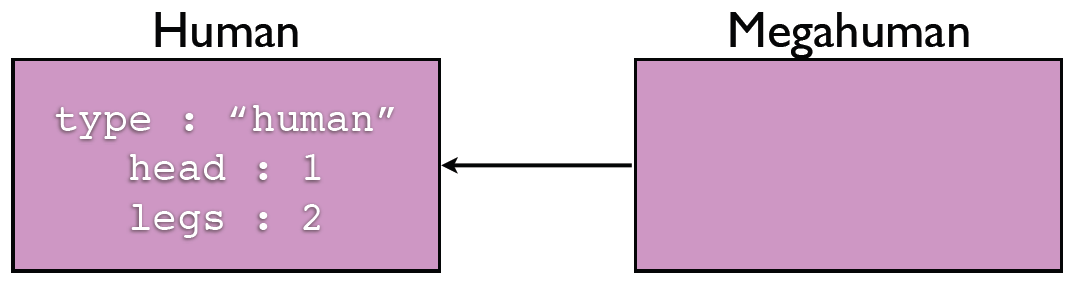
user.isAdmin = true;

В этом случае браузер смотрит, есть ли уже структура, под которую подходит такой объект. Если нет – она создаётся и объект привязывается к ней.

**Эта оптимизация является примером того, что далеко не всё то, что мы пишем, один-в-один переносится в память.**

Современные интерпретаторы очень стараются оптимизировать как код, так и структуры данных. Детали применения и реализации этого способа хранения варьируются от браузера к браузеру. О том, как это сделано в Chrome можно узнать, например, из презентации [Know Your Engines](http://www.slideshare.net/newmovie/know-yourengines-velocity2011). Она была некоторое время назад, но с тех пор мало что изменилось.

### Прототипы



/прототип/

Human.face; // undefined

Megahuman.face; // undefined

Human.face = “okay”;

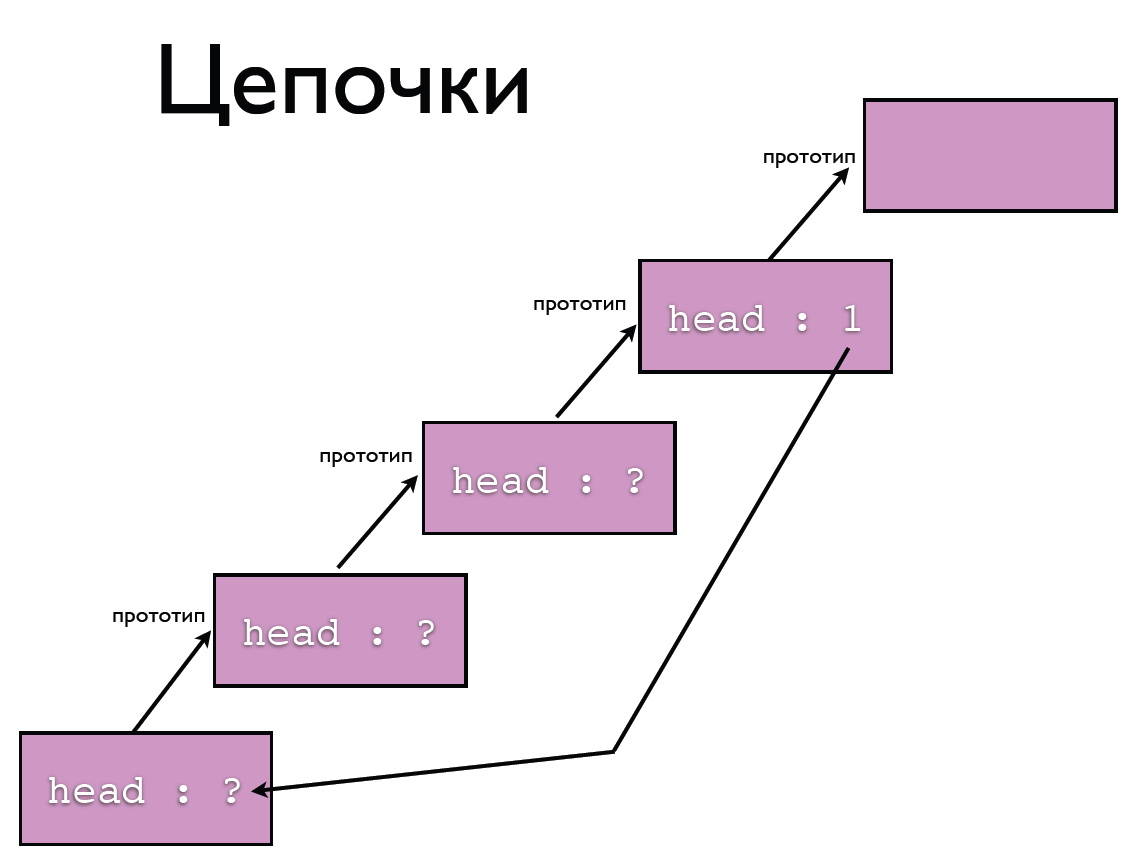
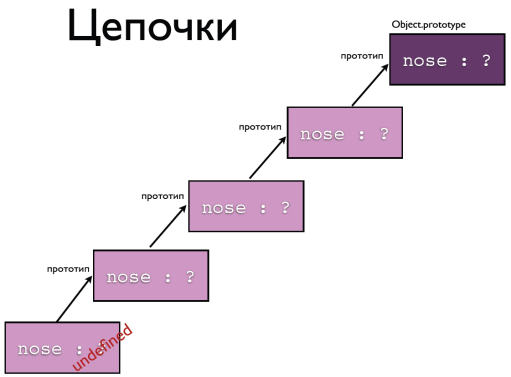
Megahuman.face; // “okay”

Megahuman.face = “awesome”;

Human.face; // “okay”

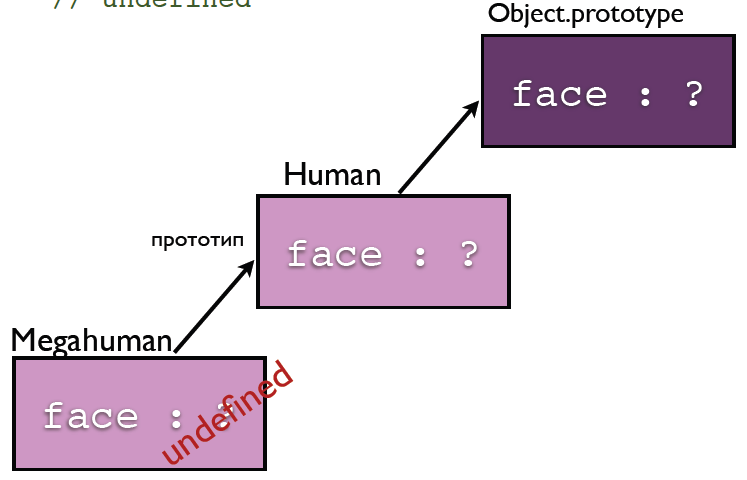
Megahuman.hands = 10;

Human.hands; // undefined



Human.face; // undefined

Megahuman.face; // undefined



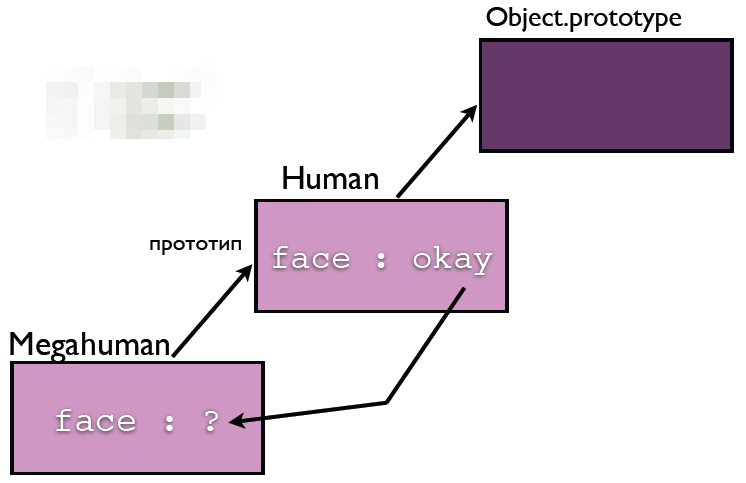
Human.face; // undefined

Megahuman.face; // undefined

Human.face = “okay”;

Human.face; // “okay”

Megahuman.face; // “okay”

****

**Как указать прототип?**

var Human = {

type : “Human”,

head : 1,

legs : 2

};

var Megahuman = Object.create(Human);

**Как удалить свойство?**

var Human = {

type : “Human”,

head : 1,

legs : 2

};

var Megahuman = Object.create(Human);

Megahuman.head = 2;

delete Megahuman.head;

Megahuman.head; // 1

**Итого**

Объекты – это ассоциативные массивы с дополнительными возможностями:

* Доступ к элементам осуществляется:
  + Напрямую по ключу obj.prop = 5
  + Через переменную, в которой хранится ключ:
  + var key = "prop";

obj[key] = 5

Удаление ключей: delete obj.name.

Существование свойства может проверять оператор in: if ("prop" in obj), как правило, работает и просто сравнение if (obj.prop !== undefined).

### Задачи

1. Создайте пустой объект user.
2. Добавьте свойство name со значением Вася.
3. Добавьте свойство surname со значением Петров.
4. Поменяйте значение name на Сергей.
5. Удалите свойство name из объекта.

var user = {};

user.name = "Вася";

user.surname = "Петров";

user.name = "Сергей";

delete user.name;

### Объекты: перебор свойств

Для перебора всех свойств из объекта используется цикл по свойствам for..in. Эта синтаксическая конструкция отличается от рассмотренного ранее цикла for(;;).

#### for..in

Синтаксис:

for (key in obj) {

/\* ... делать что-то с obj[key] ... \*/

}

При этом for..in последовательно переберёт свойства объекта obj, имя каждого свойства будет записано в key и вызвано тело цикла.

Объявление переменной в цикле for (var key in obj)

Вспомогательную переменную key можно объявить прямо в цикле:

for (var key in menu) {

// ...

}

Так иногда пишут для краткости кода. Можно использовать и любое другое название, кроме key, например for(var propName in menu).

Пример итерации по свойствам:

var menu = {

width: 300,

height: 200,

title: "Menu"

};

for (var key in menu) {

// этот код будет вызван для каждого свойства объекта

// ..и выведет имя свойства и его значение

alert( "Ключ: " + key + " значение: " + menu[key] );

}

Обратите внимание, мы использовали квадратные скобки menu[key]. Как уже говорилось, если имя свойства хранится в переменной, то обратиться к нему можно только так, не через точку.

varperson = {  
  name : "Ivan",  
  age : 25,  
  hiredYear : 2017  
}

person.sayAll = function() {  
 for (var i in this) {  
    console.log(i + " is " + this[i]); }  
}

#### Количество свойств в объекте

Как узнать, сколько свойств хранит объект?

Готового метода для этого нет.

Самый кросс-браузерный способ – это сделать цикл по свойствам и посчитать, вот так:

var menu = {

width: 300,

height: 200,

title: "Menu"

};

var counter = 0;

for (var key in menu) {

counter++;

}

alert( "Всего свойств: " + counter );

В следующих главах мы пройдём массивы и познакомимся с другим, более коротким, вызовом: Object.keys(menu).length.

#### В каком порядке перебираются свойства?

Для примера, рассмотрим объект, который задаёт список опций для выбора страны:

var codes = {

// телефонные коды в формате "код страны": "название"

"7": "Россия",

"38": "Украина",

// ..,

"1": "США"

};

Здесь мы предполагаем, что большинство посетителей из России, и поэтому начинаем с 7, это зависит от проекта.

При выборе телефонного кода мы хотели бы предлагать варианты, начиная с первого. Обычно на основе списка генерируется select, но здесь нам важно не это, а важно другое.

**Правда ли, что при переборе for(key in codes) ключи key будут перечислены именно в том порядке, в котором заданы?**

**По стандарту – нет. Но некоторое соглашение об этом, всё же, есть.**

Соглашение говорит, что если имя свойства – нечисловая строка, то такие ключи всегда перебираются в том же порядке, в каком присваивались. Так получилось по историческим причинам и изменить это сложно: поломается много готового кода.

С другой стороны, если имя свойства – число или числовая строка, то все современные браузеры сортируют такие свойства в целях внутренней оптимизации.

К примеру, рассмотрим объект с заведомо нечисловыми свойствами:

var user = {

name: "Вася",

surname: "Петров"

};

user.age = 25;

// порядок перебора соответствует порядку присвоения свойства

for (var prop in user) {

alert( prop ); // name, surname, age

}

А теперь – что будет, если перебрать объект с кодами?

var codes = {

// телефонные коды в формате "код страны": "название"

"7": "Россия",

"38": "Украина",

"1": "США"

};

for (var code in codes) alert( code ); // 1, 7, 38

При запуске этого кода в современном браузере мы увидим, что на первое место попал код США!

Нарушение порядка возникло, потому что ключи численные. Интерпретатор JavaScript видит, что строка на самом деле является числом и преобразует ключ в немного другой внутренний формат. Дополнительным эффектом внутренних оптимизаций является сортировка.

**А что, если мы хотим, чтобы порядок был именно таким, какой мы задали?**

Это возможно. Можно применить небольшой хак, который заключается в том, чтобы сделать все ключи нечисловыми, например, добавим в начало дополнительный символ '+':

var codes = {

"+7": "Россия",

"+38": "Украина",

"+1": "США"

};

for (var code in codes) {

var value = codes[code];

code = +code; // ..если нам нужно именно число, преобразуем: "+7" -> 7

alert( code + ": " + value ); // 7, 38, 1 во всех браузерах

}

**Итого**

* Цикл по ключам: for (key in obj).
* Порядок перебора соответствует порядку объявления для нечисловых ключей, а числовые – сортируются (в современных браузерах).
* Если нужно, чтобы порядок перебора числовых ключей соответствовал их объявлению в объекте, то используют трюк: числовые ключи заменяют на похожие, но содержащие не только цифры. Например, добавляют в начало +, как описано в примере выше, а потом, в процессе обработки, преобразуют такие ключи в числа.

#### Задачи

**Определите, пуст ли объект**

Создайте функцию isEmpty(obj), которая возвращает true, если в объекте нет свойств и false – если хоть одно свойство есть.

Работать должно так:

function isEmpty(obj) {

/\* ваш код \*/

}

var schedule = {};

alert( isEmpty(schedule) ); // true

schedule["8:30"] = "подъём";

alert( isEmpty(schedule) ); // false

Решение:

function isEmpty(obj) {

for (var key in obj) {

return false; // есть свойства в объекте;

}

return true; // нет свойств;

}

**Сумма свойств**

Есть объект salaries с зарплатами. Напишите код, который выведет сумму всех зарплат.

Если объект пустой, то результат должен быть 0.

Например:

"use strict";

var salaries = {

"Вася": 100,

"Петя": 300,

"Даша": 250

};

//... ваш код выведет 650

P.S. Сверху стоит use strict, чтобы не забыть объявить переменные.

"use strict";

var sum = 0;

for (var name in salaries) {

sum += salaries[name];

}

alert( sum );

**Свойство с наибольшим значением**

Есть объект salaries с зарплатами. Напишите код, который выведет имя сотрудника, у которого самая большая зарплата.

Если объект пустой, то пусть он выводит «нет сотрудников».

Например:

"use strict";

var salaries = {

"Вася": 100 ,

"Петя": 300,

"Даша": 250

};

// ... ваш код выведет "Петя"

var max = 0;

var maxName = "";

for (var name in salaries) {

if (max < salaries[name]) {

max = salaries[name];

maxName = name;

}

}

alert( maxName || "нет сотрудников" ); // если без сотрудников, то maxName="" => false и возвращается 2 вариант

**Умножьте численные свойства на 2**

Создайте функцию multiplyNumeric, которая получает объект и умножает все численные свойства на 2. Например:

// до вызова

var menu = {

width: 200,

height: 300,

title: "My menu"

};

multiplyNumeric(menu);

// после вызова

menu = {

width: 400,

height: 600,

title: "My menu"

};

var menu = {

width: 200,

height: 300,

title: "My menu"

};

function isNumeric(n) {

return !isNaN(parseFloat(n)) && isFinite(n);

}

function multiplyNumeric(obj) {

for (var key in obj) {

if (isNumeric(obj[key])) {

obj[key] \*= 2;

}

}

}

multiplyNumeric(menu);

alert( "menu width=" + menu.width + " height=" + menu.height + " title=" + menu.title );

### Объекты: передача по ссылке

Фундаментальным отличием объектов от примитивов, является их хранение и копирование «по ссылке».

#### Копирование по значению

Обычные значения: строки, числа, булевы значения, null/undefined при присваивании переменных копируются целиком или, как говорят, *«по значению»*.

var message = "Привет";

var phrase = message;

В результате такого копирования получились две полностью независимые переменные, в каждой из которых хранится значение "Привет".



#### Копирование по ссылке

С объектами – всё не так.

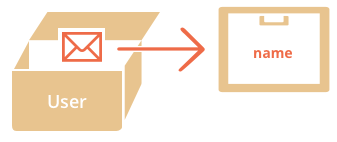
**В переменной, которой присвоен объект, хранится не сам объект, а «адрес его места в памяти», иными словами – «ссылка» на него.**

Вот как выглядит переменная, которой присвоен объект:

var user = {

name: "Вася"

};



Внимание: объект – вне переменной. В переменной – лишь «адрес» (ссылка) для него.

**При копировании переменной с объектом – копируется эта ссылка, а объект по-прежнему остается в единственном экземпляре.**

Например:

var user = { name: "Вася" }; // в переменной - ссылка

var admin = user; // скопировали ссылку

Получили две переменные, в которых находятся ссылки на один и тот же объект:



**Так как объект всего один, то изменения через любую переменную видны в других переменных:**

var user = { name: 'Вася' };

var admin = user;

admin.name = 'Петя'; // поменяли данные через admin

alert(user.name); // 'Петя', изменения видны в user

Переменная с объектом как «ключ» к сейфу с данными

Ещё одна аналогия: переменная, в которую присвоен объект, на самом деле хранит не сами данные, а ключ к сейфу, где они хранятся.

При копировании её, получается что мы сделали копию ключа, но сейф по-прежнему один.

#### Клонирование объектов

Иногда, на практике – очень редко, нужно скопировать объект целиком, создать именно полную независимую копию, «клон» объекта.

Что ж, можно сделать и это. Для этого нужно пройти по объекту, достать данные и скопировать на уровне примитивов.

Примерно так:

var user = {

name: "Вася",

age: 30

};

var clone = {}; // новый пустой объект

// скопируем в него все свойства user

for (var key in user) {

clone[key] = user[key];

}

// теперь clone - полностью независимая копия

clone.name = "Петя"; // поменяли данные в clone

alert( user.name ); // по-прежнему "Вася"

В этом коде каждое свойство объекта user копируется в clone. Если предположить, что они примитивны, то каждое скопируется по значению и мы как раз получим полный клон.

Если же свойства объектов, в свою очередь, могут хранить ссылки на другие объекты, то нужно обойти такие подобъекты и тоже склонировать их. Это называют «глубоким» клонированием.

#### Вывод в консоли

Откройте консоль браузера (обычно F12) и запустите следующий код:

var time = {

year: 2345,

month: 11,

day: 10,

hour: 11,

minute: 12,

second: 13,

microsecond: 123456

}

console.log(time); // (\*)

time.microsecond++; // (\*\*)

console.log(time);

time.microsecond++;

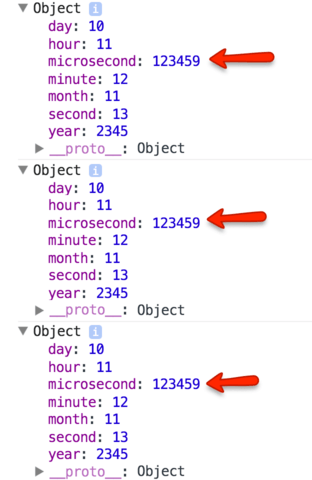
console.log(time);

time.microsecond++;

Как видно, в нём некий объект выводится строкой (\*), затем он меняется в строке (\*\*) и снова выводится, и так несколько раз. Пока ничего необычного, типичная ситуация – скрипт делает какую-то работу с объектом и выводит в консоли то, как она продвигается.

Необычное – в другом!

При раскрытии каждый объект будет выглядеть примерно так (скриншот из Chrome):



**Судя по выводу, свойство microsecond всегда было равно 123459… Или нет?**

Если посмотреть на код выше то, очевидно, нет! Это свойство меняется, а консоль нас просто дурит.

**При «раскрытии» свойств объекта в консоли – браузер всегда выводит их текущие (на момент раскрытия) значения.**

Так происходит именно потому, что вывод не делает «копию» текущего содержимого, а сохраняет лишь ссылку на объект. Запомните эту особенность консоли, в будущем, при отладке скриптов у вас не раз возникнет подобная ситуация.

**Итого**

* Объект присваивается и копируется «по ссылке». То есть, в переменной хранится не сам объект а, условно говоря, адрес в памяти, где он находится.
* Если переменная-объект скопирована или передана в функцию, то копируется именно эта ссылка, а объект остаётся один в памяти.

Это – одно из ключевых отличий объекта от примитива (числа, строки…), который при присвоении как раз копируется «по значению», то есть полностью.

### Методы объектов и контекст вызова

#### Методы объектов, this

До этого мы говорили об объекте лишь как о хранилище значений. Теперь пойдём дальше и поговорим об объектах как о сущностях со своими функциями («методами»).

##### Методы у объектов

При объявлении объекта можно указать свойство-функцию, например:

var user = {

name: 'Василий',

// метод

sayHi: function() {

alert( 'Привет!' );

}

};

// Вызов

user.sayHi();

Свойства-функции называют «методами» объектов. Их можно добавлять и удалять в любой момент, в том числе и явным присваиванием:

var user = {

name: 'Василий'

};

user.sayHi = function() { // присвоили метод после создания объекта

alert('Привет!');

};

// Вызов метода:

user.sayHi();

##### Доступ к объекту через this

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

**Для доступа к текущему объекту из метода используется ключевое слово this**.

Значением this является объект перед «точкой», в контексте которого вызван метод, например:

var user = {

name: 'Василий',

sayHi: function() {

alert( this.name );

}

};

user.sayHi(); // sayHi в контексте user

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

Вместо this внутри sayHi можно было бы обратиться к объекту, используя переменную user:

...

sayHi: function() {

alert( user.name );

}

...

…Однако, такое решение нестабильно. Если мы решим скопировать объект в другую переменную, например admin = user, а в переменную user записать что-то другое – обращение будет совсем не по адресу:

var user = {

name: 'Василий',

sayHi: function() {

alert( user.name ); // приведёт к ошибке

}

};

var admin = user;

user = null;

admin.sayHi(); // упс! внутри sayHi обращение по старому имени, ошибка!

Использование this гарантирует, что функция работает именно с тем объектом, в контексте которого вызвана.

Через this метод может не только обратиться к любому свойству объекта, но и передать куда-то ссылку на сам объект целиком:

var user = {

name: 'Василий',

sayHi: function() {

showName(this); // передать текущий объект в showName

}

};

function showName(namedObj) {

alert( namedObj.name );

}

user.sayHi(); // Василий

##### Подробнее про this

Любая функция может иметь в себе this. Совершенно неважно, объявлена ли она в объекте или отдельно от него.

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

Например, такая функция, объявленная без объекта, вполне допустима:

function sayHi() {

alert( this.firstName );

}

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

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

var user = { firstName: "Вася" };

var admin = { firstName: "Админ" };

function func() {

alert( this.firstName );

}

user.f = func;

admin.g = func;

// this равен объекту перед точкой:

user.f(); // Вася

admin.g(); // Админ

admin['g'](); // Админ (не важно, доступ к объекту через точку или квадратные скобки)

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

##### Значение this при вызове без контекста

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

Как правило, такая ситуация возникает при ошибке в разработке.

При этом this получает значение window, глобального объекта:

function func() {

alert( this ); // выведет [object Window] или [object global]

}

func();

Таково поведение в старом стандарте.

А в режиме use strict вместо глобального объекта this будет undefined:

function func() {

"use strict";

alert( this ); // выведет undefined (кроме IE9-)

}

func();

Обычно если в функции используется this, то она, всё же, служит для вызова в контексте объекта, так что такая ситуация – скорее исключение.

Ссылочный тип

Контекст this никак не привязан к функции, даже если она создана в объявлении объекта. Чтобы this передался, нужно вызвать функцию именно через точку (или квадратные скобки).

Любой более хитрый вызов приведёт к потере контекста, например:

var user = {

name: "Вася",

hi: function() { alert(this.name); },

bye: function() { alert("Пока"); }

};

user.hi(); // Вася (простой вызов работает)

// а теперь вызовем user.hi или user.bye в зависимости от имени

(user.name == "Вася" ? user.hi : user.bye)(); // undefined

В последней строке примера метод получен в результате выполнения тернарного оператора и тут же вызван. Но this при этом теряется.

Если хочется понять, почему, то причина кроется в деталях работы вызова obj.method().

Он ведь, на самом деле, состоит из двух независимых операций: точка . – получение свойства и скобки () – его вызов (предполагается, что это функция).

Функция, как мы говорили раньше, сама по себе не запоминает контекст. Чтобы «донести его» до скобок, JavaScript применяет «финт ушами» – точка возвращает не функцию, а значение специального «ссылочного» типа [Reference Type](https://people.mozilla.org/~jorendorff/es6-draft.html#sec-reference-specification-type).

Этот тип представляет собой связку «base-name-strict», где:

* *base* – как раз объект,
* *name* – имя свойства,
* *strict* – вспомогательный флаг для передачи use strict.

То есть, ссылочный тип (Reference Type) – это своеобразное «три-в-одном». Он существует исключительно для целей спецификации, мы его не видим, поскольку любой оператор тут же от него избавляется:

* Скобки () получают из base значение свойства name и вызывают в контексте base.
* Другие операторы получают из base значение свойства name и используют, а остальные компоненты игнорируют.

Поэтому любая операция над результатом операции получения свойства, кроме вызова, приводит к потере контекста.

Аналогично работает и получение свойства через квадратные скобки obj[method].

##### Задачи

**Вызов в контексте массива**

Каким будет результат? Почему?

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

arr.push(function() {

alert( this );

})

arr[2](); // ?

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

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

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

**Проверка синтаксиса**

Каков будет результат этого кода?

var obj = {

go: function() { alert(this) }

}

(obj.go)()

P.S. Есть подвох :)

**Ошибка**!

Попробуйте:

Причем сообщение об ошибке в большинстве браузеров не даёт понять, что на самом деле не так.

**Ошибка возникла из-за того, что после объявления obj пропущена точка с запятой.**

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

var obj = { go:... }(obj.go)()

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

Без ошибки: // [object Object]

**Почему this присваивается именно так?**

важность: 3

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

"use strict"

var obj, method;

obj = {

go: function() { alert(this); }

};

obj.go(); // (1) object

(obj.go)(); // (2) object

(method = obj.go)(); // (3) undefined

(obj.go || obj.stop)(); // (4) undefined

В чём дело? Объясните логику работы this.

1. Обычный вызов функции в контексте объекта.

2. То же самое, скобки ни на что не влияют.

3. Здесь не просто вызов obj.method(), а более сложный вызов вида (выражение).method(). Такой вызов работает, как если бы он был разбит на две строки:

f = obj.go; // сначала вычислить выражение

f(); // потом вызвать то, что получилось

При этом f() выполняется как обычная функция, без передачи this.

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

В спецификации это объясняется при помощи специального внутреннего типа [Reference Type](http://es5.github.com/x8.html#x8.7).

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

Сначала получить свойство obj.go.

Потом вызвать его как функцию.

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

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

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

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

**Значение this в объявлении объекта**

Что выведет alert в этом коде? Почему?

var user = {

firstName: "Василий",

export: this (\*)

};

alert( user.export.firstName );

Ответ: undefined.

Объявление объекта само по себе не влияет на this. Никаких функций, которые могли бы повлиять на контекст, здесь нет.

Так как код находится вообще вне любых функций, то this в нём равен window (в браузере так всегда для кода вне функций, вне зависимости от use strict).

Получается, что в строке (\*) мы имеем export: window, так что далее alert(user.export.firstName) выводит свойство window.firstName, которое не определено.

**Возврат this**

Что выведет alert в этом коде? Почему?

var name = "";

var user = {

name: "Василий",

export: function() {

return this;

}

};

alert( user.export().name );

Ответ: Василий.

Вызов user.export() использует this, который равен объекту до точки, то есть внутри user.export() строка return this возвращает объект user.

В итоге выводится свойство name объекта user, равное "Василий".

**Возврат объекта с this**

Что выведет alert в этом коде? Почему?

var name = "";

var user = {

name: "Василий",

export: function() {

return {

value: this

};

}

};

alert( user.export().value.name );

Ответ: Василий.

Во время выполнения user.export() значение this = user.

При создании объекта { value: this }, в свойство value копируется ссылка на текущий контекст, то есть на user.

Получается что user.export().value == user.

**Создайте калькулятор**

Создайте объект calculator с тремя методами:

* read() запрашивает prompt два значения и сохраняет их как свойства объекта
* sum() возвращает сумму этих двух значений
* mul() возвращает произведение этих двух значений

var calculator = {

...ваш код...

}

calculator.read();

alert( calculator.sum() );

alert( calculator.mul() );

[Запустить демо](http://learn.javascript.ru/object-methods)

var calculator = {

sum: function() {

return this.a + this.b;

},

mul: function() {

return this.a \* this.b;

},

read: function() {

this.a = +prompt('a?', 0);

this.b = +prompt('b?', 0);

}

}

calculator.read();

alert( calculator.sum() );

alert( calculator.mul() );

**Цепочка вызовов**

Есть объект «лестница» ladder:

var ladder = {

step: 0,

up: function() { // вверх по лестнице

this.step++;

},

down: function() { // вниз по лестнице

this.step--;

},

showStep: function() { // вывести текущую ступеньку

alert( this.step );

}

};

Сейчас, если нужно последовательно вызвать несколько методов объекта, это можно сделать так:

ladder.up();

ladder.up();

ladder.down();

ladder.showStep(); // 1

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

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

Как видно, такая запись содержит «меньше букв» и может быть более наглядной.

Такой подход называется «чейнинг» (chaining) и используется, например, во фреймворке jQuery.

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

var ladder = {

step: 0,

up: function() {

this.step++;

return this;

},

down: function() {

this.step--;

return this;

},

showStep: function() {

alert( this.step );

return this;

}

}

#### Преобразование объектов: toString и valueOf

Ранее, в главе [Преобразование типов для примитивов](http://learn.javascript.ru/types-conversion) мы рассматривали преобразование типов для примитивов. Теперь добавим в нашу картину мира объекты.

Бывают операции, при которых объект должен быть преобразован в примитив.

Например:

* Строковое преобразование – если объект выводится через alert(obj).
* Численное преобразование – при арифметических операциях, сравнении с примитивом.
* Логическое преобразование – при if(obj) и других логических операциях.

Рассмотрим эти преобразования по очереди.

##### Логическое преобразование

Проще всего – с логическим преобразованием.

**Любой объект в логическом контексте – true, даже если это пустой массив [] или объект {}.**

if ({} && []) {

alert( "Все объекты - true!" ); // alert сработает

}

##### Строковое преобразование

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

var user = {

firstName: 'Василий'

};

alert( user ); // [object Object]

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

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

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

var user = {

firstName: 'Василий',

toString: function() {

return 'Пользователь ' + this.firstName;

}

};

alert( user ); // Пользователь Василий

**Результатом toString может быть любой примитив**

Метод toString не обязан возвращать именно строку.

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

var obj = {

toString: function() {

return 123;

}

};

alert( obj ); // 123

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

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

alert( [1, 2] ); // toString для массивов выводит список элементов "1,2"

alert( new Date ); // toString для дат выводит дату в виде строки

alert( function() {} ); // toString для функции выводит её код

##### Численное преобразование

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

var room = {

number: 777,

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

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

};

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

delete room.valueOf; // valueOf удалён

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

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

**У большинства объектов нет valueOf**

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

Исключением является объект Date, который поддерживает оба типа преобразований:

alert( new Date() ); // toString: Дата в виде читаемой строки

alert( +new Date() ); // valueOf: кол-во миллисекунд, прошедших с 01.01.1970

**Детали спецификации**

Если посмотреть в стандарт, то в пункте [15.2.4.4](http://es5.github.com/x15.2.html#x15.2.4.4) говорится о том, что valueOf есть у любых объектов. Но он ничего не делает, просто возвращает сам объект (не-примитивное значение!), а потому игнорируется.

##### Две стадии преобразования

Итак, объект преобразован в примитив при помощи toString или valueOf.

Но на этом преобразования не обязательно заканчиваются. Вполне возможно, что в процессе вычислений этот примитив будет преобразован во что-то другое.

Например, рассмотрим применение к объекту операции ==:

var obj = {

valueOf: function() {

return 1;

}

};

alert( obj == true ); // true

Объект obj был сначала преобразован в примитив, используя численное преобразование, получилось 1 == true.

Далее, так как значения всё ещё разных типов, применяются правила преобразования примитивов, результат: true.

То же самое – при сложении с объектом при помощи +:

var obj = {

valueOf: function() {

return 1;

}

};

alert( obj + "test" ); // 1test

Или вот, для разности объектов:

var a = {

valueOf: function() {

return "1";

}

};

var b = {

valueOf: function() {

return "2";

}

};

alert( a + b ); // "12"

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

##### Исключение: Date

Объект Date, по историческим причинам, является исключением.

Бинарный оператор плюс + обычно использует числовое преобразование и метод valueOf. Как мы уже знаем, если подходящего valueOf нет (а его нет у большинства объектов), то используется toString, так что в итоге преобразование происходит к строке. Но если есть valueOf, то используется valueOf. Выше в примере как раз a + b это демонстрируют.

У объектов Date есть и valueOf – возвращает количество миллисекунд, и toString – возвращает строку с датой.

…Но оператор + для Date использует именно toString (хотя должен бы valueOf).

Это и есть исключение:

// бинарный плюс для даты toString, для остальных объектов valueOf

alert( new Date + "" ); // "строка даты"

Других подобных исключений нет.

**Как испугать Java-разработчика**

В языке Java (это не JavaScript, другой язык, здесь приведён для примера) логические значения можно создавать, используя синтаксис new Boolean(true/false), например new Boolean(true).

В JavaScript тоже есть подобная возможность, которая возвращает «объектную обёртку» для логического значения.

Эта возможность давно существует лишь для совместимости, она и не используется на практике, поскольку приводит к странным результатам. Некоторые из них могут сильно удивить человека, не привыкшего к JavaScript, например:

var value = new Boolean(false);

if (value) {

alert( true ); // сработает!

}

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

var value = new Boolean(false);

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

if (value) {

alert( true ); // ..но тогда почему выполняется alert в if ?!?

}

Дело в том, что new Boolean – это не примитивное значение, а объект. Поэтому в логическом контексте он преобразуется к true, в результате работает первый пример.

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

**В JavaScript вызовы new Boolean/String/Number не используются, а используются простые вызовы соответствующих функций, они преобразуют значение в примитив нужного типа, например Boolean(val) === !!val.**

**Итого**

* В логическом контексте объект – всегда true.
* При строковом преобразовании объекта используется его метод toString. Он должен возвращать примитивное значение, причём не обязательно именно строку.
* Для численного преобразования используется метод valueOf, который также может возвратить любое примитивное значение. У большинства объектов valueOf не работает (возвращает сам объект и потому игнорируется), при этом для численного преобразования используется toString.

Полный алгоритм преобразований есть в спецификации ECMAScript, смотрите пункты [11.8.5](http://es5.github.com/x11.html#x11.8.5), [11.9.3](http://es5.github.com/x11.html#x11.9.3), а также [9.1](http://es5.github.com/x9.html#x9.1) и [9.3](http://es5.github.com/x9.html#x9.3).

Заметим, для полноты картины, что некоторые тесты знаний в интернет предлагают вопросы типа:

{}[0] // чему равно?

{} + {} // а так?

Если вы запустите эти выражения в консоли, то результат может показаться странным. Подвох здесь в том, что если фигурные скобки {...} идут не в выражении, а в основном потоке кода, то JavaScript считает, что это не объект, а «блок кода» (как if, for, но без оператора, просто группировка команд вместе, используется редко).

Вот блок кода с командой:

{

alert("Блок")

}

А если команду изъять, то будет пустой блок {}, который ничего не делает. Два примера выше как раз содержат пустой блок в начале, который ничего не делает. Иначе говоря:

{}[0] // то же что и: [0]

{} + {} // то же что и: + {} // сейчас показывает "[object Object][object Object]"

То есть, такие вопросы – не на преобразование типов, а на понимание, что если { ... } находится вне выражений, то это не объект, а блок.

##### Задачи

**['x'] == 'x'**

Почему результат true ?

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

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

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

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

P.S. По той же причине верны равенства:

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

alert( [] == '' ); // true

**Преобразование**

Объявлен объект с toString и valueOf.

Какими будут результаты alert?

var foo = {

toString: function() {

return 'foo';

},

valueOf: function() {

return 2;

}

};

alert( foo ); // "foo"

alert( foo + 1 ); // 3

alert( foo + "3" ); // "23"

alert( foo + "art" ); // "2art"

**Первый alert(foo)**

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

**Второй alert(foo + 1)**

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

**Третий alert(foo + „3“)**

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

**Почему [] == [] неверно, а [ ] == ![ ] верно?**

Почему первое равенство – неверно, а второе – верно?

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

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

Какие преобразования происходят при вычислении?

**Ответ по первому равенству**

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

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

**Ответ по второму равенству**

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

alert( [] == false );

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

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

alert( '' == false );

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

alert( 0 == 0 );

Теперь результат очевиден.

**Вопросник по преобразованиям, для объектов**

new Date(0) - 0 = 0 // (1)

new Date(0) – дата, созданная по миллисекундам и соответствующая 0 мс от 1 января 1970 года 00:00:00 UTC. Оператор минус - преобразует дату обратно в число миллисекунд, то есть в 0.

new Array(1)[0] + "" = "undefined" // (2)

new Array(num) при вызове с единственным аргументом-числом создаёт массив данной длины, без элементов. Поэтому его нулевой элемент равен undefined, при сложении со строкой получается строка "undefined".

({})[0] = undefined // (3)

Фигурные скобки – это создание пустого объекта, у него нет свойства '0'. Так что значением будет undefined. Обратите внимание на внешние, круглые скобки. Если их убрать и запустить {}[0] в отладочной консоли браузера – будет [0], т.к. скобки {} будут восприняты как пустой блок кода, после которого идёт массив.

[1] + 1 = "11" // (4)

Массив преобразуется в строку "1". Оператор "+" при сложении со строкой приводит второй аргумент к строке – значит будет "1" + "1" = "11".

[1,2] + [3,4] = "1,23,4" // (5)

Массивы приводятся к строке и складываются.

[] + null + 1 = "null1" // (6)

Массив преобразуется в пустую строку "" + null + 1, оператор "+" видит, что слева строка и преобразует null к строке, получается "null" + 1, и в итоге "null1". [5] + null + 1 // "5null1"

[[0]][0][0] = 0 // (7)

[[0]] – это вложенный массив [0] внутри внешнего [ ]. Затем мы берём от него нулевой элемент, и потом еще раз.

Если это непонятно, то посмотрите на такой пример:

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

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

({} + {}) = "[object Object][object Object]" // (8)

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

**Сумма произвольного количества скобок**

Напишите функцию sum, которая будет работать так:

sum(1)(2) == 3; // 1 + 2

sum(1)(2)(3) == 6; // 1 + 2 + 3

sum(5)(-1)(2) == 6

sum(6)(-1)(-2)(-3) == 0

sum(0)(1)(2)(3)(4)(5) == 15

Количество скобок может быть любым.

Пример такой функции для двух аргументов – есть в решении задачи [Сумма через замыкание](http://learn.javascript.ru/task/closure-sum).

**Подсказка**

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

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

**Решение**

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

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

function sum(a) {

var currentSum = a;

function f(b) {

currentSum += b;

return f;

}

f.toString = function() {

return currentSum;

};

return f;

}

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

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

**В последней строчке f нет рекурсивного вызова.**

Вот так была бы рекурсия:

function f(b) {

currentSum += b;

return f(); // <-- подвызов

}

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

function f(b) {

currentSum += b;

return f; // <-- не вызывает сама себя, а возвращает ссылку на себя

}

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

#### Создание объектов через "new"

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

Для этого используют «функции-конструкторы», запуская их при помощи специального оператора new.

##### Конструктор

Конструктором становится любая функция, вызванная через new.

Например:

function **A**nimal(name) {

this.name = name;

this.canWalk = true;

}

var animal = new Animal("ёжик");

Заметим, что, технически, любая функция может быть использована как конструктор. То есть, любую функцию можно вызвать при помощи new. Как-то особым образом указывать, что она – конструктор – не надо.

Но, чтобы выделить функции, задуманные как конструкторы, их называют с большой буквы: Animal, а не animal.

Детальнее – функция, запущенная через new, делает следующее:

1. Создаётся новый пустой объект.
2. Ключевое слово this получает ссылку на этот объект.
3. Функция выполняется. Как правило, она модифицирует this (т.е. этот новый объект), добавляет методы, свойства.
4. Возвращается this.

В результате вызова new Animal("ёжик"); получаем такой объект:

animal = {

name: "ёжик",

canWalk: true

}

Иными словами, при вызове new Animal происходит что-то в таком духе (первая и последняя строка – это то, что делает интерпретатор):

function Animal(name) {

// this = {};

// в this пишем свойства, методы

this.name = name;

this.canWalk = true;

// return this;

}

Теперь многократными вызовами new Animal с разными параметрами мы можем создать столько объектов, сколько нужно. Поэтому такую функцию и называют *конструктором* – она предназначена для «конструирования» объектов.

new function() { … }

Иногда функцию-конструктор объявляют и тут же используют, вот так:

var animal = new function() {

this.name = "Васька";

this.canWalk = true;

};

Так делают, когда хотят создать единственный объект данного типа. Пример выше с тем же успехом можно было бы переписать как:

var animal = {

name: "Васька",

canWalk: true

}

…Но обычный синтаксис {...} не подходит, когда при создании свойств объекта нужны более сложные вычисления. Их можно проделать в функции-конструкторе и записать результат в this.

##### Правила обработки return

Как правило, конструкторы ничего не возвращают. Их задача – записать всё, что нужно, в this, который автоматически станет результатом.

Но если явный вызов return всё же есть, то применяется простое правило:

* При вызове return с объектом, будет возвращён он, а не this.
* При вызове return с примитивным значением, оно будет отброшено.

Иными словами, вызов return с объектом вернёт объект, а с чем угодно, кроме объекта – возвратит, как обычно, this.

Например, возврат объекта:

function BigAnimal() {

this.name = "Мышь";

return { name: "Годзилла" }; // <-- возвратим объект

}

alert( new BigAnimal().name ); // Годзилла, получили объект вместо this

А вот пример с возвратом строки:

function BigAnimal() {

this.name = "Мышь";

return "Годзилла"; // <-- возвратим примитив

}

alert( new BigAnimal().name ); // Мышь, получили this (а Годзилла пропал)

Эта особенность работы new прописана в стандарте, но используется она весьма редко.

Можно без скобок

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

var animal = new BigAnimal; // <-- без скобок

// то же самое что

var animal = new BigAnimal();

Не сказать, что выбрасывание скобок – «хороший стиль», но такой синтаксис допустим стандартом.

##### Создание методов в конструкторе

Использование функций для создания объекта дает большую гибкость. Можно передавать конструктору параметры, определяющие как его создавать, и он будет «клепать» объекты заданным образом.

Добавим в создаваемый объект ещё и метод.

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

function User(name) {

this.name = name;

this.sayHi = function() {

alert( "Моё имя: " + this.name );

};

}

var ivan = new User("Иван");

ivan.sayHi(); // Моё имя: Иван

/\*

ivan = {

name: "Иван",

sayHi: функция

}

\*/

##### Локальные переменные

В функции-конструкторе бывает удобно объявить вспомогательные локальные переменные и вложенные функции, которые будут видны только внутри:

function User(firstName, lastName) {

// вспомогательная переменная

var phrase = "Привет";

// вспомогательная вложенная функция

function getFullName() {

return firstName + " " + lastName;

}

this.sayHi = function() {

alert( phrase + ", " + getFullName() ); // использование

};

}

var vasya = new User("Вася", "Петров");

vasya.sayHi(); // Привет, Вася Петров

Мы уже говорили об этом подходе ранее, в главе [Локальные переменные для объекта](http://learn.javascript.ru/closures-usage).

Те функции и данные, которые должны быть доступны для внешнего кода, мы пишем в this – и к ним можно будет обращаться, как например vasya.sayHi(), а вспомогательные, которые нужны только внутри самого объекта, сохраняем в локальной области видимости.

**Итого**

Объекты могут быть созданы при помощи функций-конструкторов:

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

##### Задачи

**Две функции один объект**

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

function A() { ... }

function B() { ... }

var a = new A;

var b = new B;

alert( a == b ); // true

Если да – приведите пример кода с такими функциями.

Да, возможны.

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

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

var obj = {};

function A() { return obj; }

function B() { return obj; }

var a = new A();

var b = new B();

alert( a == b ); // true

**Создать Calculator при помощи конструктора**

Напишите функцию-конструктор Calculator, которая создает объект с тремя методами:

* Метод read() запрашивает два значения при помощи prompt и запоминает их в свойствах объекта.
* Метод sum() возвращает сумму запомненных свойств.
* Метод mul() возвращает произведение запомненных свойств.

Пример использования:

var calculator = new Calculator();

calculator.read();

alert( "Сумма=" + calculator.sum() );

alert( "Произведение=" + calculator.mul() );

function Calculator() {

this.read = function() {

this.a = +prompt('a?', 0);

this.b = +prompt('b?', 0);

};

this.sum = function() {

return this.a + this.b;

};

this.mul = function() {

return this.a \* this.b;

};

}

var calculator = new Calculator();

calculator.read();

alert( "Сумма=" + calculator.sum() );

alert( "Произведение=" + calculator.mul() );

**Создать Accumulator при помощи конструктора**

Напишите *функцию-конструктор* Accumulator(startingValue). Объекты, которые она создает, должны хранить текущую сумму и прибавлять к ней то, что вводит посетитель.

Более формально, объект должен:

* Хранить текущее значение в своём свойстве value. Начальное значение свойства value ставится конструктором равным startingValue.
* Метод read() вызывает prompt, принимает число и прибавляет его к свойству value.

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

Ниже вы можете посмотреть работу кода:

var accumulator = new Accumulator(1); // начальное значение 1

accumulator.read(); // прибавит ввод prompt к текущему значению

accumulator.read(); // прибавит ввод prompt к текущему значению

alert( accumulator.value ); // выведет текущее значение

function Accumulator(startingValue) {

this.value = startingValue;

this.read = function() {

this.value += +prompt('Сколько добавлять будем?', 0);

};

}

var accumulator = new Accumulator(1);

accumulator.read();

accumulator.read();

alert( accumulator.value );

**Создайте калькулятор**

Напишите конструктор Calculator, который создаёт расширяемые объекты-калькуляторы.

Эта задача состоит из двух частей, которые можно решать одна за другой.

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

Пример использования:

var calc = new Calculator;

alert( calc.calculate("3 + 7") ); // 10

Второй шаг – добавить калькулятору метод addMethod(name, func), который учит калькулятор новой операции. Он получает имя операции name и функцию от двух аргументов func(a,b), которая должна её реализовывать.

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

var powerCalc = new Calculator;

powerCalc.addMethod("\*", function(a, b) {

return a \* b;

});

powerCalc.addMethod("/", function(a, b) {

return a / b;

});

powerCalc.addMethod("\*\*", function(a, b) {

return Math.pow(a, b);

});

var result = powerCalc.calculate("2 \*\* 3");

alert( result ); // 8

Поддержка скобок и сложных математических выражений в этой задаче не требуется.

Числа и операции могут состоять из нескольких символов. Между ними ровно один пробел.

Предусмотрите обработку ошибок. Какая она должна быть – решите сами.

1-й шаг;

function Calculator() {

this.calculate = function(str) {

var arr = str.split(' ');

if(arr[1] === '+') {

return +arr[0] + +arr[2];

} else {

return arr[0] - arr[2];

}

};

}

2-й шаг:

function Calculator() {

var methods = {};

this.addMethod = function(key, val) {

if (key in methods) {

return ( "Операция \'" + key + "\' существует!" );

}

return methods[key] = val;

};

this.calculate = function(str) {

var arr = str.split(' '),

num1 = arr[0],

num2 = arr[2],

operator = arr[1];

if(methods[operator]) {

return methods[operator](num1, num2);

}

return 'Данной операции не существует';

};

}

Решение из учебника:

function Calculator() {

var methods = {

"-": function(a, b) {

return a - b;

},

"+": function(a, b) {

return a + b;

}

};

this.calculate = function(str) {

var split = str.split(' '),

a = +split[0],

op = split[1],

b = +split[2]

if (!methods[op] || isNaN(a) || isNaN(b)) {

return NaN;

}

return methods[op](a, b);

}

this.addMethod = function(name, func) {

methods[name] = func;

};

}

#### Дескрипторы, геттеры и сеттеры свойств

В этой главе мы рассмотрим возможности, которые позволяют очень гибко и мощно управлять всеми свойствами объекта, включая их аспекты – изменяемость, видимость в цикле for..in и даже незаметно делать их функциями.

Они поддерживаются всеми современными браузерами, но не IE8-. Впрочем, даже IE8 их поддерживает, но только для DOM-объектов (используются при работе со страницей, это сейчас вне нашего рассмотрения).

##### Дескрипторы в примерах

Основной метод для управления свойствами – Object.defineProperty.

Он позволяет объявить свойство объекта и, что самое главное, тонко настроить его особые аспекты, которые никак иначе не изменить.

Синтаксис:

Object.defineProperty(obj, prop, descriptor)

Аргументы:

obj

Объект, в котором объявляется свойство.

prop

Имя свойства, которое нужно объявить или модифицировать.

descriptor

Дескриптор – объект, который описывает поведение свойства.

В нём могут быть следующие поля:

* value – значение свойства, по умолчанию undefined
* writable – значение свойства можно менять, если true. По умолчанию false.
* configurable – если true, то свойство можно удалять, а также менять его в дальнейшем при помощи новых вызовов defineProperty. По умолчанию false.
* enumerable – если true, то свойство просматривается в цикле for..in и методе Object.keys(). По умолчанию false.
* get – функция, которая возвращает значение свойства. По умолчанию undefined.
* set – функция, которая записывает значение свойства. По умолчанию undefined.

Чтобы избежать конфликта, запрещено одновременно указывать значение value и функции get/set. Либо значение, либо функции для его чтения-записи, одно из двух. Также запрещено и не имеет смысла указывать writable при наличии get/set-функций.

Далее мы подробно разберём эти свойства на примерах.

##### Обычное свойство

Два таких вызова работают одинаково:

var user = {};

// 1. простое присваивание

user.name = "Вася";

// 2. указание значения через дескриптор

Object.defineProperty(user, "name", { value: "Вася", configurable: true, writable: true, enumerable: true });

Оба вызова выше добавляют в объект user обычное (удаляемое, изменяемое, перечисляемое) свойство.

##### **Свойство-константа**

Для того, чтобы сделать свойство неизменяемым, изменим его флаги writable и configurable:

"use strict";

var user = {};

Object.defineProperty(user, "name", {

value: "Вася",

writable: false, // запретить присвоение "user.name="

configurable: false // запретить удаление "delete user.name"

});

// Теперь попытаемся изменить это свойство.

// в strict mode присвоение "user.name=" вызовет ошибку

user.name = "Петя";

Заметим, что без use strict операция записи «молча» не сработает. Лишь если установлен режим use strict, то дополнительно сгенерируется ошибка.

##### Свойство, скрытое для for…in

Встроенный метод toString, как и большинство встроенных методов, не участвует в цикле for..in. Это удобно, так как обычно такое свойство является «служебным».

К сожалению, свойство toString, объявленное обычным способом, будет видно в цикле for..in, например:

var user = {

name: "Вася",

toString: function() { return this.name; }

};

for(var key in user) alert(key); // name, toString

Мы бы хотели, чтобы поведение нашего метода toString было таким же, как и стандартного.

Object.defineProperty может исключить toString из списка итерации, поставив ему флаг enumerable: false. По стандарту, у встроенного toString этот флаг уже стоит.

var user = {

name: "Вася",

toString: function() { return this.name; }

};

// помечаем toString как не подлежащий перебору в for..in

Object.defineProperty(user, "toString", {enumerable: false});

for(var key in user) alert(key); // name

Обратим внимание, вызов defineProperty не перезаписал свойство, а просто модифицировал настройки у существующего toString.

##### Свойство-функция

Дескриптор позволяет задать свойство, которое на самом деле работает как функция. Для этого в нём нужно указать эту функцию в get.

Например, у объекта user есть обычные свойства: имя firstName и фамилия surname.

Создадим свойство fullName, которое на самом деле является функцией:

var user = {

firstName: "Вася",

surname: "Петров"

}

Object.defineProperty(user, "fullName", {

get: function() {

return this.firstName + ' ' + this.surname;

}

});

alert(user.fullName); // Вася Петров

Обратим внимание, снаружи fullName – это обычное свойство user.fullName. Но дескриптор указывает, что на самом деле его значение возвращается функцией.

Также можно указать функцию, которая используется для записи значения, при помощи дескриптора set.

Например, добавим возможность присвоения user.fullName к примеру выше:

var user = {

firstName: "Вася",

surname: "Петров"

}

Object.defineProperty(user, "fullName", {

get: function() {

return this.firstName + ' ' + this.surname;

},

set: function(value) {

var split = value.split(' ');

this.firstName = split[0];

this.surname = split[1];

}

});

user.fullName = "Петя Иванов";

alert( user.firstName ); // Петя

alert( user.surname ); // Иванов

##### Указание get/set в литералах

Если мы создаём объект при помощи синтаксиса { ... }, то задать свойства-функции можно прямо в его определении.

Для этого используется особый синтаксис: get свойство или set свойство.

Например, ниже объявлен геттер-сеттер fullName:

var user = {

firstName: "Вася",

surname: "Петров",

get fullName() {

return this.firstName + ' ' + this.surname;

},

set fullName(value) {

var split = value.split(' ');

this.firstName = split[0];

this.surname = split[1];

}

};

alert( user.fullName ); // Вася Петров (из геттера)

user.fullName = "Петя Иванов";

alert( user.firstName ); // Петя (поставил сеттер)

alert( user.surname ); // Иванов (поставил сеттер)

##### Да здравствуют get/set!

Казалось бы, зачем нам назначать get/set для свойства через всякие хитрые вызовы, когда можно сделать просто функции с самого начала? Например, getFullName, setFullName…

Конечно, в ряде случаев свойства выглядят короче, такое решение просто может быть красивым. Но основной бонус – это гибкость, возможность получить контроль над свойством в любой момент!

Например, в начале разработки мы используем обычные свойства, например у User будет имя name и возраст age:

function User(name, age) {

this.name = name;

this.age = age;

}

var pete = new User("Петя", 25);

alert( pete.age ); // 25

С обычными свойствами в коде меньше букв, они удобны, причины использовать функции пока нет.

…Но рано или поздно могут произойти изменения. Например, в User может стать более целесообразно вместо возраста age хранить дату рождения birthday:

function User(name, birthday) {

this.name = name;

this.birthday = birthday;

}

var pete = new User("Петя", new Date(1987, 6, 1));

Что теперь делать со старым кодом, который выводит свойство age?

Можно, конечно, найти все места и поправить их, но это долго, а иногда и невозможно, скажем, если вы взаимодействуете со сторонней библиотекой, код в которой – чужой и влезать в него нежелательно.

Добавление get-функции age позволяет обойти проблему легко и непринуждённо:

function User(name, birthday) {

this.name = name;

this.birthday = birthday;

// age будет высчитывать возраст по birthday

Object.defineProperty(this, "age", {

get: function() {

var todayYear = new Date().getFullYear(); //2017

return todayYear - this.birthday.getFullYear();

}

});

}

var pete = new User("Петя", new Date(1987, 6, 1));

alert( pete.birthday ); // и дата рождения доступна

alert( pete.age ); // и возраст

Заметим, что pete.age снаружи как было свойством, так и осталось. То есть, переписывать внешний код на вызов функции pete.age() не нужно.

Таким образом, defineProperty позволяет нам начать с обычных свойств, а в будущем, при необходимости, можно в любой момент заменить их на функции, реализующие более сложную логику.

##### Другие методы работы со свойствами

Object.defineProperties(obj, descriptors)

Позволяет объявить несколько свойств сразу:

var user = {}

Object.defineProperties(user, {

firstName: {

value: "Петя"

},

surname: {

value: "Иванов"

},

fullName: {

get: function() {

return this.firstName + ' ' + this.surname;

}

}

});

alert( user.fullName ); // Петя Иванов

Object.keys(obj),  Object.getOwnPropertyNames(obj)

Возвращают массив – список свойств объекта.

Object.keys возвращает только enumerable-свойства.

Object.getOwnPropertyNames – возвращает все:

var obj = {

a: 1,

b: 2,

internal: 3

};

Object.defineProperty(obj, "internal", {

enumerable: false

});

alert( Object.keys(obj) ); // a,b

alert( Object.getOwnPropertyNames(obj) ); // a, internal, b

Object.getOwnPropertyDescriptor(obj, prop)

Возвращает дескриптор для свойства obj[prop].

Полученный дескриптор можно изменить и использовать defineProperty для сохранения изменений, например:

var obj = {

test: 5

};

var descriptor = Object.getOwnPropertyDescriptor(obj, 'test');

// заменим value на геттер, для этого...

delete descriptor.value; // ..нужно убрать value/writable

delete descriptor.writable;

descriptor.get = function() { // и поставить get

alert( "Preved :)" );

};

// поставим новое свойство вместо старого

// если не удалить - defineProperty объединит старый дескриптор с новым

delete obj.test;

Object.defineProperty(obj, 'test', descriptor);

obj.test; // Preved :)

…И несколько методов, которые используются очень редко:

Object.preventExtensions(obj)

Запрещает добавление свойств в объект.

Object.seal(obj)

Запрещает добавление и удаление свойств, все текущие свойства делает configurable: false.

Object.freeze(obj)

Запрещает добавление, удаление и изменение свойств, все текущие свойства делает configurable: false, writable: false.

Object.isExtensible(obj)

Возвращает false, если добавление свойств объекта было запрещено вызовом метода Object.preventExtensions.

Object.isSealed(obj)

Возвращает true, если добавление и удаление свойств объекта запрещено, и все текущие свойства являются configurable: false.

Object.isFrozen(obj)

Возвращает true, если добавление, удаление и изменение свойств объекта запрещено, и все текущие свойства являются configurable: false, writable: false.

##### Задачи

**Добавить get/set-свойства**

Вам попал в руки код объекта User, который хранит имя и фамилию в свойстве this.fullName:

function User(fullName) {

this.fullName = fullName;

}

var vasya = new User("Василий Попкин");

Имя и фамилия всегда разделяются пробелом.

Сделайте, чтобы были доступны свойства firstName и lastName, причём не только на чтение, но и на запись, вот так:

var vasya = new User("Василий Попкин");

// чтение firstName/lastName

alert( vasya.firstName ); // Василий

alert( vasya.lastName ); // Попкин

// запись в lastName

vasya.lastName = 'Сидоров';

alert( vasya.fullName ); // Василий Сидоров

Важно: в этой задаче fullName должно остаться свойством, а firstName/lastName – реализованы через get/set. Лишнее дублирование здесь ни к чему.

function User(fullName) {

this.fullName = fullName;

Object.defineProperties(this, {

firstName: {

get: function() {

return this.fullName.split(' ')[0];

},

set: function(newFirstName) {

this.fullName = newFirstName + ' ' + this.lastName;

}

},

lastName: {

get: function() {

return this.fullName.split(' ')[1];

},

set: function(newLastName) {

this.fullName = this.firstName + ' ' + newLastName;

}

}

});

}

var vasya = new User("Василий Попкин");

// чтение firstName/lastName

alert( vasya.firstName ); // Василий

alert( vasya.lastName ); // Попкин

// запись в lastName

vasya.lastName = 'Сидоров';

alert( vasya.fullName ); // Василий Сидоров

#### Статические и фабричные методы

Методы и свойства, которые не привязаны к конкретному экземпляру объекта, называют «статическими». Их записывают прямо в саму функцию-конструктор.

##### Статические свойства

В коде ниже используются статические свойства Article.count и Article.DEFAULT\_FORMAT:

function Article() {

Article.count++;

}

Article.count = 0; // статическое свойство-переменная

Article.DEFAULT\_FORMAT = "html"; // статическое свойство-константа

Они хранят данные, специфичные не для одного объекта, а для всех статей целиком.

Как правило, это чаще константы, такие как формат «по умолчанию» Article.DEFAULT\_FORMAT.

##### Статические методы

С примерами статических методов мы уже знакомы: это встроенные методы [String.fromCharCode](http://javascript.ru/String.fromCharCode), [Date.parse](http://javascript.ru/Date.parse).

Создадим для Article статический метод Article.showCount():

function Article() {

Article.count++;

//...

}

Article.count = 0;

Article.showCount = function() {

alert( this.count ); // (1)

}

// использование

new Article();

new Article();

Article.showCount(); // (2)

Здесь Article.count – статическое свойство, а Article.showCount – статический метод.

Обратим внимание на использование this в примере выше. Несмотря на то, что переменная и метод – статические, он всё ещё полезен. В строке (1) он равен Article.

**Пример: сравнение объектов**

Ещё один хороший способ применения – сравнение объектов.

Например, у нас есть объект Journal для журналов. Журналы можно сравнивать – по толщине, по весу, по другим параметрам.

Объявим «стандартную» функцию сравнения, которая будет сравнивать по дате издания. Эта функция сравнения, естественно, не привязана к конкретному журналу, но относится к журналам вообще.

Поэтому зададим её как статический метод Journal.compare:

function Journal(date) {

this.date = date;

// ...

}

// возвращает значение, большее 0, если A больше B, иначе меньшее 0

Journal.compare = function(journalA, journalB) {

return journalA.date - journalB.date;

};

В примере ниже эта функция используется для поиска самого раннего журнала из массива:

function Journal(date) {

this.date = date;

this.formatDate = function(date) {

return date.getDate() + '.' + (date.getMonth() + 1) + '.' + date.getFullYear();

};

this.getTitle = function() {

return "Выпуск от " + this.formatDate(this.date);

};

}

Journal.compare = function(journalA, journalB) {

return journalA.date - journalB.date;

};

// использование:

var journals = [

new Journal(new Date(2012, 1, 1)),

new Journal(new Date(2012, 0, 1)),

new Journal(new Date(2011, 11, 1))

];

function findMin(journals) {

var min = 0;

for (var i = 0; i < journals.length; i++) {

// используем статический метод

if (Journal.compare(journals[min], journals[i]) > 0) min = i;

}

return journals[min];

}

alert( findMin(journals).getTitle() );

**Статический метод также можно использовать для функций, которые вообще не требуют наличия объекта.**

Например, метод formatDate(date) можно сделать статическим. Он будет форматировать дату «как это принято в журналах», при этом его можно использовать в любом месте кода, не обязательно создавать журнал.

Например:

function Journal() { /\*...\*/ }

Journal.formatDate = function(date) {

return date.getDate() + '.' + (date.getMonth()+1) + '.' + date.getFullYear();

}

// ни одного объекта Journal нет, просто форматируем дату

alert( Journal.formatDate(new Date) );

##### Фабричные методы

Рассмотрим ситуацию, когда объект нужно создавать различными способами. Например, это реализовано во встроенном объекте [Date](http://learn.javascript.ru/datetime). Он по-разному обрабатывает аргументы разных типов:

* new Date() – создаёт объект с текущей датой,
* new Date(milliseconds) – создаёт дату по количеству миллисекунд milliseconds,
* new Date(year, month, day ...) – создаёт дату по компонентам год, месяц, день…
* new Date(datestring) – читает дату из строки datestring

**«Фабричный статический метод» – удобная альтернатива такому конструктору. Так называется статический метод, который служит для создания новых объектов (поэтому и называется «фабричным»).**

Пример встроенного фабричного метода – [String.fromCharCode(code)](http://javascript.ru/String.fromCharCode). Этот метод создает строку из кода символа:

var str = String.fromCharCode(65);

alert( str ); // 'A'

Но строки – слишком простой пример, посмотрим что-нибудь посложнее.

Допустим, нам нужно создавать объекты User: анонимные new User() и с данными new User({name: 'Вася', age: 25}).

Можно, конечно, создать полиморфную функцию-конструктор User:

function User(userData) {

if (userData) { // если указаны данные -- одна ветка if

this.name = userData.name;

this.age = userData.age;

} else { // если не указаны -- другая

this.name = 'Аноним';

}

this.sayHi = function() {

alert(this.name)

};

// ...

}

// Использование

var guest = new User();

guest.sayHi(); // Аноним

var knownUser = new User({

name: 'Вася',

age: 25

});

knownUser.sayHi(); // Вася

Подход с использованием фабричных методов был бы другим. Вместо разбора параметров в конструкторе – делаем два метода: User.createAnonymous и User.createFromData.

Код:

function User() {

this.sayHi = function() {

alert(this.name)

};

}

User.createAnonymous = function() {

var user = new User;

user.name = 'Аноним';

return user;

}

User.createFromData = function(userData) {

var user = new User;

user.name = userData.name;

user.age = userData.age;

return user;

}

// Использование

var guest = User.createAnonymous();

guest.sayHi(); // Аноним

var knownUser = User.createFromData({

name: 'Вася',

age: 25

});

knownUser.sayHi(); // Вася

Преимущества использования фабричных методов:

* Лучшая читаемость кода. Как конструктора – вместо одной большой функции несколько маленьких, так и вызывающего кода – явно видно, что именно создаётся.
* Лучший контроль ошибок, т.к. если в createFromData ничего не передали, то будет ошибка, а полиморфный конструктор создал бы анонимного посетителя.
* Удобная расширяемость. Например, нужно добавить создание администратора, без аргументов. Фабричный метод сделать легко: User.createAdmin = function() { ... }. А для полиморфного конструктора вызов без аргумента создаст анонима, так что нужно добавить параметр – «тип посетителя» и усложнить этим код.

**Поэтому полиморфные конструкторы лучше использовать там, где нужен именно полиморфизм**, т.е. когда непонятно, какого типа аргумент передадут, и хочется в одном конструкторе охватить все варианты.

А в остальных случаях отличная альтернатива – фабричные методы.

[**Итого**](http://learn.javascript.ru/static-properties-and-methods#итого)

Статические свойства и методы объекта удобно применять в следующих случаях:

* Общие действия и подсчёты, имеющие отношения ко всем объектам данного типа. В примерах выше это подсчёт количества.
* Методы, не привязанные к конкретному объекту, например сравнение.
* Вспомогательные методы, которые полезны вне объекта, например для форматирования даты.
* Фабричные методы.

##### Задачи

**Счетчик объектов**

Добавить в конструктор Article:

* Подсчёт общего количества созданных объектов.
* Запоминание даты последнего созданного объекта.

Используйте для этого статические свойства.

Пусть вызов Article.showStats() выводит то и другое.

Использование:

function Article() {

this.created = new Date();

// ... ваш код ...

}

new Article();

new Article();

Article.showStats(); // Всего: 2, Последняя: (дата)

new Article();

Article.showStats(); // Всего: 3, Последняя: (дата)

function Article() {

this.created = new Date();

Article.count++; // увеличиваем счетчик при каждом вызове

Article.last = this.created; // и запоминаем дату

}

Article.count = 0; // начальное значение

// (нельзя оставить undefined, т.к. Article.count++ будет NaN)

Article.showStats = function() {

alert( 'Всего: ' + this.count + ', Последняя: ' + this.last );

};

#### Явное указание this: "call", "apply"

Итак, мы знаем, что this – это текущий объект при вызове «через точку» и новый объект при конструировании через new.

В этой главе наша цель получить окончательное и полное понимание this в JavaScript. Для этого не хватает всего одного элемента: способа явно указать this при помощи методов call и apply.

##### Метод call

Синтаксис метода call:

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

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

**Вызов func.call(context, a, b...) – то же, что обычный вызов func(a, b...), но с явно указанным this(=context).**

Например, у нас есть функция showFullName, которая работает с this:

function showFullName() {

alert( this.firstName + " " + this.lastName );

}

Пока объекта нет, но это нормально, ведь JavaScript позволяет использовать this везде. Любая функция может в своём коде упомянуть this, каким будет это значение – выяснится в момент запуска.

Вызов showFullName.call(user) запустит функцию, установив this = user, вот так:

function showFullName() {

alert( this.firstName + " " + this.lastName );

}

var user = {

firstName: "Василий",

lastName: "Петров"

};

// функция вызовется с this=user

showFullName.call(user) // "Василий Петров"

После контекста в call можно передать аргументы для функции. Вот пример с более сложным вариантом showFullName, который конструирует ответ из указанных свойств объекта:

var user = {

firstName: "Василий",

surname: "Петров",

patronym: "Иванович"

};

function showFullName(firstPart, lastPart) {

alert( this[firstPart] + " " + this[lastPart] );

}

// f.call(контекст, аргумент1, аргумент2, ...)

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

showFullName.call(user, 'firstName', 'patronym') // "Василий Иванович"

##### «Одалживание метода»

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

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

**Используем эту технику для упрощения манипуляций с arguments.**

Как мы знаем, arguments не массив, а обычный объект, поэтому таких полезных методов как push, pop, join и других у него нет. Но иногда так хочется, чтобы были…

Нет ничего проще! Давайте скопируем метод join из обычного массива:

function printArgs() {

arguments.join = [].join; // одолжили метод (1)

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

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

}

printArgs(1, 2, 3);

1. В строке (1) объявлен пустой массив [] и скопирован его метод [].join. Обратим внимание, мы не вызываем его, а просто копируем. Функция, в том числе встроенная – обычное значение, мы можем скопировать любое свойство любого объекта, и [].join здесь не исключение.
2. В строке (2) запустили join в контексте arguments, как будто он всегда там был.

**Почему вызов сработает?**

Здесь метод [join](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/join) массива скопирован и вызван в контексте arguments. Не произойдёт ли что-то плохое от того, что arguments – не массив? Почему он, вообще, сработал?

Ответ на эти вопросы простой. В соответствии [со спецификацией](http://es5.github.com/x15.4.html#x15.4.4.5), внутри join реализован примерно так:

function join(separator) {

if (!this.length) return '';

var str = this[0];

for (var i = 1; i < this.length; i++) {

str += separator + this[i];

}

return str;

}

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

В качестве this подойдёт даже обычный объект:

var obj = { // обычный объект с числовыми индексами и length

0: "А",

1: "Б",

2: "В",

length: 3

};

obj.join = [].join;

alert( obj.join(';') ); // "A;Б;В"

…Однако, копирование метода из одного объекта в другой не всегда приемлемо!

Представим на минуту, что вместо arguments у нас – произвольный объект. У него тоже есть числовые индексы, length и мы хотим вызвать в его контексте метод [].join. То есть, ситуация похожа на arguments, но (!) вполне возможно, что у объекта есть *свой* метод join.

Поэтому копировать [].join, как сделано выше, нельзя: если он перезапишет собственный join объекта, то будет страшный бардак и путаница.

Безопасно вызвать метод нам поможет call:

function printArgs() {

var join = [].join; // скопируем ссылку на функцию в переменную

// вызовем join с this=arguments,

// этот вызов эквивалентен arguments.join(':') из примера выше

var argStr = join.call(arguments, ':');

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

}

printArgs(1, 2, 3);

Мы вызвали метод без копирования. Чисто, безопасно.

##### Ещё пример: [].slice.call(arguments)

В JavaScript есть очень простой способ сделать из arguments настоящий массив. Для этого возьмём метод массива: [slice](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/slice).

По стандарту вызов arr.slice(start, end) создаёт новый массив и копирует в него элементы массива arr от start до end. А если start и end не указаны, то копирует весь массив.

Вызовем его в контексте arguments:

function printArgs() {

// вызов arr.slice() скопирует все элементы из this в новый массив

var args = [].slice.call(arguments);

alert( args.join(', ') ); // args - полноценный массив из аргументов

}

printArgs('Привет', 'мой', 'мир'); // Привет, мой, мир

Как и в случае с join, такой вызов технически возможен потому, что slice для работы требует только нумерованные свойства и length. Всё это в arguments есть.

##### Метод apply

Если нам неизвестно, с каким количеством аргументов понадобится вызвать функцию, можно использовать более мощный метод: apply.

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

func.call(context, arg1, arg2);

// идентичен вызову

func.apply(context, [arg1, arg2]);

В частности, эти две строчки сработают одинаково:

showFullName.call(user, 'firstName', 'surname');

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

Преимущество apply перед call отчётливо видно, когда мы формируем массив аргументов динамически.

Например, в JavaScript есть встроенная функция Math.max(a, b, c...), которая возвращает максимальное значение из аргументов:

alert( Math.max(1, 5, 2) ); // 5

При помощи apply мы могли бы найти максимум в произвольном массиве, вот так:

var arr = [];

arr.push(1);

arr.push(5);

arr.push(2);

// получить максимум из элементов arr

alert( Math.max.apply(null, arr) ); // 5

В примере выше мы передали аргументы через массив – второй параметр apply… Но вы, наверное, заметили небольшую странность? В качестве контекста this был передан null.

Строго говоря, полным эквивалентом вызову Math.max(1,2,3) был бы вызов Math.max.apply(Math, [1,2,3]). В обоих этих вызовах контекстом будет объект Math.

Но в данном случае в качестве контекста можно передавать что угодно, поскольку в своей внутренней реализации метод Math.max не использует this. Действительно, зачем this, если нужно всего лишь выбрать максимальный из аргументов? Вот так, при помощи apply мы получили короткий и элегантный способ вычислить максимальное значение в массиве!

Вызов call/apply с null или undefined

В современном стандарте call/apply передают this «как есть». А в старом, без use strict, при указании первого аргумента null или undefined в call/apply, функция получает this = window, например:

Современный стандарт:

function f() {

"use strict";

alert( this ); // undefined

}

f.call(null); // null;

Без use strict:

function f() {

alert( this ); // window

}

f.call(null); // [object Window]

##### Итого про this

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

* При вызове функции как метода:

obj.func(...) // this = obj

obj["func"](...)

* При обычном вызове:

func(...) // this = window (ES3) /undefined (ES5)

В new:

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

Явное указание:

func.apply(context, args) // this = context (явная передача)

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

##### Задачи

**Перепишите суммирование аргументов**

Есть функция sum, которая суммирует все элементы массива:

function sum(arr) {

return arr.reduce(function(a, b) {

return a + b;

});

}

alert( sum([1, 2, 3]) ); // 6 (=1+2+3)

Создайте аналогичную функцию sumArgs(), которая будет суммировать все свои аргументы:

function sumArgs() {

/\* ваш код \*/

}

alert( sumArgs(1, 2, 3) ); // 6, аргументы переданы через запятую, без массива

Для решения примените метод reduce к arguments, используя call, apply или одалживание метода.

P.S. Функция sum вам не понадобится, она приведена в качестве примера использования reduce для похожей задачи.

**Первый вариант:**

function sumArgs() {

// скопируем reduce из массива

arguments.reduce = [].reduce;

return arguments.reduce(function(a, b) {

return a + b;

});

}

alert( sumArgs(4, 5, 6) ); // 15

**Второй вариант:**

Метод call здесь вполне подойдёт, так как требуется вызвать reduce в контексте arguments с одним аргументом.

function sumArgs() {

// запустим reduce из массива напрямую

return [].reduce.call(arguments, function(a, b) {

return a + b;

});

}

alert( sumArgs(4, 5, 6) ); // 15

**Примените функцию к аргументам**

Напишите функцию applyAll(func, arg1, arg2...), которая получает функцию func и произвольное количество аргументов.

Она должна вызвать func(arg1, arg2...), то есть передать в func все аргументы, начиная со второго, и возвратить результат.

Например:

// Применить Math.max к аргументам 2, -2, 3

alert( applyAll(Math.max, 2, -2, 3) ); // 3

// Применить Math.min к аргументам 2, -2, 3

alert( applyAll(Math.min, 2, -2, 3) ); // -2

Область применения applyAll, конечно, шире, можно вызывать её и со своими функциями:

function sum() { // суммирует аргументы: sum(1,2,3) = 6

return [].reduce.call(arguments, function(a, b) {

return a + b;

});

}

function mul() { // перемножает аргументы: mul(2,3,4) = 24

return [].reduce.call(arguments, function(a, b) {

return a \* b;

});

}

alert( applyAll(sum, 1, 2, 3) ); // -> sum(1, 2, 3) = 6

alert( applyAll(mul, 2, 3, 4) ); // -> mul(2, 3, 4) = 24

function applyAll(func) {

return func.apply(this, [].slice.call(arguments, 1));

}

alert( applyAll(sum, 1, 2, 3) ); // 6

alert( applyAll(mul, 2, 3, 4) ); // 24

alert( applyAll(Math.max, 2, -2, 3) ); // 3

alert( applyAll(Math.min, 2, -2, 3) ); // -2

### Массивы с числовыми индексами

*Массив* – разновидность объекта, которая предназначена для хранения пронумерованных значений и предлагает дополнительные методы для удобного манипулирования такой коллекцией. Это объект, напоминающий список, который обладает

некоторыми методами и свойствами. Ни размер, ни тип элементов массива не фиксированы. “Длина” массива не является верхней границей.

Они обычно используются для хранения упорядоченных коллекций данных, например – списка товаров на странице, студентов в группе и т.п.

#### Объявление

Литеральный вариант:

var names = ["Иванов", "Петров", "Сидоров", "Кузнецов"];

var arr = []; // Пустой массив;

Для доступа к элементам массива используется оператор [ ] (квадратные скобки).  
Слева от скобок должна быть ссылка на сам массив, внутри скобок должно находиться выражение, возвращающее неотрицательное целое значение.

Запись в массив:

myArray[0] = "Иванов"; // запись 0-го элемента в массив myArray

Чтение осуществляется аналогично :

*x = myArray[0];* // в результате в переменной x окажется значение первой ячейки массива - строковое значение "Иванов".

**Элементы нумеруются, начиная с нуля.**

Чтобы получить нужный элемент из массива – указывается его номер в квадратных скобках:

var fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits[0] ); // Яблоко

alert( fruits[1] ); // Апельсин

alert( fruits[2] ); // Слива

Элемент можно всегда заменить:

fruits[2] = 'Груша'; // теперь ["Яблоко", "Апельсин", "Груша"]

…Или добавить:

fruits[3] = 'Лимон'; // теперь ["Яблоко", "Апельсин", "Груша", "Лимон"]

Общее число элементов, хранимых в массиве, содержится в его свойстве length:

var fruits = ["Яблоко", "Апельсин", "Груша"];

alert( fruits.length ); // 3

**Через alert можно вывести и массив целиком.**

При этом его элементы будут перечислены через запятую:

var fruits = ["Яблоко", "Апельсин", "Груша"];

alert( fruits ); // Яблоко,Апельсин,Груша

**В массиве может храниться любое число элементов любого типа.**

В том числе, строки, числа, объекты, вот например:

// микс значений

var arr = [ 1, 'Имя', { name: 'Петя' }, true ];

// получить объект из массива и тут же -- его свойство

alert( arr[2].name ); // Петя

#### Методы pop/push, shift/unshift

Одно из применений массива – это [очередь](http://ru.wikipedia.org/wiki/%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29). В классическом программировании так называют упорядоченную коллекцию элементов, такую что элементы добавляются в конец, а обрабатываются – с начала.



В реальной жизни эта структура данных встречается очень часто. Например, очередь сообщений, которые надо показать на экране.

Очень близка к очереди еще одна структура данных: [стек](http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA). Это такая коллекция элементов, в которой новые элементы добавляются в конец и берутся с конца.



Например, стеком является колода карт, в которую новые карты кладутся сверху, и берутся – тоже сверху.

Для того, чтобы реализовывать эти структуры данных, и просто для более удобной работы с началом и концом массива существуют специальные методы.

#### Конец массива

pop

Удаляет *последний* элемент из массива и возвращает его:

var fruits = ["Яблоко", "Апельсин", "Груша"];

alert( fruits.pop() ); // удалили "Груша"

alert( fruits ); // Яблоко, Апельсин

push

Добавляет элемент *в конец* массива:

var fruits = ["Яблоко", "Апельсин"];

fruits.push("Груша");

alert( fruits ); // Яблоко, Апельсин, Груша

Вызов fruits.push(...) равнозначен fruits[fruits.length] = ....

#### Начало массива

shift

Удаляет из массива *первый* элемент и возвращает его:

var fruits = ["Яблоко", "Апельсин", "Груша"];

alert( fruits.shift() ); // удалили Яблоко

alert( fruits ); // Апельсин, Груша

unshift

Добавляет элемент *в начало* массива:

var fruits = ["Апельсин", "Груша"];

fruits.unshift('Яблоко');

alert( fruits ); // Яблоко, Апельсин, Груша

Методы push и unshift могут добавлять сразу по несколько элементов:

var fruits = ["Яблоко"];

fruits.push("Апельсин", "Персик");

fruits.unshift("Ананас", "Лимон");

// результат: ["Ананас", "Лимон", "Яблоко", "Апельсин", "Персик"]

alert( fruits );

#### Внутреннее устройство массива

Массив – это объект, где в качестве ключей выбраны цифры, с дополнительными методами и свойством length.

Так как это объект, то в функцию он передаётся по ссылке:

function eat(arr) {

arr.pop();

}

var arr = ["нам", "не", "страшен", "серый", "волк"]

alert( arr.length ); // 5

eat(arr);

eat(arr);

alert( arr.length ); // 3, в функцию массив не скопирован, а передана ссылка

**Ещё одно следствие – можно присваивать в массив любые свойства.**

Например:

var fruits = []; // создать массив

fruits[99999] = 5; // присвоить свойство с любым номером

fruits.age = 25**;** // назначить свойство со строковым именем. PS fruits.lenght = 10000;

… Но массивы для того и придуманы в JavaScript, чтобы удобно работать именно *с упорядоченными, нумерованными данными*. Для этого в них существуют специальные методы и свойство length.

Как правило, нет причин использовать массив как обычный объект, хотя технически это и возможно.

Вывод массива с «дырами»

Если в массиве есть пропущенные индексы, то при выводе в большинстве браузеров появляются «лишние» запятые, например:

var a = [];

a[0] = 0;

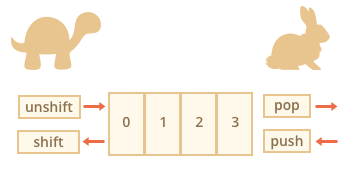
a[5] = 5;

alert( a ); // 0,,,,,5

Эти запятые появляются потому, что алгоритм вывода массива идёт от 0 до arr.length и выводит всё через запятую. Отсутствие значений даёт несколько запятых подряд.

#### Влияние на быстродействие

Методы push/pop выполняются быстро, а shift/unshift – медленно.



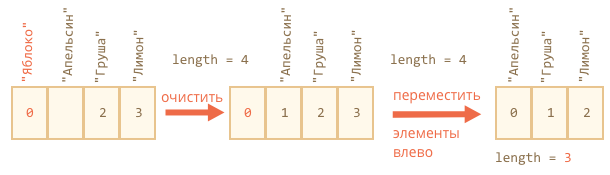
Чтобы понять, почему работать с концом массива – быстрее, чем с его началом, разберём подробнее происходящее при операции:

fruits.shift(); // убрать 1 элемент с начала

При этом, так как все элементы находятся в своих ячейках, просто удалить элемент с номером 0 недостаточно. Нужно еще и переместить остальные элементы на их новые индексы.

Операция shift должна выполнить целых три действия:

1. Удалить нулевой элемент.
2. Переместить все свойства влево, с индекса 1 на 0, с 2 на 1 и так далее.
3. Обновить свойство length.



**Чем больше элементов в массиве, тем дольше их перемещать, это много операций с памятью.**

Аналогично работает unshift: чтобы добавить элемент в начало массива, нужно сначала перенести вправо, в увеличенные индексы, все существующие.

А что же с push/pop? Им как раз перемещать ничего не надо. Для того, чтобы удалить элемент, метод pop очищает ячейку и уменьшает значение length.

Действия при операции:

fruits.pop(); // убрать 1 элемент с конца



**Перемещать при pop не требуется, так как прочие элементы после этой операции остаются на тех же индексах.**

Аналогично работает push.

#### Перебор элементов

Для перебора элементов обычно используется цикл:

var arr = ["Яблоко", "Апельсин", "Груша"];

for (var i = 0; i < arr.length; i++) {

alert( arr[i] );

}

Не используйте for..in для массивов

Так как массив является объектом, то возможен и вариант for..in:

var arr = ["Яблоко", "Апельсин", "Груша"];

for (var key in arr) {

alert( arr[key] ); // Яблоко, Апельсин, Груша

}

Недостатки этого способа:

1. Цикл for..in выведет *все свойства* объекта, а не только цифровые.

В браузере, при работе с объектами страницы, встречаются коллекции элементов, которые по виду как массивы, но имеют дополнительные нецифровые свойства. При переборе таких «похожих на массив» коллекций через for..in эти свойства будут выведены, а они как раз не нужны.

Бывают и библиотеки, которые предоставляют такие коллекции. Классический for надёжно выведет только цифровые свойства, что обычно и требуется.

1. Цикл for (var i=0; i<arr.length; i++) в современных браузерах выполняется в 10-100 раз быстрее. Казалось бы, по виду он сложнее, но браузер особым образом оптимизирует такие циклы.

**Если коротко: цикл for(var i=0; i<arr.length...) надёжнее и быстрее.**

**Особенности работы length**

Встроенные методы для работы с массивом автоматически обновляют его длину length.

**Длина length – не количество элементов массива, а последний индекс** + 1.

Так уж оно устроено.

Это легко увидеть на следующем примере:

var arr = [];

arr[1000] = true;

alert(arr.length); // 1001

Кстати, если у вас элементы массива нумеруются случайно или с большими пропусками, то стоит подумать о том, чтобы использовать обычный объект. Массивы предназначены именно для работы с непрерывной упорядоченной коллекцией элементов.

#### Используем length для укорачивания массива

Обычно нам не нужно самостоятельно менять length… Но есть один фокус, который можно провернуть.

**При уменьшении length массив укорачивается.**

Причем этот процесс необратимый, т.е. даже если потом вернуть length обратно – значения не восстановятся:

var arr = [1, 2, 3, 4, 5];

arr.length = 2; // укоротить до 2 элементов

alert( arr ); // [1, 2]

arr.length = 5; // вернуть length обратно, как было

alert( arr[3] ); // undefined: значения не вернулись

Самый простой способ очистить массив – это arr.length=0.

Еще один важный момент: length = индекс последнего элемента + 1, но не всегда length это кол-во элементов массива:

var a = ["a", "b", "c"];

a.length; // 3

a[0]; // "a"

a[1]; // "b"

a[2]; // "c"

a[3]; // undefined!

a[10] = "what?";

a.length; // 11,элементы массива с 3 по 9 будут undefined!

var a = [1, 2];

a[“something”] = 45;

a // [ 1, 2, something : 45 ]

a.length // 2

#### Создание вызовом new Array

new Array()

Существует еще один синтаксис для создания массива:

var arr = new Array("Яблоко", "Груша", "и т.п.");

var num = new Array(10);  // создание массива из 10 элементов

Он редко используется, т.к. квадратные скобки [] короче.

Кроме того, у него есть одна особенность. Обычно new Array(элементы, ...) создаёт массив из данных элементов, но если у него один аргумент-число new Array(число), то он создает массив *без элементов, но с заданной длиной*.

Проверим это:

var arr = new Array(2, 3);

alert( arr[0] ); // 2, создан массив [2, 3], всё ок

arr = new Array(5); // создаст массив [5] ?

alert( arr[0] ); // undefined! у нас массив без элементов, длины 5

alert( arr ); // [ , , , , ]

Что же такое этот «массив без элементов, но с длиной»? Как такое возможно?

Оказывается, очень даже возможно и соответствует объекту {length: 2}. Получившийся массив ведёт себя так, как будто его элементы равны undefined.

Это может быть неожиданным сюрпризом, поэтому обычно используют квадратные скобки.

new Array(3).toString(); //",,"

#### Многомерные массивы

Массивы в JavaScript могут содержать в качестве элементов другие массивы. Это можно использовать для создания многомерных массивов, например матриц:

var matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

];

alert( matrix[1][1] ); // центральный элемент

#### Внутреннее представление массивов

Hardcore coders only

Эта секция относится ко внутреннему устройству структуры данных и требует специальных знаний. Она не обязательна к прочтению.

Числовые массивы, согласно спецификации, являются объектами, в которые добавили ряд свойств, методов и автоматическую длину length. Но внутри они, как правило, устроены по-другому.

**Современные интерпретаторы стараются оптимизировать их и хранить в памяти не в виде хэш-таблицы, а в виде непрерывной области памяти, по которой легко пробежаться от начала до конца.**

Операции с массивами также оптимизируются, особенно если массив хранит только один тип данных, например только числа. Порождаемый набор инструкций для процессора получается очень эффективным.

Чтобы у интерпретатора получались эти оптимизации, программист не должен мешать.

В частности:

* Не ставить массиву произвольные свойства, такие как arr.test = 5. То есть, работать именно как с массивом, а не как с объектом.
* Заполнять массив непрерывно и по возрастающей. Как только браузер встречает необычное поведение массива, например устанавливается значение arr[0], а потом сразу arr[1000], то он начинает работать с ним, как с обычным объектом. Как правило, это влечёт преобразование его в хэш-таблицу.

Если следовать этим принципам, то массивы будут занимать меньше памяти и быстрее работать.

**Итого**

Массивы существуют для работы с упорядоченным набором элементов.

**Объявление:**

// предпочтительное

var arr = [элемент1, элемент2...];

// new Array

var arr = new Array(элемент1, элемент2...);

При этом new Array(число) создаёт массив заданной длины, *без элементов*. Чтобы избежать ошибок, предпочтителен первый синтаксис.

**Свойство length** – длина массива. Если точнее, то последний индекс массива плюс 1. Если её уменьшить вручную, то массив укоротится. Если length больше реального количества элементов, то отсутствующие элементы равны undefined.

Массив можно использовать как очередь или стек.

**Операции с концом массива:**

* arr.push(элемент1, элемент2...) добавляет элементы в конец.
* var elem = arr.pop() удаляет и возвращает последний элемент.

**Операции с началом массива:**

* arr.unshift(элемент1, элемент2...) добавляет элементы в начало.
* var elem = arr.shift() удаляет и возвращает первый элемент.

Эти операции перенумеровывают все элементы, поэтому работают медленно.

В следующей главе мы рассмотрим другие методы для работы с массивами.

#### Задачи

**Получить последний элемент массива**

Как получить последний элемент из произвольного массива?

У нас есть массив goods. Сколько в нем элементов – не знаем, но можем прочитать из goods.length.

Напишите код для получения последнего элемента goods.

Последний элемент имеет индекс на 1 меньший, чем длина массива.

Например:

var fruits = ["Яблоко" , "Груша", "Слива"];

Длина этого массива fruits.length равна 3. Здесь «Яблоко» имеет индекс 0, «Груша» – индекс 1, «Слива» – индекс 2.

То есть, для массива длины goods:

var lastItem = goods[goods.length - 1]; // получить последний элемент

**Добавить новый элемент в массив**

Как добавить элемент в конец произвольного массива?

У нас есть массив goods. Напишите код для добавления в его конец значения «Компьютер».

Текущий последний элемент имеет индекс goods.length-1. Значит, индексом нового элемента будет goods.length:

goods[goods.length] = 'Компьютер'

**Создание массива**

Создайте массив styles с элементами «Джаз», «Блюз».

Добавьте в конец значение «Рок-н-Ролл»

Замените предпоследнее значение с конца на «Классика». Код замены предпоследнего значения должен работать для массивов любой длины.

Удалите первое значение массива и выведите его alert.

Добавьте в начало значения «Рэп» и «Регги».

Массив в результате каждого шага:

Джаз, Блюз

Джаз, Блюз, Рок-н-Ролл

Джаз, Классика, Рок-н-Ролл

Классика, Рок-н-Ролл

Рэп, Регги, Классика, Рок-н-Ролл

var styles = ["Джаз", "Блюз"];

styles.push("Рок-н-Ролл");

styles[styles.length - 2] = "Классика";

alert( styles.shift() );

styles.unshift("Рэп", "Регги");

**Получить случайное значение из массива**

Напишите код для вывода alert случайного значения из массива:

var arr = ["Яблоко", "Апельсин", "Груша", "Лимон"];

P.S. Код для генерации случайного целого от min to max включительно:

var rand = min + Math.floor(Math.random() \* (max + 1 - min));

Для вывода нужен случайный номер от 0 до arr.length-1 включительно.

var arr = ["Яблоко", "Апельсин", "Груша", "Лимон"];

var rand = Math.floor(Math.random() \* arr.length);

alert( arr[rand] );

**Поиск в массиве**

Создайте функцию find(arr, value), которая ищет в массиве arr значение value и возвращает его номер, если найдено, или -1, если не найдено.

Например:

arr = ["test", 2, 1.5, false];

find(arr, "test"); // 0

find(arr, 2); // 1

find(arr, 1.5); // 2

find(arr, 0); // -1

function find(array, value) {

for (var i = 0; i < array.length; i++) {

if (array[i] === value) return i;

}

return -1;

}

Однако, в нем ошибка, т.к. сравнение == не различает 0 и false.

Поэтому лучше использовать ===. Кроме того, в современном стандарте JavaScript существует встроенная функция [Array#indexOf](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/indexOf), которая работает именно таким образом. Имеет смысл ей воспользоваться, если браузер ее поддерживает.

function find(array, value) {

if (array.indexOf) { // если метод существует

return array.indexOf(value);

}

for (var i = 0; i < array.length; i++) {

if (array[i] === value) return i;

}

return -1;

}

var arr = ["a", -1, 2, "b"];

var index = find(arr, 2);

alert( index );

… Но еще лучшим вариантом было бы определить find по-разному в зависимости от поддержки браузером метода indexOf:

// создаем пустой массив и проверяем поддерживается ли indexOf

if ([].indexOf) {

var find = function(array, value) {

return array.indexOf(value);

}

} else {

var find = function(array, value) {

for (var i = 0; i < array.length; i++) {

if (array[i] === value) return i;

}

return -1;

}

}

Этот способ – лучше всего, т.к. не требует при каждом запуске find проверять поддержку indexOf.

**Фильтр диапазона**

Создайте функцию filterRange(arr, a, b), которая принимает массив чисел arr и возвращает новый массив, который содержит только числа из arr из диапазона от a до b. То есть, проверка имеет вид a ≤ arr[i] ≤ b. Функция не должна менять arr.

Пример работы:

var arr = [5, 4, 3, 8, 0];

var filtered = filterRange(arr, 3, 5);

// теперь filtered = [5, 4, 3]

// arr не изменился

Алгоритм решения

Создайте временный пустой массив var results = [].

Пройдите по элементам arr в цикле и заполните его.

Возвратите results.

Решение

function filterRange(arr, a, b) {

var result = [];

for (var i = 0; i < arr.length; i++) {

if (arr[i] >= a && arr[i] <= b) {

result.push(arr[i]);

}

}

return result;

}

var arr = [5, 4, 3, 8, 0];

var filtered = filterRange(arr, 3, 5);

alert( filtered );

**Решето Эратосфена**

Целое число, большее 1, называется *простым*, если оно не делится нацело ни на какое другое, кроме себя и 1.

Древний алгоритм «Решето Эратосфена» для поиска всех простых чисел до n выглядит так:

1. Создать список последовательных чисел от 2 до n: 2, 3, 4, ..., n.
2. Пусть p=2, это первое простое число.
3. Зачеркнуть все последующие числа в списке с разницей в p, т.е. 2\*p, 3\*p, 4\*p и т.д. В случае p=2 это будут 4,6,8....
4. Поменять значение p на первое не зачеркнутое число после p.
5. Повторить шаги 3-4 пока p2 < n.
6. Все оставшиеся не зачеркнутыми числа – простые.

Посмотрите также [анимацию алгоритма](http://learn.javascript.ru/task/eratosthenes-sieve/sieve.gif).

Реализуйте «Решето Эратосфена» в JavaScript, используя массив.

Найдите все простые числа до 100 и выведите их сумму.

Их сумма равна 1060.

// шаг 1

var arr = [];

for (var i = 2; i < 100; i++) {

arr[i] = true

}

// шаг 2

var p = 2;

do {

// шаг 3

for (i = 2 \* p; i < 100; i += p) {

arr[i] = false;

}

// шаг 4

for (i = p + 1; i < 100; i++) {

if (arr[i]) break;

}

p = i;

} while (p \* p < 100); // шаг 5

// шаг 6 (готово)

// посчитать сумму

var sum = 0;

for (i = 0; i < arr.length; i++) {

if (arr[i]) {

sum += i;

}

}

alert( sum );

**Подмассив наибольшей суммы**

На входе массив чисел, например: arr = [1, -2, 3, 4, -9, 6].

Задача – найти непрерывный подмассив arr, сумма элементов которого максимальна.

Ваша функция должна возвращать только эту сумму.

Например:

getMaxSubSum([-1, 2, 3, -9]) = 5 (сумма выделенных)

getMaxSubSum([2, -1, 2, 3, -9]) = 6

getMaxSubSum([-1, 2, 3, -9, 11]) = 11

getMaxSubSum([-2, -1, 1, 2]) = 3

getMaxSubSum([100, -9, 2, -3, 5]) = 100

getMaxSubSum([1, 2, 3]) = 6 (неотрицательные - берем всех)

Если все элементы отрицательные, то не берём ни одного элемента и считаем сумму равной нулю:

getMaxSubSum([-1, -2, -3]) = 0

Постарайтесь придумать решение, которое работает за O(n2), а лучше за O(n) операций.

**Подсказка (медленное решение)**

Можно просто посчитать для каждого элемента массива все суммы, которые с него начинаются.

Например, для [-1, 2, 3, -9, 11]:

// Начиная с -1:

-1

-1 + 2

-1 + 2 + 3

-1 + 2 + 3 + (-9)

-1 + 2 + 3 + (-9) + 11

// Начиная с 2:

2

2 + 3

2 + 3 + (-9)

2 + 3 + (-9) + 11

// Начиная с 3:

3

3 + (-9)

3 + (-9) + 11

// Начиная с -9

-9

-9 + 11

// Начиная с -11

-11

Сделайте вложенный цикл, который на внешнем уровне бегает по элементам массива, а на внутреннем – формирует все суммы элементов, которые начинаются с текущей позиции.

Медленное решение

**Подсказка (быстрое решение)**

Будем идти по массиву и накапливать в некоторой переменной s текущую частичную сумму. Если в какой-то момент s окажется отрицательной, то мы просто присвоим s=0. Утверждается, что максимум из всех значений переменной s, случившихся за время работы, и будет ответом на задачу.

**Докажем этот алгоритм.**

В самом деле, рассмотрим первый момент времени, когда сумма s стала отрицательной. Это означает, что, стартовав с нулевой частичной суммы, мы в итоге пришли к отрицательной частичной сумме – значит, и весь этот префикс массива, равно как и любой его суффикс имеют отрицательную сумму.

Следовательно, от всего этого префикса массива в дальнейшем не может быть никакой пользы: он может дать только отрицательную прибавку к ответу.

**Быстрое решение**

function getMaxSubSum(arr) {

var maxSum = 0,

partialSum = 0;

for (var i = 0; i < arr.length; i++) {

partialSum += arr[i];

if (partialSum < 0) partialSum = 0;

maxSum = Math.max(maxSum, partialSum);

}

return maxSum;

}

alert( getMaxSubSum([-1, 2, 3, -9]) ); // 5

alert( getMaxSubSum([-1, 2, 3, -9, 11]) ); // 11

alert( getMaxSubSum([-2, -1, 1, 2]) ); // 3

alert( getMaxSubSum([100, -9, 2, -3, 5]) ); // 100

alert( getMaxSubSum([1, 2, 3]) ); // 6

alert( getMaxSubSum([-1, -2, -3]) ); // 0

Информацию об алгоритме вы также можете прочитать здесь: <http://e-maxx.ru/algo/maximum_average_segment> и здесь: [Maximum subarray problem](http://en.wikipedia.org/wiki/Maximum_subarray_problem).

Этот алгоритм требует ровно одного прохода по массиву, его сложность имеет оценку O(n).

### Массивы: методы

В этой главе мы рассмотрим встроенные методы массивов JavaScript.

**Методы**- это действия, которые могут быть выполнены над объектом. Они могут присутствовать у объекта, а могут и не присутствовать.

Синтаксис доступа к методу выглядит следующим образом:  
имяОбъекта.имяМетода(); // В скобках указываются параметры, которые необходимо передать в метод.

Например у строкового объекта String есть метод **toUpperCase()**, который возвращает содержимое  нашего объекта, то есть текст, заглавными буквами. В частности выполнение кода:

**var textString = "**Hello world!**";**  
**console.log(**textString.toUpperCase()**);**

приведет к выводу в консоль текста "**HELLO WORLD!**" (все буквы станут заглавными).

#### Метод split

Ситуация из реальной жизни. Мы пишем сервис отсылки сообщений и посетитель вводит имена тех, кому его отправить: Маша, Петя, Марина, Василий.... Но нам-то гораздо удобнее работать с массивом имен, чем с одной строкой.

К счастью, есть метод split(s), который позволяет превратить строку в массив, разбив ее по разделителю s. В примере ниже таким разделителем является строка из запятой и пробела.

var names = 'Маша, Петя, Марина, Василий';

var arr = names.split(', ');

for (var i = 0; i < arr.length; i++) {

alert( 'Вам сообщение ' + arr[i] );

}

Второй аргумент split

У метода split есть необязательный второй аргумент – ограничение на количество элементов в массиве. Если их больше, чем указано – остаток массива будет отброшен:

alert( "a,b,c,d".split(',', 2) ); // a,b

Разбивка по буквам

Вызов split с пустой строкой разобьёт по буквам:

var str = "тест";

alert( str.split('') ); // т,е,с,т

#### Метод join

Вызов arr.join(str) и делает в точности противоположное split. Он берет массив и склеивает его в строку, используя str как разделитель.

Например:

var arr = ['Маша', 'Петя', 'Марина', 'Василий'];

var str = arr.join(';');

alert( str ); // Маша;Петя;Марина;Василий

new Array + join = Повторение строки

Код для повторения строки 3 раза:

alert( new Array(4).join("ля") ); // ляляля

Как видно, new Array(4) делает массив без элементов длины 4, который join объединяет в строку, вставляя *между его элементами* строку "ля".

В результате, так как элементы пусты, получается повторение строки. Такой вот небольшой трюк.

#### Удаление из массива

Так как массивы являются объектами, то для удаления ключа можно воспользоваться обычным delete:

var arr = ["Я", "иду", "домой"];

delete arr[1]; // значение с индексом 1 удалено

// теперь arr = ["Я", undefined, "домой"];

alert( arr[1] ); // undefined

Да, элемент удален из массива, но не так, как нам этого хочется. Образовалась «дырка».

Это потому, что оператор delete удаляет пару «ключ-значение». Это – все, что он делает. Обычно же при удалении из массива мы хотим, чтобы оставшиеся элементы сдвинулись и заполнили образовавшийся промежуток.

Поэтому для удаления используются специальные методы: из начала – shift, с конца – pop, а из середины – splice, с которым мы сейчас познакомимся.

#### Метод splice

Метод splice – это универсальный раскладной нож для работы с массивами. Умеет все: удалять элементы, вставлять элементы, заменять элементы – по очереди и одновременно.

Его синтаксис:

arr.splice(index[, deleteCount, elem1, ..., elemN])

Удалить deleteCount элементов, начиная с номера index, а затем вставить elem1, ..., elemN на их место. Возвращает массив из удалённых элементов.

Этот метод проще всего понять, рассмотрев примеры.

Начнём с удаления:

var arr = ["Я", "изучаю", "JavaScript"];

arr.splice(1, 1); // начиная с позиции 1, удалить 1 элемент

alert( arr ); // осталось ["Я", "JavaScript"]

В следующем примере мы удалим 3 элемента и вставим другие на их место:

var arr = ["Я", "сейчас", "изучаю", "JavaScript"];

// удалить 3 первых элемента и добавить другие вместо них

arr.splice(0, 3, "Мы", "изучаем")

alert( arr ) // теперь ["Мы", "изучаем", "JavaScript"]

Здесь видно, что splice возвращает массив из удаленных элементов:

var arr = ["Я", "сейчас", "изучаю", "JavaScript"];

// удалить 2 первых элемента

var removed = arr.splice(0, 2);

alert( removed ); // "Я", "сейчас" <-- array of removed elements

Метод splice также может вставлять элементы без удаления, для этого достаточно установить deleteCount в 0:

var arr = ["Я", "изучаю", "JavaScript"];

// с позиции 2

// удалить 0

// вставить "сложный", "язык"

arr.splice(2, 0, "сложный", "язык");

alert( arr ); // "Я", "изучаю", "сложный", "язык", "JavaScript"

Допускается использование отрицательного номера позиции, которая в этом случае отсчитывается с конца:

var arr = [1, 2, 5]

// начиная с позиции индексом -1 (перед последним элементом)

// удалить 0 элементов,

// затем вставить числа 3 и 4

arr.splice(-1, 0, 3, 4);

alert( arr ); // результат: 1,2,3,4,5

#### Метод slice

Метод slice(begin, end) копирует участок массива от begin до end, не включая end. Исходный массив при этом не меняется.

Например:

var arr = ["Почему", "надо", "учить", "JavaScript"];

var arr2 = arr.slice(1, 3); // элементы 1, 2 (не включая 3)

alert( arr2 ); // надо, учить

Аргументы ведут себя так же, как и в строковом slice:

Если не указать end – копирование будет до конца массива:

var arr = ["Почему", "надо", "учить", "JavaScript"];

alert( arr.slice(1) ); // взять все элементы, начиная с номера 1

Можно использовать отрицательные индексы, они отсчитываются с конца:

var arr2 = arr.slice(-2); // копировать от 2-го элемента с конца и дальше

Если вообще не указать аргументов – скопируется весь массив:

var fullCopy = arr.slice();

Совсем как в строках

Синтаксис метода slice одинаков для строк и для массивов. Тем проще его запомнить.

#### Сортировка, метод sort(fn)

Метод sort() сортирует массив *на месте*. Например:

var arr = [ 1, 2, 15 ];

arr.sort();

alert( arr ); // 1, 15, 2

Не заметили ничего странного в этом примере?

Порядок стал 1, 15, 2, это точно не сортировка чисел. Почему?

**Это произошло потому, что по умолчанию sort сортирует, преобразуя элементы к строке.**

Поэтому и порядок у них строковый, ведь "2" > "15".

#### Свой порядок сортировки

Для указания своего порядка сортировки в метод arr.sort(fn) нужно передать функцию fn от двух элементов, которая умеет сравнивать их.

Внутренний алгоритм функции сортировки умеет сортировать любые массивы – апельсинов, яблок, пользователей, и тех и других и третьих – чего угодно. Но для этого ему нужно знать, как их сравнивать. Эту роль и выполняет fn.

Если эту функцию не указать, то элементы сортируются как строки.

Например, укажем эту функцию явно, отсортируем элементы массива как числа:

function compareNumeric(a, b) {

if (a > b) return 1;

if (a < b) return -1;

}

var arr = [ 1, 2, 15 ];

arr.sort(compareNumeric);

alert(arr); // 1, 2, 15

Обратите внимание, мы передаём в sort() именно саму функцию compareNumeric, без вызова через скобки. Был бы ошибкой следующий код:

arr.sort( compareNumeric() ); // не сработает

Как видно из примера выше, функция, передаваемая sort, должна иметь два аргумента.

Алгоритм сортировки, встроенный в JavaScript, будет передавать ей для сравнения элементы массива. Она должна возвращать:

* Положительное значение, если a > b < вначале должно идти первое значение, а потом второе >,
* Отрицательное значение, если a < b < вначале должно идти первое значение, а потом второе > ,
* Если равны – можно 0, но вообще – не важно, что возвращать, если их взаимный порядок не имеет значения.

Алгоритм сортировки

В методе sort, внутри самого интерпретатора JavaScript, реализован универсальный алгоритм сортировки. Как правило, это [««быстрая сортировка»»](http://algolist.manual.ru/sort/quick_sort.php), дополнительно оптимизированная для небольших массивов.

Он решает, какие пары элементов и когда сравнивать, чтобы отсортировать побыстрее. Мы даём ему функцию – способ сравнения, дальше он вызывает её сам.

Кстати, те значения, с которыми sort вызывает функцию сравнения, можно увидеть, если вставить в неё alert:

[1, -2, 15, 2, 0, 8].sort(function(a, b) {

alert( a + " <> " + b );

});

Сравнение compareNumeric в одну строку

Функцию compareNumeric для сравнения элементов-чисел можно упростить до одной строчки.

function compareNumeric(a, b) {

return a - b;

}

Эта функция вполне подходит для sort, так как возвращает положительное число, если a > b, отрицательное, если наоборот, и 0, если числа равны. Если результат, возвращаемый функцией сравнения (b – a), меньше 0, то сортировка поставит а перед b, и наоборот. Например, чтобы вывести в консоль значения , отсортированные в обратном алфавитном порядке мы можем воспользоваться вот такой командой:

console.log(myArray.sort(function(a, b) {return b - a;}));

Однако надо отметить, что такая функция сравнения сработает только если элементы нашего массива состоят из цифр. Если мы хотим изменить направление сортировки для строковых элементов, функция сравнения должна будет выглядеть чуть иначе, например вот так:

console.log(myArray.sort(function(a, b) {return b.localeCompare(a);}));

В данном случае мы использовали метод **localeCompare()** объекта **String**, позволяющий сравнивать объекты типа **String** или строковые литералы.

#### reverse

Метод [arr.reverse()](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/reverse) меняет порядок элементов в массиве на обратный.

var arr = [1, 2, 3];

arr.reverse();

alert( arr ); // 3,2,1

#### concat

Метод [arr.concat(value1, value2, … valueN)](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/concat) создаёт новый массив, в который копируются элементы из arr, а также value1, value2, ... valueN.

Например:

var arr = [1, 2];

var newArr = arr.concat(3, 4);

alert( newArr ); // 1,2,3,4

У concat есть одна забавная особенность.

Если аргумент concat – массив, то concat добавляет элементы из него.

Например:

var arr = [1, 2];

var newArr = arr.concat([3, 4], 5); // то же самое, что arr.concat(3,4,5)

alert( newArr ); // 1,2,3,4,5

var a = "первая строка";  
var b = " вторая строка";

var c = " третья строка";   
result = a.concat(b , c);

В результате значение переменной result станет "первая строка вторая строка третья строка"

var a = [1, 2];

var c = [9, 2, 5, 6 ,7];

var d = c.concat(a, "end"); // [9, 2, 5, 6 ,7, 1, 2, "end"];

#### indexOf/lastIndexOf

Эти методы не поддерживаются в IE8-. Для их поддержки подключите библиотеку [ES5-shim](https://github.com/kriskowal/es5-shim).

Метод [«arr.indexOf(searchElement[, fromIndex])»](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/indexOf) возвращает номер элемента searchElement в массиве arr или -1, если его нет.

Поиск начинается с номера fromIndex, если он указан. Если нет – с начала массива.

**Для поиска используется строгое сравнение ===.**

Например:

var arr = [1, 0, false];

alert( arr.indexOf(0) ); // 1

alert( arr.indexOf(false) ); // 2

alert( arr.indexOf(null) ); // -1

Как вы могли заметить, по синтаксису он полностью аналогичен методу [indexOf для строк](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/String/indexOf).

Метод [«arr.lastIndexOf(searchElement[, fromIndex])»](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/lastIndexOf) ищет справа-налево: с конца массива или с номера fromIndex, если он указан.

Методы indexOf/lastIndexOf осуществляют поиск перебором

Если нужно проверить, существует ли значение в массиве – его нужно перебрать. Только так. Внутренняя реализация indexOf/lastIndexOf осуществляет полный перебор, аналогичный циклу for по массиву. Чем длиннее массив, тем дольше он будет работать.

Коллекция уникальных элементов

Рассмотрим задачу – есть коллекция строк, и нужно быстро проверять: есть ли в ней какой-то элемент. Массив для этого не подходит из-за медленного indexOf. Но подходит объект! Доступ к свойству объекта осуществляется очень быстро, так что можно сделать все элементы ключами объекта и проверять, есть ли уже такой ключ.

Например, организуем такую проверку для коллекции строк "div", "a" и "form":

var store = {}; // объект для коллекции

var items = ["div", "a", "form"];

for (var i = 0; i < items.length; i++) {

var key = items[i]; // для каждого элемента создаём свойство

store[key] = true; // значение здесь не важно

}

Теперь для проверки, есть ли ключ key, достаточно выполнить if (store[key]). Если есть – можно использовать значение, если нет – добавить.

Такое решение работает только со строками, но применимо к любым элементам, для которых можно вычислить строковый «уникальный ключ».

#### Object.keys(obj)

Ранее мы говорили о том, что свойства объекта можно перебрать в цикле for..in.

Если мы хотим работать с ними в виде массива, то к нашим услугам – замечательный метод [Object.keys(obj)](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Object/keys). Он поддерживается везде, кроме IE8-:

var user = {

name: "Петя",

age: 30

}

var keys = Object.keys(user);

alert( keys ); // name, age

**Итого**

Методы массивов:

* push/pop, shift/unshift, splice – для добавления и удаления элементов.
* join/split – для преобразования строки в массив и обратно.
* slice – копирует участок массива.
* sort – для сортировки массива. Если не передать функцию сравнения – сортирует элементы как строки.
* reverse – меняет порядок элементов на обратный.
* concat – объединяет массивы.
* indexOf/lastIndexOf – возвращают позицию элемента в массиве (не поддерживается в IE8-).

Дополнительно:

* Object.keys(arr) возвращает массив свойств объекта.

Изученных нами методов достаточно в 95% случаях, но существуют и другие. Для знакомства с ними рекомендуется заглянуть в справочник [Array](http://javascript.ru/Array) и [Array в Mozilla Developer Network](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array).

#### Задачи

**Добавить класс в строку**

В объекте есть свойство className, которое содержит список «классов» – слов, разделенных пробелом:

var obj = {

className: 'open menu'

}

Создайте функцию addClass(obj, cls), которая добавляет в список класс cls, но только если его там еще нет:

addClass(obj, 'new'); // obj.className='open menu new'

addClass(obj, 'open'); // без изменений (класс уже существует)

addClass(obj, 'me'); // obj.className='open menu new me'

alert( obj.className ); // "open menu new me"

P.S. Ваша функция не должна добавлять лишних пробелов.

Решение заключается в превращении obj.className в массив при помощи split. После этого в нем можно проверить наличие класса, и если нет – добавить.

function addClass(obj, cls) {

var classes = obj.**className** ? obj.className.split(' ') : [];

// split из строки в массив с разделителем пробел;

// obj.ClassName = “open menu“ после split = ["open", "menu"]

for (var i = 0; i < classes.length; i++) {

if (classes[i] == cls) return; // класс уже есть

}

classes.push(cls); // добавить

obj.className = classes.join(' '); //склеивает массив в строку и обновить свойство

}

P.S. «Альтернативный» подход к проверке наличия класса вызовом obj.className.indexOf(cls) был бы неверным. В частности, он найдёт cls = "menu" в строке классов obj.className = "open mymenu".

P.P.S. Проверьте, нет ли в вашем решении присвоения obj.className += " " + cls. Не добавляет ли оно лишний пробел в случае, если изначально obj.className = ""?

### Массив: перебирающие методы

Современный стандарт JavaScript предоставляет много методов для «умного» перебора массивов, которые есть в современных браузерах…

…Ну а для их поддержки в IE8- просто подключите библиотеку [ES5-shim](https://github.com/kriskowal/es5-shim).

#### forEach

Метод [«arr.forEach(callback[, thisArg])»](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/forEach) используется для перебора массива.

Он для каждого элемента массива вызывает функцию callback.

Этой функции он передаёт три параметра callback(item, i, arr):

* **item** – очередной элемент массива.
* **i** – его номер.
* **arr** – массив, который перебирается.

Например:

var arr = ["Яблоко", "Апельсин", "Груша"];

arr.forEach(function(item, i, arr) {

alert( i + ": " + item + " (массив:" + arr + ")" );

}); // 0: Яблоко (массив:Яблоко,Апельсин,Груша); 1: Апельсин (массив:Яблоко,Апельсин,Груша);

Второй, необязательный аргумент forEach позволяет указать контекст this для callback. Мы обсудим его в деталях чуть позже, сейчас он нам не важен.

Метод forEach ничего не возвращает, его используют только для перебора, как более «элегантный» вариант, чем обычный цикл for.

#### filter

Метод «arr.filter(callback[, thisArg])» используется для *фильтрации* массива через функцию.

Он создаёт новый массив, в который войдут только те элементы arr, для которых вызов callback(item, i, arr) возвратит true.

Например:

var arr = [1, -1, 2, -2, 3];

var positiveArr = arr.filter(function(number) {

return number > 0;

});

alert( positiveArr ); // 1,2,3

#### map

Метод «arr.map(callback[, thisArg])» используется для *трансформации* массива.

Он создаёт новый массив, который будет состоять из результатов вызова callback(item, i, arr) для каждого элемента arr.

Например:

var names = ['HTML', 'CSS', 'JavaScript'];

var nameLengths = names.map(function(name) {

return name.length;

});

// получили массив с длинами

alert( nameLengths ); // 4,3,10

#### every/some

Эти методы используются для проверки массива.

* Метод «arr.every(callback[, thisArg])» возвращает true, если вызов callback вернёт true для *каждого*элемента arr.
* Метод «arr.some(callback[, thisArg])» возвращает true, если вызов callback вернёт true для *какого-нибудь* элемента arr.

var arr = [1, -1, 2, -2, 3];

function isPositive(number) {

return number > 0;

}

alert( arr.every(isPositive) ); // false, не все положительные

alert( arr.some(isPositive) ); // true, есть хоть одно положительное

#### reduce/reduceRight

Метод «arr.reduce(callback[, initialValue])» используется для последовательной обработки каждого элемента массива с сохранением промежуточного результата.

Это один из самых сложных методов для работы с массивами. Но его стоит освоить, потому что временами с его помощью можно в несколько строк решить задачу, которая иначе потребовала бы в разы больше места и времени.

Метод reduce используется для вычисления на основе массива какого-либо единого значения, иначе говорят «для свёртки массива». Чуть далее мы разберём пример для вычисления суммы.

Он применяет функцию callback по очереди к каждому элементу массива слева направо, сохраняя при этом промежуточный результат.

Аргументы функции callback(**previousValue**, **currentItem**, index, arr):

* **previousValue** – последний результат вызова функции, он же «промежуточный результат».
* **currentItem** – текущий элемент массива, элементы перебираются по очереди слева-направо.
* index – номер текущего элемента.
* arr – обрабатываемый массив.

Кроме callback, методу можно передать «начальное значение» – аргумент initialValue. Если он есть, то на первом вызове значение previousValue будет равно initialValue, а если у reduce нет второго аргумента, то оно равно первому элементу массива, а перебор начинается со второго.

Проще всего понять работу метода reduce на примерах.

function adder(a,b) {

return a+b;

}

Например, в качестве «свёртки» мы хотим получить сумму всех элементов массива.

Вот решение в одну строку:

var arr = [1, 2, 3, 4, 5]

// для каждого элемента массива запустить функцию,

// промежуточный результат передавать первым аргументом далее

var result = arr.reduce(function(sum, current) {

return sum + current;

}, 0);

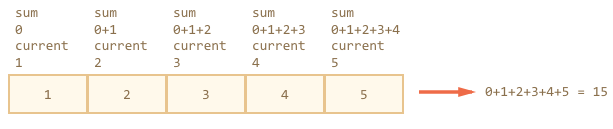
alert( result ); // 15

Разберём, что в нём происходит.

При первом запуске sum – исходное значение, с которого начинаются вычисления, равно нулю (второй аргумент reduce).

Сначала анонимная функция вызывается с этим начальным значением и первым элементом массива, результат запоминается и передаётся в следующий вызов, уже со вторым аргументом массива, затем новое значение участвует в вычислениях с третьим аргументом и так далее.

Поток вычислений получается такой



В виде таблицы где каждая строка – вызов функции на очередном элементе массива:

|  | sum | current | результат |
| --- | --- | --- | --- |
| первый вызов | 0 | 1 | 1 |
| второй вызов | 1 | 2 | 3 |
| третий вызов | 3 | 3 | 6 |
| четвёртый вызов | 6 | 4 | 10 |
| пятый вызов | 10 | 5 | 15 |

Как видно, результат предыдущего вызова передаётся в первый аргумент следующего.

Кстати, полный набор аргументов функции для reduce включает в себя function(sum, current, i, array), то есть номер текущего вызова i и весь массив arr, но здесь в них нет нужды.

Посмотрим, что будет, если не указать initialValue в вызове arr.reduce:

var arr = [1, 2, 3, 4, 5]

// убрали 0 в конце

var result = arr.reduce(function(sum, current) {

return sum + current

});

alert( result ); // 15

Результат – точно такой же! Это потому, что при отсутствии initialValue в качестве первого значения берётся первый элемент массива, а перебор стартует со второго.

Таблица вычислений будет такая же, за вычетом первой строки.

Метод reduceRight() применяет функцию к аккумулятору и каждому значению массива (справа-налево), сводя его к одному значению.

Синтаксис

arr.reduceRight(callback[, initialValue])

**Параметры**

**callback**

Функция, выполняющаяся для каждого элемента массива, принимает четыре аргумента:

**previousValue**

Значение, возвращённое предыдущим выполнением функции callback, либо значение initialValue, если оно предоставлено (смотрите пояснения ниже).

**currentValue**

Текущий обрабатываемый элемент массива.

**index**

Индекс текущего обрабатываемого элемента массива.

**array**

Массив, для которого была вызвана функция reduceRight.

**initialValue**

Необязательный параметр. Объект, используемый в качестве первого аргумента при первом вызове функции callback.

Описание

Метод reduceRight() выполняет функцию callback один раз для каждого элемента, присутствующего в массиве, за исключением пустот, принимая четыре аргумента: начальное значение (или значение от предыдущего вызова callback), значение текущего элемента, текущий индекс и массив, по которому происходит итерация.

Вызов функции обратного вызова callback будет выглядеть так:

array.reduceRight(function(previousValue, currentValue, index, array) {

// ...

});

При первом вызове функции, параметры previousValue и currentValue могут принимать одно из двух значений. Если при вызове reduceRight() передан аргумент initialValue, то значение previousValue будет равным значению initialValue, а значение currentValue будет равным последнему значению в массиве. Если аргумент initialValue не задан, то значение previousValue будет равным последнему значению в массиве, а значение currentValue будет равным второму с конца значению в массиве.

Если массив пустой и аргумент initialValue не указан, будет брошено исключение [TypeError](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/TypeError). Если массив состоит только из одного элемента (независимо от его положения в массиве) и аргумент initialValue не указан, или если аргумент initialValue указан, но массив пустой, то будет возвращено одно это значение, без вызова функции callback.

Вот так будут выглядеть некоторый примеры прогона функции:

[0, 1, 2, 3, 4].reduceRight(function(previousValue, currentValue, index, array) {

return previousValue + currentValue;

});

Функция обратного вызова будет вызвана четыре раза, аргументы и возвращаемое значение при каждом вызове будут следующими:

|  | previousValue | currentValue | index | array | возвращаемое значение |
| --- | --- | --- | --- | --- | --- |
| первый вызов | 4 | 3 | 3 | [0, 1, 2, 3, 4] | 7 |
| второй вызов | 7 | 2 | 2 | [0, 1, 2, 3, 4] | 9 |
| третий вызов | 9 | 1 | 1 | [0, 1, 2, 3, 4] | 10 |
| четвёртый вызов | 10 | 0 | 0 | [0, 1, 2, 3, 4] | 10 |

Значение, возвращённое методом reduceRight() будет равным последнему результату выполнения функции обратного вызова (10).

Если же вы зададите начальное значение initialValue, результат будет выглядеть так:

[0, 1, 2, 3, 4].reduceRight(function(previousValue, currentValue, index, array) {

return previousValue + currentValue;

}, 10);

|  | previousValue | currentValue | index | array | возвращаемое значение |
| --- | --- | --- | --- | --- | --- |
| первый вызов | 10 | 4 | 4 | [0, 1, 2, 3, 4] | 14 |
| второй вызов | 14 | 3 | 3 | [0, 1, 2, 3, 4] | 17 |
| третий вызов | 17 | 2 | 2 | [0, 1, 2, 3, 4] | 19 |
| четвёртый вызов | 19 | 1 | 1 | [0, 1, 2, 3, 4] | 20 |
| пятый вызов | 20 | 0 | 0 | [0, 1, 2, 3, 4] | 20 |

Значение, возвращённое методом reduceRight() на этот раз, конечно же, будет равным 20.

Примеры

**Пример: суммирование всех значений в массиве**

var total = [0, 1, 2, 3].reduceRight(function(a, b) {

return a + b;

});

// total == 6

**Пример: разворачивание массива массивов**

var flattened = [[0, 1], [2, 3], [4, 5]].reduceRight(function(a, b) {

return a.concat(b);

}, []);

// flattened равен [4, 5, 2, 3, 0, 1]

**Полифилл**

Метод Array.prototype.reduceRight() был добавлен к стандарту ECMA-262 в 5-м издании; поэтому он может отсутствовать в других реализациях стандарта. Вы можете работать с ним, добавив следующий код в начало ваших скриптов, он позволяет использовать reduceRight() в реализациях, которые не поддерживают этот метод.

// Шаги алгоритма ECMA-262, 5-е издание, 15.4.4.22

// Ссылка (ru): http://es5.javascript.ru/x15.4.html#x15.4.4.22

if ('function' !== typeof Array.prototype.reduceRight) {

Array.prototype.reduceRight = function(callback/\*, initialValue\*/) {

'use strict';

if (null === this || 'undefined' === typeof this) {

throw new TypeError('Array.prototype.reduce called on null or undefined');

}

if ('function' !== typeof callback) {

throw new TypeError(callback + ' is not a function');

}

var t = Object(this), len = t.length >>> 0, k = len - 1, value;

if (arguments.length >= 2) {

value = arguments[1];

} else {

while (k >= 0 && !k in t) {

k--;

}

if (k < 0) {

throw new TypeError('Reduce of empty array with no initial value');

}

value = t[k--];

}

for (; k >= 0; k--) {

if (k in t) {

value = callback(value, t[k], k, t);

}

}

return value;

};

}

**Итого**

Мы рассмотрели методы:

* forEach – для *перебора* массива.
* filter – для *фильтрации* массива.
* every/some – для *проверки* массива.
* map – для *трансформации* массива в массив.
* reduce/reduceRight – для *прохода по массиву с вычислением значения*.

Во многих ситуациях их использование позволяет написать код короче и понятнее, чем обычный перебор через for.

#### Задачи

**Перепишите цикл через map**

Код ниже получает из массива строк новый массив, содержащий их длины:

var arr = ["Есть", "жизнь", "на", "Марсе"];

var arrLength = [];

for (var i = 0; i < arr.length; i++) {

arrLength[i] = arr[i].length;

}

alert( arrLength ); // 4,5,2,5

Перепишите выделенный участок: уберите цикл, используйте вместо него метод map.

var arrLength = arr.map(function(arr){

return arr.lenght;

})

**Массив частичных сумм**

На входе массив чисел, например: arr = [1,2,3,4,5].

Напишите функцию getSums(arr), которая возвращает массив его частичных сумм.

Иначе говоря, вызов getSums(arr) должен возвращать новый массив из такого же числа элементов, в котором на каждой позиции должна быть сумма элементов arr до этой позиции включительно.

То есть:

для arr = [ 1, 2, 3, 4, 5 ]

getSums( arr ) = [ 1, 1+2, 1+2+3, 1+2+3+4, 1+2+3+4+5 ] = [ 1, 3, 6, 10, 15 ]

Еще пример: getSums([-2,-1,0,1]) = [-2,-3,-3,-2].

Функция не должна модифицировать входной массив.

В решении используйте метод arr.reduce.

Метод arr.reduce подходит здесь идеально. Достаточно пройтись по массиву слева-направа, накапливая текущую сумму в переменной и, кроме того, добавляя её в результирующий массив.

Неправильный вариант может выглядеть так:

function getSums(arr) {

var result = [];

if (!arr.length) return result;

arr.reduce(function(sum, item) {

result.push(sum); // добавить эл-т в конец массива;

return sum + item;

});

return result;

}

alert(getSums([1,2,3,4,5])); // результат: 1,3,6,10

Перед тем, как читать дальше, посмотрите на него внимательно. Заметили, в чём ошибка?

Если вы его запустите, то обнаружите, что результат не совсем тот. В получившемся массиве всего четыре элемента, отсутствует последняя сумма.

Это из-за того, что последняя сумма является результатом метода reduce, он на ней заканчивает проход и далее функцию не вызывает, поэтому она оказывается не добавленной в result.

Исправим это:

function getSums(arr) {

var result = [];

if (!arr.length) return result; // проверка на наличие элементов в массиве. Т.е. если массив пустой, сразу остановить функцию и как результат суммы вывести пустой массив

var totalSum = arr.reduce(function(sum, item) {

result.push(sum); // добавить эл-т в конец в массива, без нее все эл-ты сложатся => 15

return sum + item;

});

result.push(totalSum); // без этого будет в ответе 1,3,6,10

return result;

}

alert(getSums([1,2,3,4,5])); // 1,3,6,10,15

alert(getSums([-2,-1,0,1])); // -2,-3,-3,-2

var arr = [ 1, 2, 3, 4, 5 ];  
function getSums(arr) {  
 var res=[];  
 arr.reduce((sum,cur,i)=>res[i]=sum+cur,0);  
 return res;  
}  
console.log(getSums( arr ))

### Псевдомассив аргументов "arguments"

**Параметры отличаются от аргументов – это те переменные, которые описаны в функции до ее вызова**

• Дополнительный параметр, доступный во время вызова функции

• Доступ ко всем аргументам (а не только параметрам)

• Позволяет писать функции с неопределенным количеством аргументов

function myFuncName (x, y, z) {...}; //x,y,z - параметры

myFuncName (1,2,3,4,5,6,7,8); // 1,2,3…8 argumrnts

Среднее арфиметическое любого кол-ва аргкментов:

const average = () => {

let I = 0;

let sum = 0;

for(i=0; I < arguments.length; i++) {

sum += arguments[i];

}

return sum/arguments.length;

}

arguments() // NaN поскольку 0/не число

arguments(1, 4, 5) // 3.3333333335;

В JavaScript любая функция может быть вызвана с произвольным количеством аргументов.

Например:

function go(a,b) {

alert( "a=" +a + ", b=" +b );

}

go(1); // a=1, b=undefined

go(1,2); // a=1, b=2

go(1,2,3); // a=1, b=2, третий аргумент не вызовет ошибку

В JavaScript нет «перегрузки» функций

В некоторых языках программист может создать две функции с одинаковым именем, но разным набором аргументов, а при вызове интерпретатор сам выберет нужную:

function log(a) {

...

}

function log(a, b, c) {

...

}

log(a); // вызовется первая функция

log(a, b, c); // вызовется вторая функция

Это называется «полиморфизмом функций» или «перегрузкой функций». В JavaScript ничего подобного нет.

**Может быть только одна функция с именем log, которая вызывается с любыми аргументами.**

А уже внутри она может посмотреть, с чем вызвана и по-разному отработать.

В примере выше второе объявление log просто переопределит первое.

#### Доступ к «лишним» аргументам

Как получить значения аргументов, которых нет в списке параметров?

Доступ к ним осуществляется через «псевдо-массив» [arguments](https://developer.mozilla.org/en/JavaScript/Reference/functions_and_function_scope/arguments).

Он содержит список аргументов по номерам: arguments[0], arguments[1]…, а также свойство length.

Например, выведем список всех аргументов:

function sayHi() {

for (var i = 0; i < arguments.length; i++) {

alert( "Привет, " + arguments[i] );

}

}

sayHi("Винни", "Пятачок"); // 'Привет, Винни', 'Привет, Пятачок'

Все параметры находятся в arguments, даже если они есть в списке. Код выше сработал бы также, будь функция объявлена sayHi(a,b,c).

Связь между arguments и параметрами

В старом стандарте JavaScript псевдо-массив arguments и переменные-параметры ссылаются на одни и те же значения.

В результате изменения arguments влияют на параметры и наоборот.

Например:

function f(**x**) {

arguments[0] = 5; // меняет переменную x

alert( **x** ); // 5

}

f(**1**); // 5

Наоборот:

function f(**x**) {

**x** = 5;

alert( arguments[0] ); // 5, обновленный x

}

f(**1**); // 5

В современной редакции стандарта это поведение изменено. Аргументы отделены от локальных переменных:

function f(x) {

"use strict"; // для браузеров с поддержкой строгого режима

arguments[0] = 5;

alert( x ); // не 5, а 1! Переменная "отвязана" от arguments

}

f(1);

Если вы не используете строгий режим, то чтобы переменные не менялись «неожиданно», рекомендуется никогда не изменять arguments.

#### arguments – это не массив

Частая ошибка новичков – попытка применить методы Array к arguments. Это невозможно:

function sayHi() {

var a = arguments.shift(); // ошибка! нет такого метода!

}

sayHi(1);

Дело в том, что arguments – это не массив Array.

В действительности, это обычный объект, просто ключи числовые и есть length. На этом сходство заканчивается. Никаких особых методов у него нет, и методы массивов он тоже не поддерживает.

Впрочем, никто не мешает сделать обычный массив из arguments, например так:

var args = [];

for (var i = 0; i < arguments.length; i++) {

args[i] = arguments[i];

}

Такие объекты иногда называют *«коллекциями»* или *«псевдомассивами»*.

#### Пример: копирование свойств copy(dst, src1, src2...)

Иногда встаёт задача – скопировать в существующий объект свойства из одного или нескольких других.

Напишем для этого функцию copy. Она будет работать с любым числом аргументов, благодаря использованию arguments.

Синтаксис:

copy(dst, src1, src2…)

Копирует свойства из объектов src1, src2,... в объект dst. Возвращает получившийся объект.

Использование:

Для объединения нескольких объектов в один:

var vasya = {

age: 21,

name: 'Вася',

surname: 'Петров'

};

var user = {

isAdmin: false,

isEmailConfirmed: true

};

var student = {

university: 'My university'

};

// добавить к vasya свойства из user и student

copy(vasya, user, student);

alert( vasya**.isAdmin** ); // false

alert( vasya.**university** ); // My university

Для создания копии объекта user:

// скопирует все свойства в пустой объект

var userClone = copy({}, user);

Такой «клон» объекта может пригодиться там, где мы хотим изменять его свойства, при этом не трогая исходный объект user.

В нашей реализации мы будем копировать только свойства первого уровня, то есть вложенные объекты как-то особым образом не обрабатываются. Впрочем, её можно расширить.

А вот и реализация:

function copy() {

var dst = arguments[0];

for (var i = 1; i < arguments.length; i++) {

var arg = arguments[i];

for (var key in arg) {

dst[key] = arg[key];

}

}

return dst;

}

Здесь первый аргумент copy – это объект, в который нужно копировать, он назван dst. Для упрощения доступа к нему можно указать его прямо в объявлении функции:

function copy(dst) {

// остальные аргументы остаются безымянными

for (var i = 1; i < arguments.length; i++) {

var arg = arguments[i];

for (var key in arg) {

dst[key] = arg[key];

}

}

return dst;

}

### Аргументы по умолчанию через ||

Если функция вызвана с меньшим количеством аргументов, чем указано, то отсутствующие аргументы считаются равными undefined.

Зачастую в случае отсутствия аргумента мы хотим присвоить ему некоторое «стандартное» значение или, иначе говоря, значение «по умолчанию». Это можно удобно сделать при помощи оператора логическое ИЛИ ||.

Например, функция showWarning, описанная ниже, должна показывать предупреждение. Для этого она принимает ширину width, высоту height, заголовок title и содержимое contents, но большая часть этих аргументов необязательна:

function showWarning(width, height, title, contents) {

width = width || 200; // если не указана width, то width = 200

height = height || 100; // если нет height, то height = 100

title = title || "Предупреждение";

//...

}

Это отлично работает в тех ситуациях, когда «нормальное» значение параметра в логическом контексте отлично от false. В коде выше, при передаче width = 0 или width = null, оператор ИЛИ заменит его на значение по умолчанию.

А что, если мы хотим использовать значение по умолчанию только если width === undefined? В этом случае оператор ИЛИ уже не подойдёт, нужно поставить явную проверку:

function showWarning(width, height, title, contents) {

if (width === undefined) width = 200;

if (height === undefined) height = 100;

if (title === undefined) title = "Предупреждение";

//...

}

#### Устаревшее свойство arguments.callee

Используйте NFE вместо arguments.callee

Это свойство устарело, при use strict оно не работает.

Единственная причина, по которой оно тут – это то, что его можно встретить в старом коде, поэтому о нём желательно знать.

Современная спецификация рекомендует использовать [именованные функциональные выражения (NFE)](http://learn.javascript.ru/named-function-expression#functions-nfe).

В старом стандарте JavaScript объект arguments не только хранил список аргументов, но и содержал в свойстве arguments.callee ссылку на функцию, которая выполняется в данный момент.

Например:

function f() {

alert( arguments.callee === f ); // true

}

f();

Эти два примера будут работать одинаково:

// подвызов через NFE

var factorial = function f(n) {

return n==1 ? 1 : n\*f(n-1);

};

// подвызов через arguments.callee

var factorial = function(n) {

return n==1 ? 1 : n\*arguments.callee(n-1);

};

В учебнике мы его использовать не будем, оно приведено для общего ознакомления.

#### arguments.callee.caller

Устаревшее свойство arguments.callee.caller хранит ссылку на *функцию, которая вызвала данную*.

**Это свойство тоже устарело**

Это свойство было в старом стандарте, при use strict оно не работает, как и arguments.callee.

Также ранее существовало более короткое свойство arguments.caller. Но это уже раритет, оно даже не кросс-браузерное. А вот свойство arguments.callee.caller поддерживается везде, если не использован use strict, поэтому в старом коде оно встречается.

Пример работы:

f1();

function f1() {

alert( arguments.callee.caller ); // null, меня вызвали из глобального кода

f2();

}

function f2() {

alert( arguments.callee.caller ); // f1, функция, из которой меня вызвали

f3();

}

function f3() {

alert( arguments.callee.caller ); // f2, функция, из которой меня вызвали

}

В учебнике мы это свойство также не будем использовать.

#### «Именованные аргументы»

*Именованные аргументы* – альтернативная техника работы с аргументами, которая вообще не использует arguments.

Некоторые языки программирования позволяют передать параметры как-то так: f(width=100, height=200), то есть по именам, а что не передано, тех аргументов нет. Это очень удобно в тех случаях, когда аргументов много, сложно запомнить их порядок и большинство вообще не надо передавать, по умолчанию подойдёт.

Такая ситуация часто встречается в компонентах интерфейса. Например, у «меню» может быть масса настроек отображения, которые можно «подкрутить» но обычно нужно передать всего один-два главных параметра, а остальные возьмутся по умолчанию.

В JavaScript для этих целей используется передача аргументов в виде объекта, а в его свойствах мы передаём параметры.

Получается так:

function showWarning(options) {

var width = options.width || 200; // по умолчанию

var height = options.height || 100;

var contents = options.contents || "Предупреждение";

// ...

}

Вызвать такую функцию очень легко. Достаточно передать объект аргументов, указав в нем только нужные:

showWarning({

contents: "Вы вызвали функцию" // и всё понятно!

});

Сравним это с передачей аргументов через список:

showWarning(null, null, "Предупреждение!");

// мысль программиста "а что это за null, null в начале? ох, надо глядеть описание функции"

Не правда ли, объект – гораздо проще и понятнее?

Еще один бонус кроме красивой записи – возможность повторного использования объекта аргументов:

var opts = {

width: 400,

height: 200,

contents: "Текст"

};

showWarning(opts);

opts.contents = "Другой текст";

showWarning(opts); // вызвать с новым текстом, без копирования других аргументов

Именованные аргументы применяются во многих JavaScript-фреймворках.

**Итого**

Полный список аргументов, с которыми вызвана функция, доступен через arguments.

Это псевдомассив, то есть объект, который похож на массив, в нём есть нумерованные свойства и length, но методов массива у него нет.

В старом стандарте было свойство arguments.callee со ссылкой на текущую функцию, а также свойство arguments.callee.caller, содержащее ссылку на функцию, которая вызвала данную. Эти свойства устарели, при use strict обращение к ним приведёт к ошибке.

Для указания аргументов по умолчанию, в тех случаях, когда они заведомо не false, удобен оператор ||.

В тех случаях, когда возможных аргументов много и, в особенности, когда большинство их имеют значения по умолчанию, вместо работы с arguments организуют передачу данных через объект, который как правило называют options.

Возможен и гибридный подход, при котором первый аргумент обязателен, а второй – options, который содержит всевозможные дополнительные параметры:

function showMessage(text, options) {

// показать сообщение text, настройки показа указаны в options

}

#### Задачи

Проверка на аргумент-undefined

Как в функции отличить отсутствующий аргумент от undefined?

function f(x) {

// ..ваш код..

// выведите 1, если первый аргумент есть, и 0 - если нет

}

f(undefined); // 1

f(); // 0

Узнать количество реально переданных аргументов можно по значению arguments.length:

function f(x) {

alert( arguments.length ? 1 : 0 ); //arguments.lenght = 0 => false и покажет 0;

}

f(undefined);

f();

**Сумма аргументов**

Напишите функцию sum(...), которая возвращает сумму всех своих аргументов:

sum() = 0

sum(1) = 1

sum(1, 2) = 3

sum(1, 2, 3) = 6

sum(1, 2, 3, 4) = 10

function sum() {

var result = 0;

for (var i = 0; i < arguments.length; i++) {

if(isNaN(arguments[i])) continue; //если преобразовать к числу нельзя - пропускаем

result += arguments[i];

}

return result;

}

# Оператор typeof

Оператор typeof возвращает тип аргумента.

У него есть два синтаксиса: со скобками и без:

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

Работают они одинаково, но первый синтаксис короче.

**Результатом typeof является строка, содержащая тип:**

typeof undefined // "undefined"

typeof 0 // "number"

typeof true // "boolean"

typeof "foo" // "string"

typeof {} // "object"

typeof null // "object" (1)

typeof function(){} // "function" (2)

typeof Symbol() // "symbol"

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

1. Результат typeof null == "object" – это официально признанная ошибка в языке, которая сохраняется для совместимости. На самом деле null – это не объект, а отдельный тип данных.
2. Функции не являются отдельным базовым типом в JavaScript, а подвидом объектов. Но typeof выделяет функции отдельно, возвращая для них "function". На практике это весьма удобно, так как позволяет легко определить функцию.

# Зарезервированные имена

• break

• case

• catch

• continue

• debugger

• default

• delete

• do

• else

• finally

• for

• function

• if

• in

• instanceof

• new

• return

• switch

• this

• throw

• try

• typeof

• var

• void

• while

• with

• class

• enum

• export

• extends

• import

• super

# Основные операторы

Арность — это количество аргументов др.определение: это число аргументов или операндов, которые принимает функция.

Арность всех +, -, \* и / составляет 2, поэтому мы можем называть эти операторы двоичными.

Ассоциативность (или фиксированность) определяет, как операторы группируются в отсутствии скобок. Рассмотрим выражение a ~ b ~ c. Если у оператора ~ левая ассоциативность, это выражение будет трактоваться как (a ~ b) ~ c

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;

x = -x;

alert( x ); // -1, применили унарный минус

Или +1;

*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

## Сложение строк, бинарный +

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

var a = "моя" + "строка";

alert( a ); // моястрока

Иначе говорят, что «плюс производит конкатенацию (сложение) строк».

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"

**Пробел входит в строку и является его частью.**

## Преобразование к числу, унарный плюс +

Когда мы получаем значения из HTML-полей или от пользователя, то они обычно в форме строк.

А что, если их нужно, к примеру, сложить? Бинарный плюс сложит их как строки:

var apples = "2";

var oranges = "3";

alert( apples + oranges ); // "23", так как бинарный плюс складывает строки

Поэтому используем унарный плюс, чтобы преобразовать к числу:

var apples = "2";

var oranges = "3";

alert( +apples + +oranges ); // 5, число, оба операнда предварительно преобразованы в числа

Сначала выполнятся унарные плюсы, приведут строки к числам, а затем – бинарный '+' их сложит.

+true // 1

+false // 0

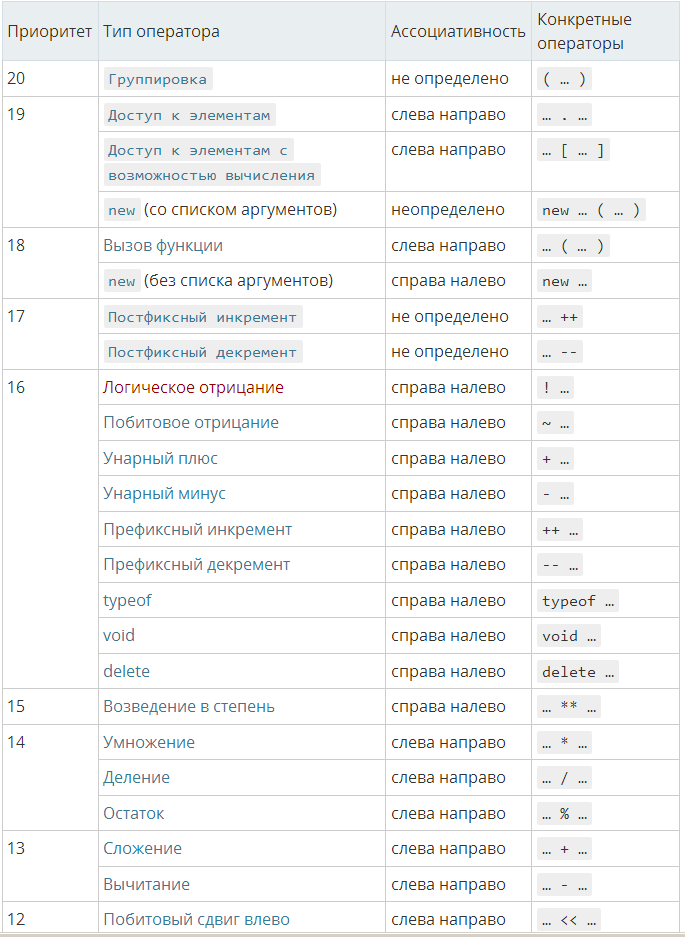
+null // 0

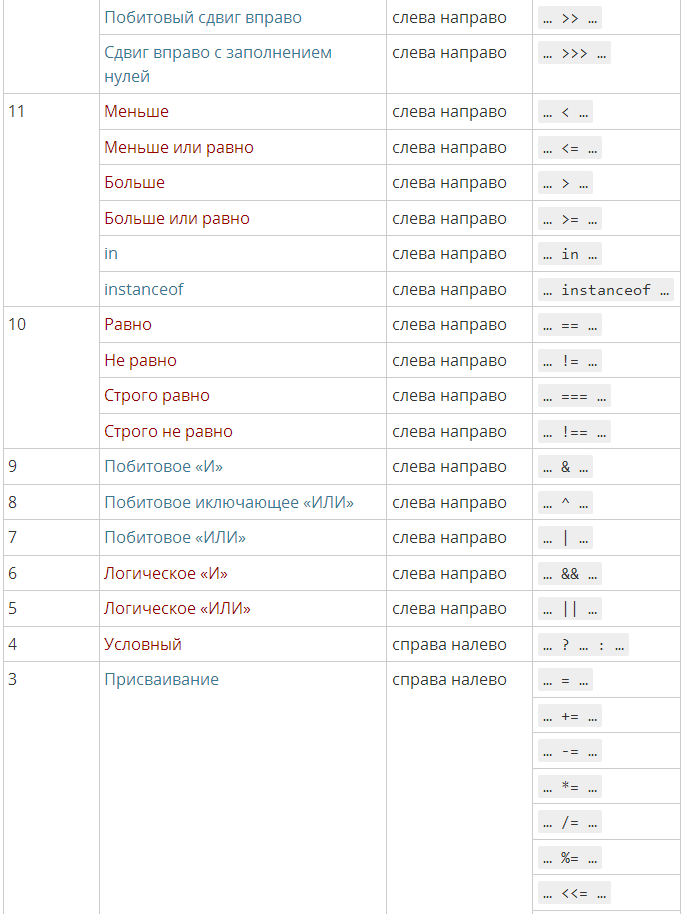
+function(val){ return val } // NaN

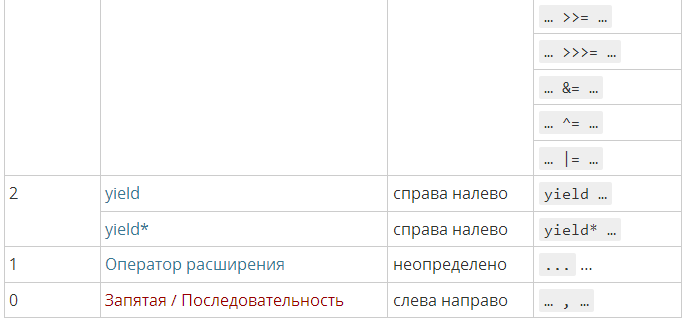
## Приоритет

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

В таблице каждому оператору задан числовой приоритет. Тот, у кого число больше – выполнится раньше. Если приоритет одинаковый, то порядок выполнения – слева направо.







Оператор "=" возвращает значение

Все операторы возвращают значение. Вызов x = выражение не является исключением.

Он записывает выражение в x, а затем возвращает его. Благодаря этому присваивание можно использовать как часть более сложного выражения:

var a = 1;

var b = 2;

var c = 3 - (a = b + 1);

alert( a ); // 3

alert( c ); // 0

Операторы делятся по типу возвращаемого значения на математические, строковые и логические.

Математические операторы:

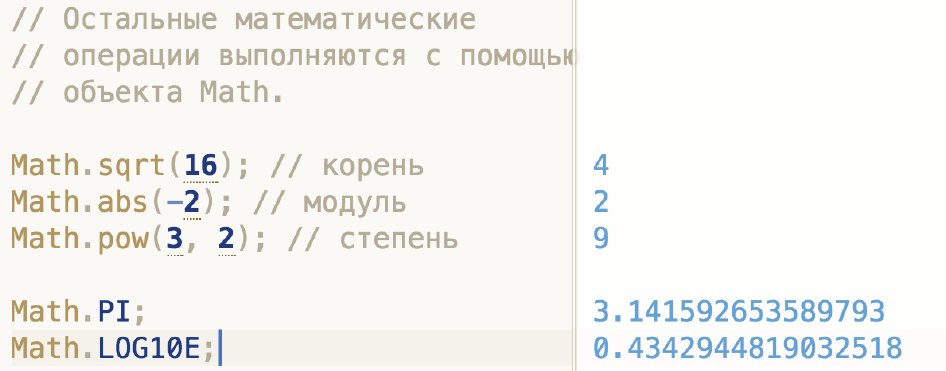
1 + 2 // сложение

1 – 2 // вычитание

1 \* 2 // умножение

1 / 2 // деление

10 % 3 // нахождение остатка



## Инкремент/декремент: ++, --

**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

// Postfix

var x = 3;

y = x++; // x = 4, y = 3

// Prefix

var a = 2;

b = ++a; // a = 3, b = 3

// Postfix

var x = 3;

y = x--; // x = 2, y = 3

// Prefix

var a = 2;

b = --a; // a = 1, b = 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. Так что 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

alert( i ); // 2

alert( 2 \* i++ ); // 4

alert( i ); // 3

При этом, нужно с осторожностью использовать такую запись, потому что в более длинной строке при быстром «вертикальном» чтении кода легко пропустить такой i++, и будет неочевидно, что переменая увеличивается.

Три строки, по одному действию в каждой – длиннее, зато нагляднее:

var i = 1;

alert( 2 \* i );

i++;

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

## 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:

|  |  |  |
| --- | --- | --- |
| 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. |

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.

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.

Один бит может быть нулем или единицей. 8 бит = байт.

один 0001

два 0010

три 0011

четыре 0100

пять 0101

шесть 0110

семь 0111

восемь 1000

девять 1001

одиннадцать 1011

четырнадцать 1110

пятнадцать 1111

Каждый следующий установленный бит (начиная справа) увеличивает число в два раза: 0001 в нашем примере = 1; 0010 два; 0100 четыре; 1000 восемь и т.д. Это и есть т.н. ***двоичная форма представления данных***.

Компьютер хранит данные в памяти именно так. Для обозначения какого-нибудь символа (цифры, буквы, запятой, точки...) в компьютере используется определенное количество бит. Компьютер \"распознает\" 256 (от 0 до 255) различных символов по их коду. Этого достаточно, чтобы вместить все цифры (0 - 9), буквы латинского алфавита (a - z, A - Z), русского (а - я, А - Я), а также другие символы. Для представления символа с максимально возможным кодом (255) нужно 8 бит. Эти 8 бит называются байтом. Т.о. один любой символ - это всегда 1 байт (см. рис. 1)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| р | н | р | н | н | р | н | р |

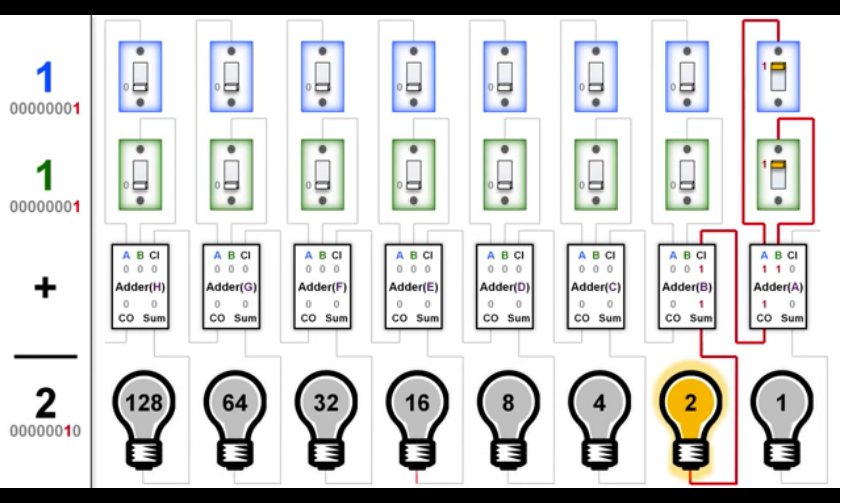
Рис. 1. Один байт с кодом буквы Z

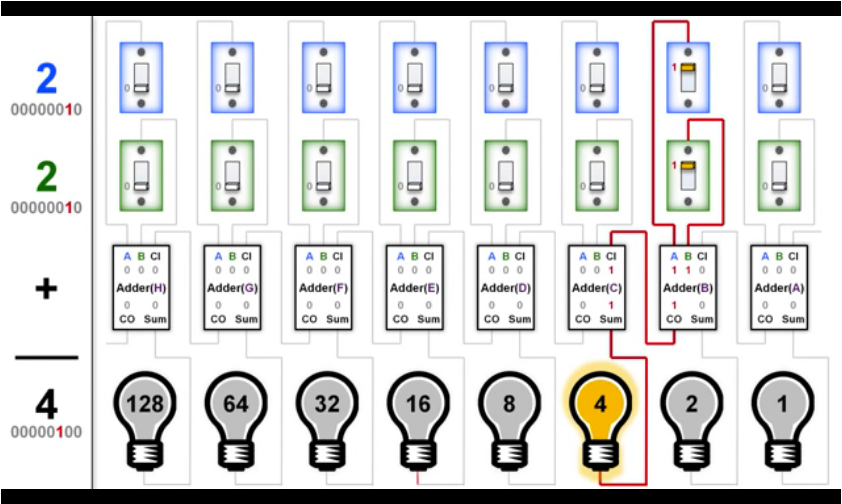
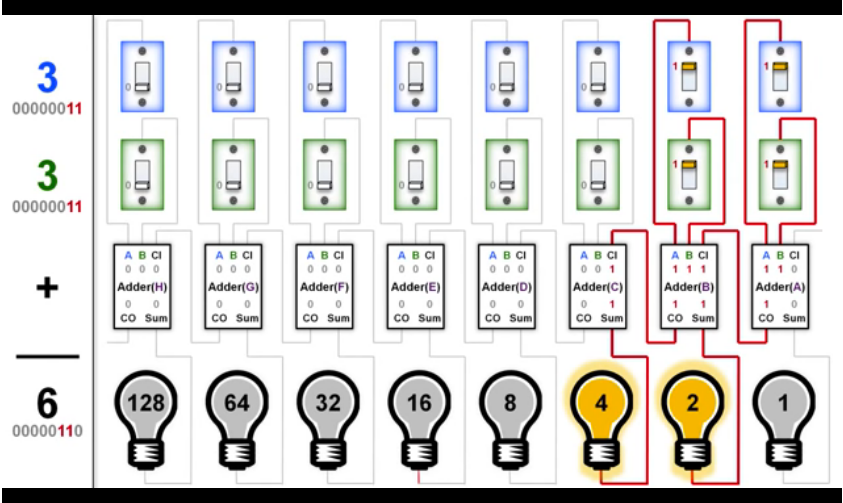
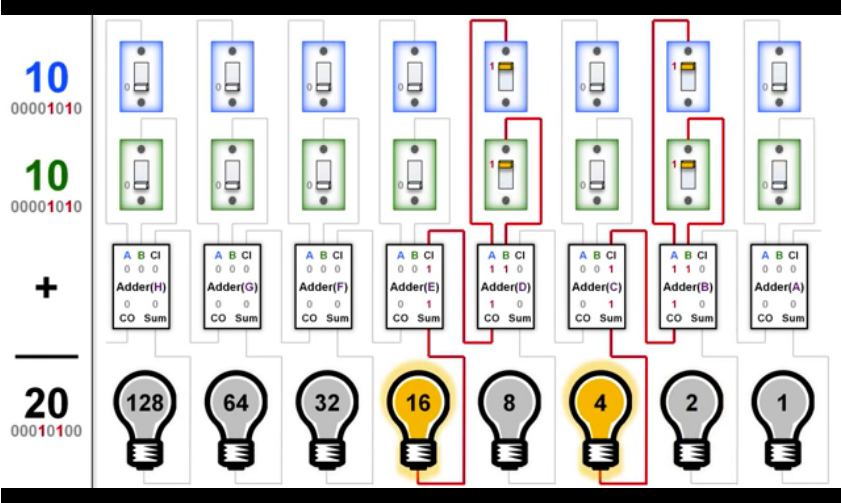
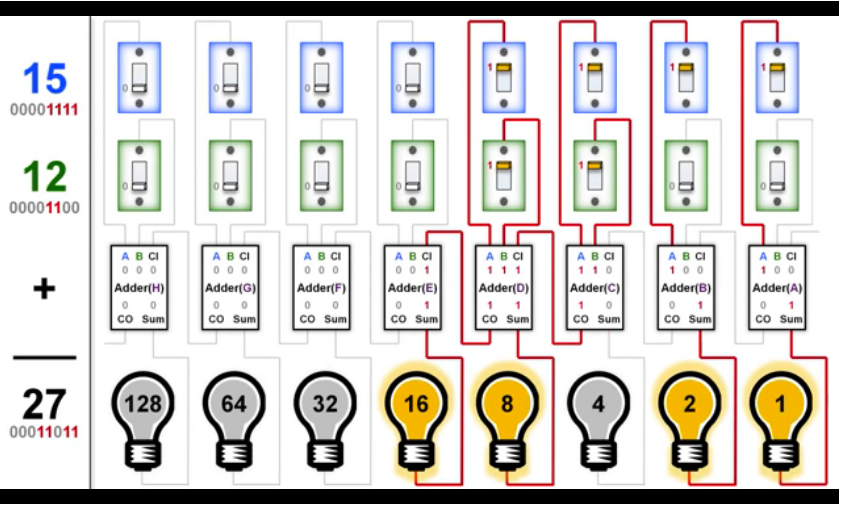
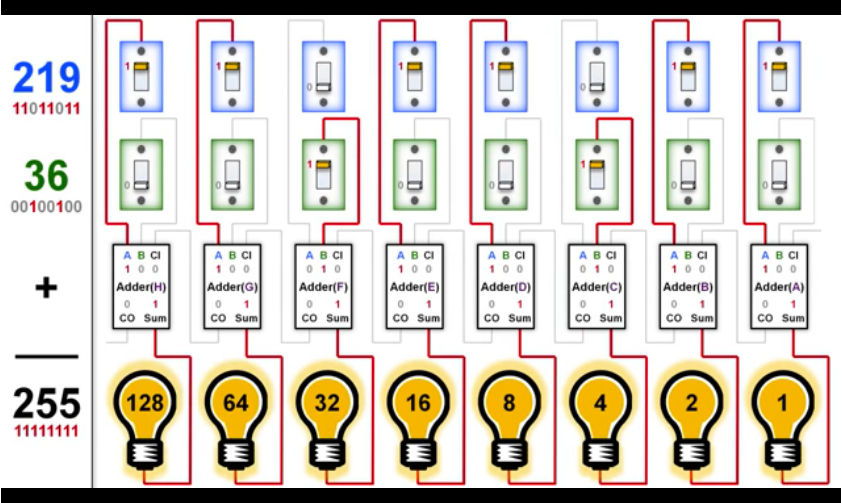
(буквы н и р обозначают: намагничено или размагничено соответственно)

Т.о. слово \"диск\" будет занимать 4 байта или 4\*8 = 32 бита. Как вы уже поняли, компьютер хранит в памяти не сами буквы этого слова, а последовательность \"единичек\" и \"ноликов\".

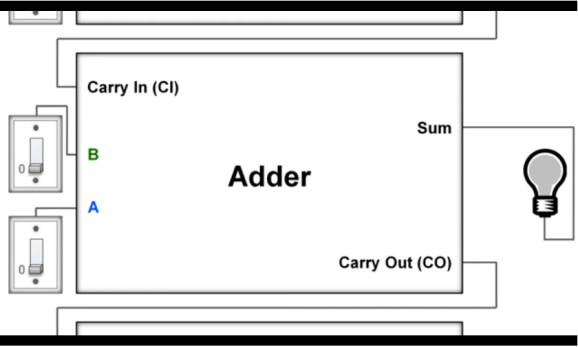
**Как компьютеры складывают числа**

<https://rakh.im/how_computers_add_numbers/>





full adder = полный сумматор



XOR = исключающее ИЛИ (true – когда один из операндов true, все остальное false)

NAND = и + нет, противоположность И(AND).

### Как перевести двоичное число в десятичное:

Надо сложить двойки в степенях, соответствующих позициям, где в двоичном стоят единицы. *Например:*

Возьмем число 20. В двоичной системе оно имеет следующий вид: 10100b

Итак (начнем слева направо, считая от 4 до 0; число в нулевой степени всегда равно единице (вспоминаем школьную программу по математике)):

10100b = 1\*24 + 0\*23 + 1\*22 + 0\*21 + 0\*20 = 20

---------------------------------------------

16+0+4+0+0 = 20

### Как перевести десятичное число в двоичное:

Можно делить его на два, записывая остаток справа налево:

20/2 = 10, остаток 0

10/2=5, остаток 0

5/2=2, остаток 1

2/2=1, остаток 0

1/2=0, остаток 1

В результате получае м: 10100b = 20

### Как перевести шестнадцатеричное число в десятичное:

В шестнадцатеричной системе номер позиции цифры в числе соответствует степени, в которую надо возвести число 16:

8Ah = 8\*16 + 10 (0Ah) = 138

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.

**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:

000000000000000000000001001110

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

11111111111111111111111011000101

Второй шаг – к полученному двоичному числу прибавить 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*.

### Вспомогательные функции parseInt, toString

Для удобной работы с примерами в этой статье, если вы захотите протестировать что-то в консоли, 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

Без них перевод в двоичную систему и обратно был бы куда менее удобен.

### & (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)

3.257582347 | 0; // 3 округление

### ^ (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)

### ~ (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

### << (Битовый сдвиг влево)

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

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)

Операция << 2 сдвинула и отбросила два левых нулевых бита и добавила справа два новых нулевых.

Левый сдвиг почти равен умножению на 2

Битовый сдвиг << N обычно имеет тот же эффект, что и умножение на два N раз, например:

alert( 3 << 1 ); // 6, умножение на 2

alert( 3 << 2 ); // 12, умножение на 2 два раза

alert( 3 << 3 ); // 24, умножение на 2 три раза

Конечно, следует иметь в виду, что побитовые операторы работают только с 32-битными числами, поэтому верхний порог такого «умножения» ограничен:

alert(10000000000 << 1); // -1474836480, отброшен крайний-левый бит

alert(10000000000 \* 2); // 20000000000, обычное умножение

### >> (Правый битовый сдвиг, переносящий знак)

Этот оператор сдвигает биты вправо, отбрасывая лишние. При этом слева добавляется ***копия* крайнего-левого бита**.

Знак числа (представленный крайним-левым битом) при этом не меняется, так как новый крайний-левый бит имеет то же значение, что и исходном числе.

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)

Операция >> 2 сдвинула вправо и отбросила два правых бита 01 и добавила слева две копии первого бита 00.

Аналогично, -9 >> 2 даст -3:

-9 (по осн.10)

= 11111111111111111111111111110111 (по осн.2)

--------------------------------

-9 >> 2 (по осн.10)

= 11111111111111111111111111111101 (по осн.2) = -3 (по осн.10)

Здесь операция >> 2 сдвинула вправо и отбросила два правых бита 11 и добавила слева две копии первого бита 11. , Знак числа сохранён, так как крайний-левый (знаковый) бит сохранил значение 1.

Правый сдвиг почти равен целочисленному делению на 2

Битовый сдвиг >> N обычно имеет тот же результат, что и целочисленное деление на два N раз:

alert( 100 >> 1 ); // 50, деление на 2

alert( 100 >> 2 ); // 25, деление на 2 два раза

alert( 100 >> 3 ); // 12, деление на 2 три раза, целая часть от результата

### >>> (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)

### 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 32-битных 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 в степени 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

При этом следует иметь в виду, что максимальный верхний порог такого умножения меньше, чем обычно, так как побитовый оператор манипулирует 32-битными целыми, в то время как обычные операторы работают с числами длиной 64 бита.

Оператор сдвига в другую сторону a >> b, производит обратную операцию – целочисленное деление a на 2b.

alert( 8 >> 2 ); // 2 = 8/4, убрали 2 нуля в двоичном представлении

alert( 11 >> 2 ); // 2, целочисленное деление (менее значимые биты просто отброшены)

## Сокращённая арифметика 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

Так можно сделать для операторов +,-,\*,/,% и бинарных <<,>>,>>>,&,|,^.

Вызов с присваиванием имеет в точности такой же приоритет, как обычное присваивание, то есть выполнится после большинства других операций:

var n = 2;

n \*= 3 + 5;

alert( n ); // 16 (n = 2 \* 8)

let b = 10;

b += b \*\* 2 > 20 ? 1 : 10; // 11

## Оператор запятая

Один из самых необычных операторов – запятая ',' .

Его можно вызвать явным образом, например:

var a = (5, 6);

alert( a );

Запятая позволяет перечислять выражения, разделяя их запятой ',' . Каждое из них – вычисляется и отбрасывается, за исключением последнего, кото

рое 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++) {

...

} // если console.log(a, b, c):

1 3 3 операции выполняются 1 раз, кроме последней.

2 3 3

3 3 3

4 3 3

5 3 3

6 3 3

7 3 3

8 3 3

9 3 3

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

Приведение типов —если операнды переданные оператору разного типа, они автоматически приводятся к типу, в котором

операцию можно выполнить:

Всего есть три преобразования:

1. Строковое преобразование.
2. Числовое преобразование.
3. Преобразование к логическому значению.

Приведение типов (coercion)существует в двух формах: явное и неявное. Явное приведение, это просто то приведение, которое можно очевидным образом увидеть в коде в виде конвертации из одного типа в другой, в свою очередь неявное приведение — это когда конвертация типа может произойти в результате менее очевидных побочных эффектов других операций.

Вот пример явного приведения:

var a = "42";

var b = Number( a );

a; // "42"

b; // 42 — число!

А вот пример неявного приведения:

var a = "42";

var b = a \* 1; // здесь "42" неявно приводится к 42

a; // "42"

b; // 42 — число!

**Строковое преобразование**

Строковое преобразование происходит, когда требуется представление чего-либо в виде строки. Например, его производит функция alert.

var a = true;

alert( a ); // "true"

Можно также осуществить преобразование явным вызовом String(val):

alert( String(null) === "null" ); // true

Как видно из примеров выше, преобразование происходит наиболее очевидным способом, «как есть»: false становится "false", null – "null", undefined – "undefined" и т.п.

Также для явного преобразования применяется оператор "+", у которого один из аргументов строка. В этом случае он приводит к строке и другой аргумент, например:

alert( true + "test" ); // "truetest"

alert( "123" + undefined ); // "123undefined"

**Численное преобразование**

Численное преобразование происходит в математических функциях и выражениях, а также при сравнении данных различных типов (кроме сравнений ===, !==).

Для преобразования к числу в явном виде можно вызвать Number(val), либо, что короче, поставить перед выражением унарный плюс "+":

var a = +"123"; // 123

var a = Number("123"); // 123, тот же эффект

| Значение | Преобразуется в... |
| --- | --- |
| undefined | NaN |
| null | 0 |
| true / false | 1 / 0 |
| Строка | Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то 0, иначе из непустой строки "считывается" число, при ошибке результат NaN. |

Например:

// после обрезания пробельных символов останется "123"

alert( +" \n 123 \n \n" ); // 123

Ещё примеры:

* Логические значения:

alert( +true ); // 1

alert( +false ); // 0

Сравнение разных типов – значит численное преобразование:

alert( "\n0 " == 0 ); // true

 При этом строка "\n0" преобразуется к числу, как указано выше: начальные и конечные пробелы обрезаются, получается строка "0", которая равна 0.

 С логическими значениями:

alert( "\n" == false );

alert( "1" == true );

* Здесь сравнение "==" снова приводит обе части к числу. В первой строке слева и справа получается 0, во второй 1.

**Специальные значения**

Посмотрим на поведение специальных значений более внимательно.

Интуитивно, значения null/undefined ассоциируются с нулём, но при преобразованиях ведут себя иначе.

Специальные значения преобразуются к числу так:

|  |  |
| --- | --- |
| Значение | Преобразуется в... |
| undefined | NaN |
| null | 0 |

Это преобразование осуществляется при арифметических операциях и сравнениях > >= < <=, но не при проверке равенства ==. Алгоритм проверки равенства для этих значений в спецификации прописан отдельно. В нём считается, что null и undefined равны "==" между собой, но эти значения не равны никакому другому значению.

Это ведёт к забавным последствиям.

Например, null не подчиняется законам математики – он «больше либо равен нулю»: null>=0, но не больше и не равен:

alert( null >= 0 ); // true, т.к. null преобразуется к 0

alert( null > 0 ); // false (не больше), т.к. null преобразуется к 0

alert( null == 0 ); // false (и не равен!), т.к. == рассматривает null особо.

Значение undefined вообще «несравнимо»:

alert( undefined > 0 ); // false, т.к. undefined -> NaN

alert( undefined == 0 ); // false, т.к. это undefined (без преобразования)

alert( undefined < 0 ); // false, т.к. undefined -> NaN

Для более очевидной работы кода и во избежание ошибок лучше не давать специальным значениям участвовать в сравнениях > >= < <=.

Используйте в таких случаях переменные-числа или приводите к числу явно.

**Логическое преобразование**

Преобразование к true/false происходит в логическом контексте, таком как if(value), и при применении логических операторов.

Все значения, которые интуитивно «пусты», становятся false. Их несколько: 0, пустая строка, null, undefined и NaN.

Остальное, в том числе и любые объекты – true.

Полная таблица преобразований:

|  |  |
| --- | --- |
| Значение | Преобразуется в... |
| undefined, null | false |
| Числа | Все true, кроме 0, NaN -- false. |
| Строки | Все true, кроме пустой строки "" -- false |
| Объекты | Всегда true |

Для явного преобразования используется двойное логическое отрицание !!value или вызов Boolean(value)

.

Обратите внимание: строка "0" становится true

В отличие от многих языков программирования (например PHP), "0" в JavaScript является true, как и строка из пробелов:

alert( !!"0" ); // true

alert( !!" " ); // любые непустые строки, даже из пробелов - true!

Логическое преобразование интересно тем, как оно сочетается с численным.

Два значения могут быть равны, но одно из них в логическом контексте true, другое – false.

Например, равенство в следующем примере верно, так как происходит численное преобразование:

alert( 0 == "\n0\n" ); // true

… А в логическом контексте левая часть (0) даст false, правая ("\n0\n") – true, так как любая не пустая строка в логическом контексте равна true:

if ("\n0\n") {

alert( "true, совсем не как 0!" );

}

С точки зрения преобразования типов в JavaScript это совершенно нормально. При сравнении с помощью «==» – численное преобразование, а в if – логическое, только и всего.

"" + 1 + 0 = "10" // (1)

"" - 1 + 0 = -1 // (2)

true + false = 1

6 / "3" = 2

"2" \* "3" = 6

4 + 5 + "px" = "9px"

"$" + 4 + 5 = "$45"

"4" – 2 = 2

"4px" – 2 = NaN

7 / 0 = Infinity

" -9\n" + 5 = " -9\n5"

" -9\n" – 5 = -14

5 && 2 = 2

2 && 5 = 5

5 || 0 = 5

0 || 5 = 5

null + 1 = 1 // (3)

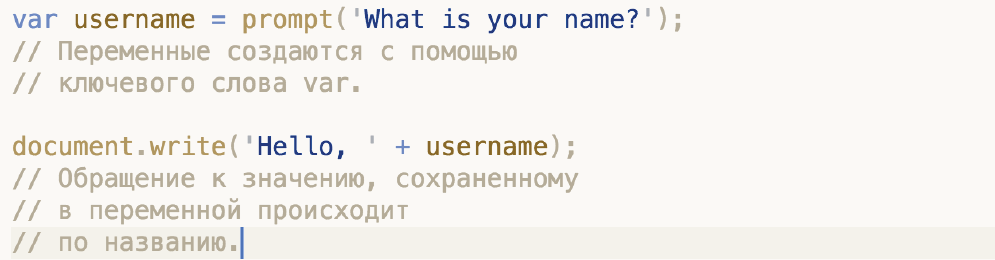
undefined + 1 = NaN // (4)

null == "\n0\n" = false // (5)

+null == +"\n0\n" = true // (6)

1. Оператор "+" в данном случае прибавляет 1 как строку, и затем 0.
2. Оператор "-" работает только с числами, так что он сразу приводит "" к 0.
3. null при численном преобразовании становится 0
4. undefined при численном преобразовании становится NaN
5. При сравнении == с null преобразования не происходит, есть жёсткое правило: null == undefined и только.
6. И левая и правая часть == преобразуются к числу 0.

Переменные Способ запомнить значения, чтобы использовать их после.



# Асинхронные скрипты

Если скрипт внешний, то пока браузер не выполнит его, он не покажет часть страницы под ним.

Async /атрибут только для внешних скриптов/

<script async src=”…”> браузер не останавливает обработку страницы, а спокойно работает дальше. Когда скрипт будет загружен – он выполнится. Раньше сработает тот скрипт, что раньше загрузится. Не ждет пока весь HTML-документ будет готов.

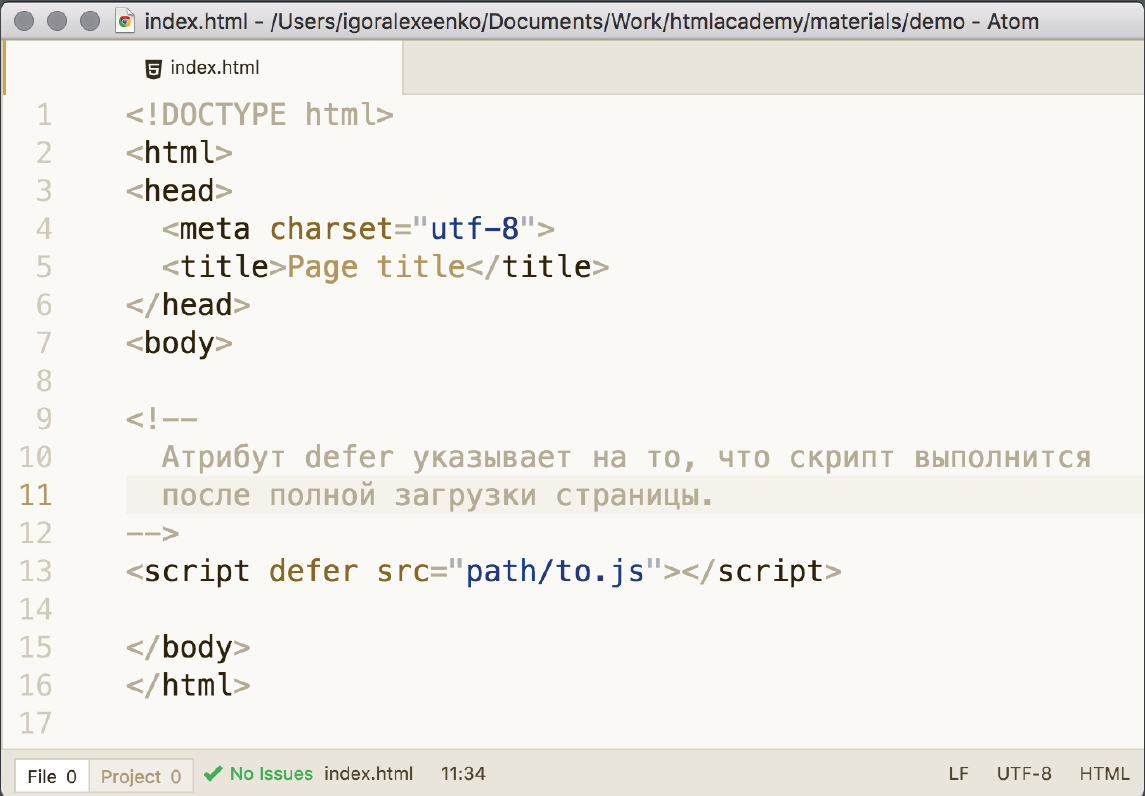
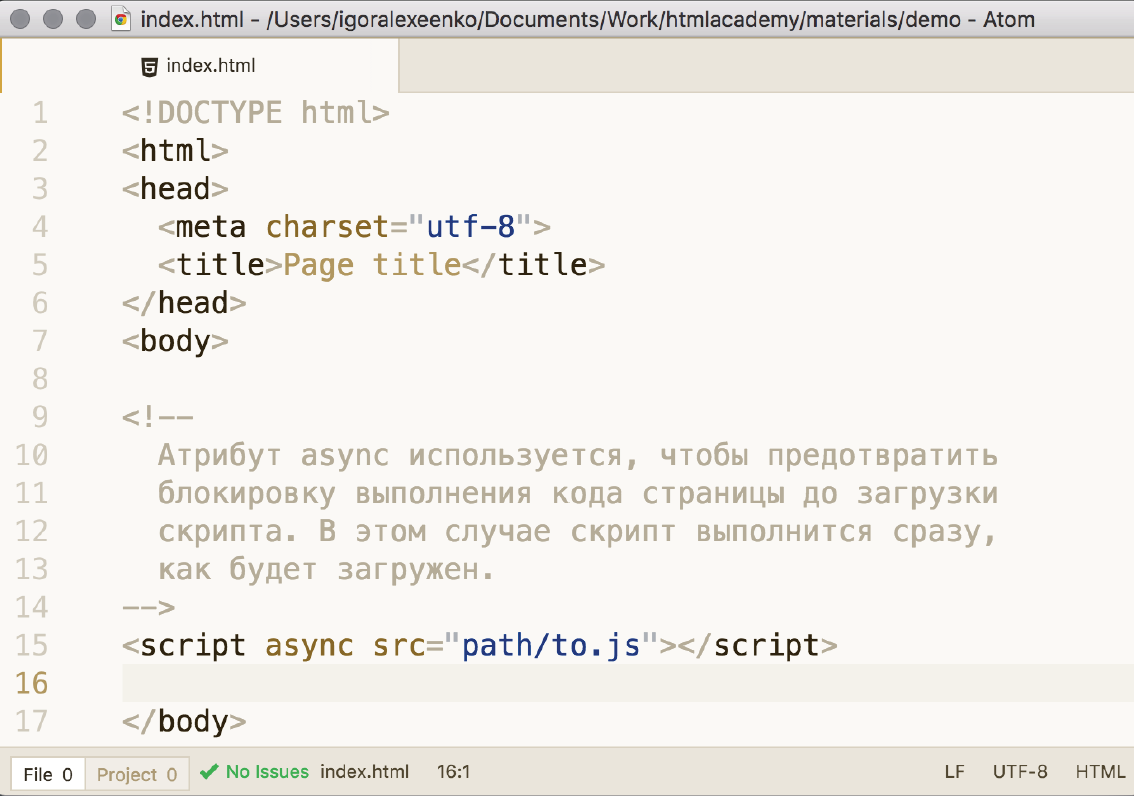
Defer /атрибут только для внешних скриптов/

скрипт выполняется асинхронно, не заставляет ждать страницу, при этом относительный порядок скриптов в defer будет сохранен. То, если после первого скрипта по счету загрузится другой он будет ждать первый. Используют defer, когда скрипты между собой связаны. Так же скрипт с defer сработает, когда весь HTML-документ будет обработан браузером.

При одновременном указании async и defer в современных браузерах будет использован только async, в IE9 только defer

Если же нужно сохранить порядок выполнения, то есть добавить несколько скриптов, которые выполнятся строго один за другим, то используется свойство script.async = false.  
Выглядит это примерно так:

function addScript(src){  
var script = document.createElement('script');  
script.src = src;  
script.async = false; // чтобы гарантировать порядок  
document.head.appendChild(script);  
}  
  
addScript('1.js'); // загружаться эти скрипты начнут сразу  
addScript('2.js'); // выполнятся, как только загрузятся  
addScript('3.js'); // но, гарантированно, в порядке 1 -> 2 -> 3



# Директива use strict

Необходима для переведения кода в режим полного соответствия современному стандарту(IS5).

Ставится в начале скрипта:

“use strict” или ‘use strict’

IE9- игнорируют use strict.

IE8 поддерживает ES3(старая версия стандарта JS). Новые возможности стандарта можно добавить в этот бразуер, подключив библиотеку ES5 shim, а именно скрипты es5-shim.js и es5-sham.js из неё.

# Модули

Код можно дробить на отдельные модули. По сути, модуль — это инструкция (*statement*), которая вызывается неявно — посредством создания файла и выполнения его с помощью интерпретатора ES (прямо, при «запуске» файла программистом, или косвенно, в результате импорта другим модулем). В ES6 есть чёткое соотношение: один файл — один модуль. Каждый модуль имеет отдельную область видимости (*Lexical environment*) — т. е. все объявления переменных, функций и классов не будут доступны за пределами модуля (файла), если не экспортированы явно. На верхнем уровне модуля (т. е. вне других инструкций и выражений) можно использовать операторы **import** для импорта других модулей и их экспортируемых сущностей, и **export** для экспорта собственных сущностей модуля.

## Экспорт и два способа импорта

Поставьте export перед тем, что вы хотите экспортировать. Такая операция сделает это импортируемым куда угодно:

export const pi = 3.14;

export const e = 2.718;

export const square = (x) => {

return x \* x;

};

export const surfaceArea = (r) => {

return 4 \* pi \* square(r);

};

Импортируйте специфичные штуки таким способом:

import { surfaceArea, square } from './math';

const surfaceOfMars = surfaceArea(3390);

const surfaceOfMercury = surfaceArea(2440);

const yearSquared = square(2017);

'./math' означает "из файла math.js, расположенного в той же (текущей) папке".

Или импортируйте всё:

import \* as mathematics from './math';

const surfaceOfMars = mathematics.surfaceArea(3390);

const surfaceOfMercury = mathematics.surfaceArea(2440);

const yearSquared = mathematics.square(2017);

Это значит: "импортировать весь модуль и назвать его mathematics в этом модуле". Вот почему к импортированным сущностям обращение происходит через mathematics вот так: mathematics.surfaceArea.

**Экспорт по умолчанию**

Вы можете сделать одну позицию экспортируемой по умолчанию.

const pi = 3.14;

const e = 2.718;

const square = (x) => {

return x \* x;

};

const surfaceArea = (r) => {

return 4 \* pi \* square(r);

};

export default surfaceArea;

Импортирование чего-то, что было экспортировано по умолчанию, выполняется просто:

import surfaceArea from './math';

const surfaceOfMars = surfaceArea(3390);

Основной алгоритм, по которому нужно анализировать файл с кодом, над которым вы сейчас работаете.

Внимательно изучите все импорты, описанные в начале файла. Так вы узнаете, какие модули и функции доступны внутри вашего файла (не считая глобальных функций и модулей, которые доступны и без импорта, например, Math).

Попробуйте классифицировать импортируемые функции. Если импорт выглядит так from './..., то есть содержит ./, значит импортируется модуль, содержимое которого находится в текущей файловой системе. Это автоматически означает несколько вещей. Первое: вы всегда можете открыть этот файл и посмотреть, что там написано. Второе: вы не сможете импортировать этот модуль в другой среде (ведь этого файла там нет).

Если from 'name' содержит только имя, без **./** в начале, значит модуль подгружается либо из стандартной библиотеки **nodejs**, либо из установленных пакетов. Визуально невозможно отличить одно от другого. Попробуйте загуглить имя таким способом: "nodejs name". Если в выдаче будет ссылка на официальную документацию, значит это модуль **nodejs**; если на репозиторий **npm** — значит, это обычный пакет, который почти наверняка лежит на гитхабе, что можно проверить таким запросом: "github js name", где "name" это имя пакета.

Объединение кода, расположенного в разных модулях, происходит через:

1. Экспорт чего-то из модуля.

## Оператор export

Оператор **export** позволяет экспортировать сущности модуля, чтобы они были доступны из других модулей. У каждого модуля есть неявный объект [[Exports]], в котором хранятся ссылки на все экспортируемые сущности, а ключом является идентификатор сущности (например, имя переменной). Это очень напоминает module.exports из модульной системы NodeJS, но [[Exports]] всегда объект, и его нельзя получить напрямую. Единственный способ его изменить — использовать оператор export.

Этот оператор имеет несколько модификаций, рассмотрим все возможные случаи.

### Экспорт объявляемой сущности

По сути, это обычное объявление переменной, функции или класса, с ключевым словом "**export**" перед ним.

export var variable;

export const CONSTANT = 0;

export let scopedVariable = 20;

export function func(){};

export class Foo {};

В данном случае система экспортов ES6 удобнее, чем в NodeJS, где пришлось бы сначала объявить сущность, а потом добавить её в объект *module.exports*.

var variable;

exports.variable = variable;

const CONSTANT = 0;

exports.CONSTANT = CONSTANT;

...

Но есть и гораздо более важное различие между этими двумя системами. В NodeJS свойству объекта *exports* присваивается значение выражения. В ES6 оператор **export** добавляет в [[*Exports*]] ссылку (или привязку, *binding*) на объявленную сущность. Это значит, что [[*Exports*]].<*имя сущности*> всегда будет возвращать текущее значение этой сущности.

export var bla = 10; // [[Exports]].bla === 10

bla = 45; // [[Exports]].bla === 45

### Экспорт уже объявленной сущности

Здесь всё то же самое, только экспортируем мы сущность, которая уже была объявлена выше. Для этого применяются фигурные скобки после ключевого слова **export**, в которых через запятую указываются все сущности (ну, их идентификаторы — например, имя переменной), которые необходимо экспортировать.

var bar = 10;

function foo() {}

export { bar, foo };

С помощью ключевого слова "**as**" можно «переименовать» сущность при экспорте (точнее будет сказать, изменить ключ для [[*Exports*]]).

var bar = 10;

function foo() {}

export { bar as bla, foo as choo };

Для такого вида экспорта также верно, что [[*Exports*]] хранит у себя лишь ссылку на сущность, даже в случае «переименования».

var bar = 10;

export { bar as bla }; // [[Exports]].bla === 10

bar = 42; // [[Exports]].bla === 42

### Экспорт по умолчанию

Этот вариант использования **export** отличается от двух описанных выше, и, на мой взгляд, он немного нелогичен. Заключается он в использовании после **export** ключевого слова **default**, после которого может идти одно из трех: выражение, объявление функции, объявление класса.

export default 42;

export default function func() {}

export default class Foo {}

Каждый из этих трех вариантов использования добавляет в [[*Exports*]] свойство с ключом «default». Экспортирование по умолчанию *выражения* (первый пример, «export default 42;») — единственный случай при использовании **export**, когда значением свойства [[*Exports*]] становится не ссылка на какую-либо сущность, а значение выражения. В случае же экспорта по умолчанию функции (не анонимной, естественно) или класса — они будут объявлены в области видимости модуля, а [[*Exports*]].default будет ссылкой на эту сущность.

1. Импорт в другой модуль.

## Оператор import

Экспортированное по умолчанию свойство считается «главным» в этом модуле. Его импорт осуществляется с помощью оператора import следующей модификации:

import <любое имя> from '<путь к модулю>';

В этом вся польза дефолтного экспорта — при импорте можно назвать его, как угодно.

// sub.js

export default class Sub {};

// main.js

import Base from './sub.js'; // И да, иногда это может сбить столку, поэтому лучше всё же использовать имя модуля

Импорт обычных экспортируемых свойств выглядит немного иначе:

// file1.js

export let bla = 20;

// file2.js

import { bla } from './file1.js'; // нужно точно указать имя сущности

// file3.js

import { bla as scopedVariable } from './file1.js'; // но можно переименовать

Рассмотрим модуль «file2.js». Оператор import получает объект [[Exports]] импортируемого модуля ('file1.js'), находит в нём нужное свойство («bla»), а после создаёт привязку идентификатора "bla" к значению [[Exports]].bla. Т. е., точно так же, как и [[Exports]].bla, bla в модуле «file2.js» всегда будет возвращать текущее значение переменной «bla» из модуля «file1.js». Равно как и scopedVariable из модуля «file3.js».

// count.js

export let counter = 0;

export function inc() {

++counter;

}

// main.js

import { counter, inc } from './count.js';

console.log(counter); // 0

inc();

console.log(counter); // 1

### Импорт всех экспортируемых свойств

import \* as sub from './sub.js';

По сути, таким образом мы получаем копию [[Exports]] модуля «sub.js».

### Включение модуля без импорта

Иногда бывает нужно, чтобы файл просто запустился.

import './worker';

### Реэкспорт

Повторный экспорт модулем свойства, которое он импортирует из другого модуля. Осуществляется это оператором export.

// main.js

export { something } from './another.js';

Два замечания, которые тут стоит сделать: первое — something после реэкспорта НЕ становится доступной внутри модуля main.js, для этого придётся сделать отдельный импорт (уж не знаю, почему так, видимо, чтобы сохранить дух оператора export); и второе — система ссылок работает и тут: модуль, который импортирует из «main.js» something, будет получать актуальное значение переменной something в «another.js»;

Так же можно зареэкспортить все свойства из другого модуля.

export \* from './another';

Однако важно помнить, что если вы объявите в своём модуле экспорт с таким же именем, как у реэкспортного, то он затрёт реэскпортное.

// sub.js

export const bla = 3, foo = 4;

// another.js

export const bla = 5;

export \* from './sub';

// main.js

import \* as another from './another';

console.log(another); // { bla: 5, foo: 4 }

Это решается переименованием конфликтных свойств при реэкспорте.

И, почему-то, синтаксиса для реэкспорта дефолтных свойств у export нет, но можно сделать так:

export { default as sub } from './sub';

### Несколько слов о свойствах импортов

**Поддержка циклических ссылок**

Собственно, вся эта пляска с биндингами вместо присвоения нужна для нормального разрешения циклических ссылок. Т. к. это не значение (которое может быть и undefined), а ссылка на то место, где когда-то что-то будет лежать — ничего не упадёт, даже если цикл.

**Импорты всплывают**

Импорты «всплывают» наверх модуля.

// sub.js

console.log('sub');

// main.js

console.log('main');

import './sub.js';

Если запустить main.js, то в консоль сначала выведется «sub», и только потом «main» — именно из-за всплытия импортов.

**Экспорт по умолчанию — это ещё не конец**

Вот такие конструкции вполне допустимы.

// jquery.js

export default function $() {}

export const AUTHOR = "Джон Резиг";

// main.js

import $, { AUTHOR } from 'jquery.js';

И вообще, по сути, default — это просто ещё один именованный экспорт.

import Base from './Base';

То же самое, что и

import { default as Base } from './Base';

## Задачи

**myMathModule.js**

В файле myMathModule.js:

1. Создайте функцию getTriangleArea, которая принимает два аргумента h и b и вычисляет площадь треугольника по формуле A = 1/2 \* h \* b, где h — высота, а b — основание треугольника.
2. Экспортируйте функцию.

**solution.js**

В файле solution.js:

1. Импортируйте getTriangleArea из модуля myMathModule.
2. Создайте функцию, которая принимает аргумент **n** и возвращает площадь треугольника высотой **n** и основанием **n2/2**. Используйте функцию square.
3. Экспортируйте созданную функцию по умолчанию.

myMathModule.js

export const getTriangleArea = (h, b) => {

const area = (h \* b) / 2;

return area;

};

**solution.js**

import { getTriangleArea } from './myMathModule';

const solution = n => getTriangleArea**(n, square(n) / 2);**

export default solution;

# Именованные функциональные выражения

Специально для работы с рекурсией в JavaScript существует особое расширение функциональных выражений, которое называется «Named Function Expression» (сокращённо NFE) или, по-русски, *«именованное функциональное выражение»*.

## Named Function Expression

Обычное функциональное выражение:

var f = function(...) { /\* тело функции \*/ };

Именованное с именем sayHi:

var f = function sayHi(...) { /\* тело функции \*/ };

Что же это за имя, которое идёт в дополнение к f, и зачем оно?

Имя функционального выражения (sayHi) имеет особый смысл. Оно доступно только изнутри самой функции.

Это ограничение видимости входит в стандарт JavaScript и поддерживается всеми браузерами, кроме IE8-.

Например:

var f = function sayHi(name) {

alert( sayHi ); // изнутри функции - видно (выведет код функции)

};

alert( sayHi ); // снаружи - не видно (ошибка: undefined variable 'sayHi')

Кроме того, имя NFE нельзя перезаписать:

var test = function sayHi(name) {

sayHi = "тест"; // попытка перезаписи

alert( sayHi ); // function... (перезапись не удалась)

};

test();

В режиме use strict код выше выдал бы ошибку.

Как правило, имя NFE используется для единственной цели – позволить изнутри функции вызвать саму себя.

## Пример использования

NFE используется в первую очередь в тех ситуациях, когда функцию нужно передавать в другое место кода или перемещать из одной переменной в другую.

**Внутреннее имя позволяет функции надёжно обращаться к самой себе, где бы она ни находилась.**

Вспомним, к примеру, функцию-факториал из задачи Вычислить факториал:

function f(n) {

return n ? n \* f(n - 1) : 1;

};

alert( f(5) ); // 120

Попробуем перенести её в другую переменную g:

function f(n) {

return n ? n \* f(n - 1) : 1;

};

var g = f;

f = null;

alert( g(5) ); // запуск функции с новым именем - ошибка при выполнении!

Ошибка возникла потому что функция из своего кода обращается к своему старому имени f. А этой функции уже нет, f = null.

Для того, чтобы функция всегда надёжно работала, объявим её как Named Function Expression:

var f = function factorial(n) {

return n ? n\*factorial(n-1) : 1;

};

var g = f; // скопировали ссылку на функцию-факториал в g

f = null;

alert( g(5) ); // 120, работает!

В браузере IE8- создаются две функции

Как мы говорили выше, в браузере IE до 9 версии имя NFE видно везде, что является ошибкой с точки зрения стандарта.

…Но на самом деле ситуация ещё забавнее. Старый IE создаёт в таких случаях целых две функции: одна записывается в переменную f, а вторая – в переменную factorial.

Например:

var f = function factorial(n) { /\*...\*/ };

// в IE8- false

// в остальных браузерах ошибка, т.к. имя factorial не видно

alert( f === factorial );

Все остальные браузеры полностью поддерживают именованные функциональные выражения.

Устаревшее специальное значение arguments.callee

Если вы давно работаете с JavaScript, то, возможно, знаете, что раньше для этой цели также служило специальное значение arguments.callee.

Если это название вам ни о чём не говорит – всё в порядке, читайте дальше, мы обязательно обсудим его [в отдельной главе](http://learn.javascript.ru/arguments-pseudoarray#arguments-callee).

Если же вы в курсе, то стоит иметь в виду, что оно официально исключено из современного стандарта. А NFE – это наше настоящее.

**Итого**

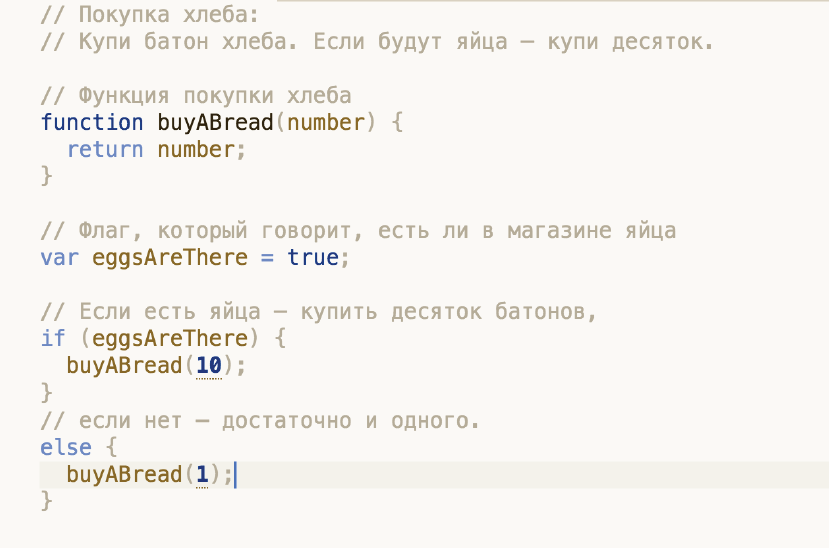
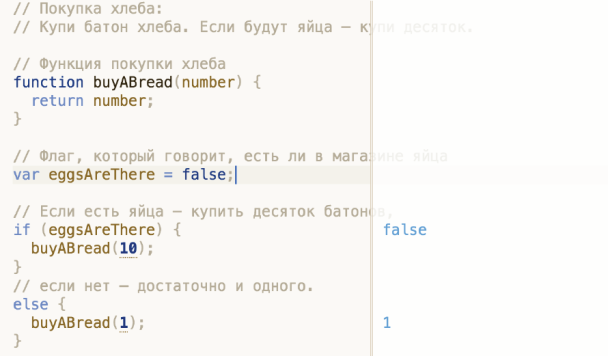
Если функция задана как Function Expression, ей можно дать имя.

Оно будет доступно только внутри функции (кроме IE8-).

Это имя предназначено для надёжного рекурсивного вызова функции, даже если она записана в другую переменную.

Обратим внимание, что с Function Declaration так поступить нельзя. Такое «специальное» внутреннее имя функции задаётся только в синтаксисе Function Expression.

# 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

...

}

Можно и просто передать уже готовое логическое значение, к примеру, заранее вычисленное в переменной:

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

if (cond) {

...

}

**Неверное условие, else**

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

var year = prompt('Введите год появления стандарта 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.

На заметку:

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

access = age > 14;

hiJack = (friend == "Jack") ? "Hi, Jack!" : "Hi";

приведет к выводу "**Hi, Jack**" если переменная **friend** будет иметь значение **"Jack"** или к выводу **"Hi"** при любом другом значении.

const absValue = (num === 0 || num > 0) ? num : -num; // при num >= 0 => num, при num < 0 => меняют знак на +;

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».

// функция-предикат определяет, является ли число num четным

const isEven = num => num % 2 === 0;

// функция увеличивает полученное число num на 10

const increaseNum = num => num + 10;

const num = 6;

const result = isEven(increaseNum(num) + 1) ? num + 1 : 1 - num; // -5

**Несколько операторов „?“**

Последовательность операторов '?' позволяет вернуть значение в зависимости не от одного условия, а от нескольких.

Например:

var age = prompt('возраст?', 18);

var message = (age < 3) ? 'Здравствуй, малыш!' :

(age < 18) ? 'Привет!' :

(age < 100) ? 'Здравствуйте!' :

'Какой необычный возраст!';

alert( message );

Поначалу может быть сложно понять, что происходит. Однако, внимательно приглядевшись, мы замечаем, что это обычная последовательная проверка!

Вопросительный знак проверяет сначала age < 3, если верно – возвращает 'Здравствуй, малыш!', если нет – идет за двоеточие и проверяет age < 18. Если это верно – возвращает 'Привет!', иначе проверка age < 100 и 'Здравствуйте!'… И наконец, если ничего из этого не верно, то 'Какой необычный возраст!'.

То же самое через if..else:

if (age < 3) {

message = 'Здравствуй, малыш!';

} else if (age < 18) {

message = 'Привет!';

} else if (age < 100) {

message = 'Здравствуйте!';

} else {

message = 'Какой необычный возраст!';

}

**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');

}

При чтении кода глаз идёт вертикально и конструкции, занимающие несколько строк, с понятной вложенностью, воспринимаются гораздо легче. Возможно, вы и сами почувствуете, пробежавшись глазами, что синтаксис с if более прост и очевиден чем с оператором '?'.

Смысл оператора '?' – вернуть то или иное значение, в зависимости от условия. Пожалуйста, используйте его по назначению, а для выполнения разных веток кода есть if.

## Задачи

Выведется ли alert?

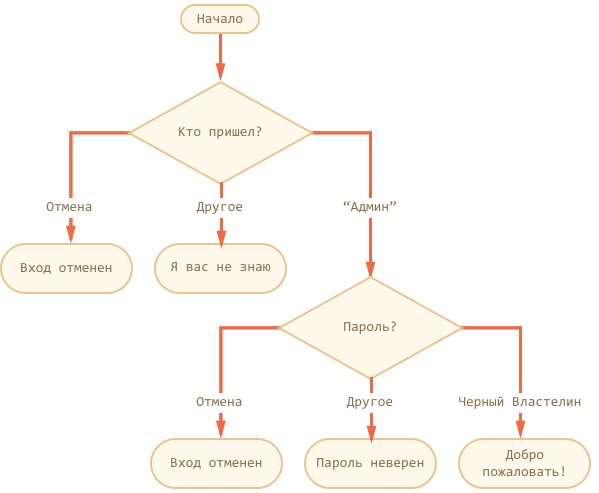
if ("0") {

alert( 'Привет' );

}

**Да, выведется,** т.к. внутри if стоит строка "0".

Любая строка, кроме пустой (а здесь она не пустая), в логическом контексте является true.



var userName = prompt('Кто пришёл?', '');

if (userName == 'Админ') {

var pass = prompt('Пароль?', '');

if (pass == 'Чёрный Властелин') {

alert( 'Добро пожаловать!' );

} else if (pass == null) { // (\*)

alert( 'Вход отменён' );

} else {

alert( 'Пароль неверен' );

}

} else if (userName == null) { // (\*\*)

alert( 'Вход отменён' );

} else {

alert( 'Я вас не знаю' );

}

Обратите внимание на проверку if в строках (\*) и (\*\*). Везде, кроме Safari, нажатие «Отмена» возвращает null, а вот Safari возвращает при отмене пустую строку, поэтому в браузере Safari можно было бы добавить дополнительную проверку на неё.

if (a + b < 4) {

result = 'Мало';

} else {

result = 'Много';

}

Тоже самое: result = (a + b < 4) ? 'Мало' : 'Много';

var message;

if (login == 'Вася') {

message = 'Привет';

} else if (login == 'Директор') {

message = 'Здравствуйте';

} else if (login == '') {

message = 'Нет логина';

} else {

message = '';

}

var message = (login == 'Вася') ? 'Привет' :

(login == 'Директор') ? 'Здравствуйте' :

(login == '') ? 'Нет логина' :

'';

# Циклические алгоритмы

Алгоритм — законченный и упорядоченный набор действий, которые нужно предпринять, чтобы достигнуть прогнозируемого результата.

Для многократного повторения одного участка кода – предусмотрены циклы.

## Рекурсия

В теле функции могут быть вызваны другие функции для выполнения подзадач.

Частный случай подвызова – когда функция вызывает сама себя. Это называется *рекурсией*.

Рекурсия используется для ситуаций, когда выполнение одной сложной задачи можно представить как некое действие в совокупности с решением той же задачи в более простом варианте.

## Степень pow(x, n) через рекурсию

В качестве первого примера использования рекурсивных вызовов – рассмотрим задачу возведения числа x в натуральную степень n.

Её можно представить как совокупность более простого действия и более простой задачи того же типа вот так:

pow(x, n) = x \* pow(x, n - 1)

То есть, xn = x \* xn-1.

Например, вычислим pow(2, 4), последовательно переходя к более простой задаче:

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

На шаге 1 нам нужно вычислить pow(2,3), поэтому мы делаем шаг 2, дальше нам нужно pow(2,2), мы делаем шаг 3, затем шаг 4, и на нём уже можно остановиться, ведь очевидно, что результат возведения числа в степень 1 – равен самому числу.

Далее, имея результат на шаге 4, он подставляется обратно в шаг 3, затем имеем pow(2,2) – подставляем в шаг 2 и на шаге 1 уже получаем результат.

Этот алгоритм на JavaScript:

function pow(x, n) {

if (n != 1) { // пока n != 1, сводить вычисление pow(x,n) к pow(x,n-1)

return x \* pow(x, n - 1);

} else {

return x;

}

}

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

**Или /рекурсивный процесс / Декларативное (лат. разъяснять) программирование –** описывает что делать, чтобы найти факториал .Вы разъясняете: я хочу, чтобы факториал n был n раз факториалом n-1. **/:**

const factorial = (n) => {

if (n === 1) {

return 1;

}

else {

return n \* factorial(n-1);

}

}

const answer = factorial(3);

**Требования рекурсии:**

1. Простой базовый случай или терминальный сценарий. Простыми словами, когда остановиться. В нашем примере это была 1: мы остановили вычисление факториала, когда достигли 1.
2. Правило двигаться по рекурсии, углубляться. В нашем случае, это было n \* factorial(n-1).

**Ожидание умножения**

Ничего не умножается, пока мы спускаемся к базовому случаю factorial(1). Затем мы начинаем подниматься обратно, по одному шагу.

factorial(3);

3 \* factorial(2);

3 \* 2 \* factorial(1);

3 \* 2 \* 1;

3 \* 2;

6;

**Говорят, что «функция pow *рекурсивно вызывает сама себя*» до n == 1.**

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 рекурси и в браузерах ограничена, точно можно рассчитывать на 10000 вложенных вызовов, но некоторые интерпретаторы допускают и больше.

Итак, рекурсию используют, когда вычисление функции можно свести к её более простому вызову, а его – ещё к более простому, и так далее, пока значение не станет очевидно.

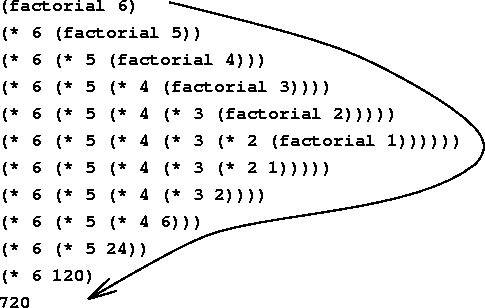
## Отличие итеративного процесса от рекурсивного

Главная фишка в аккумуляторе или, иными словами, в запоминании.

Рекурсивный процесс постоянно говорит «я это запомню и потом посчитаю» на каждом шаге рекурсии. «Потом» наступает в самом конце.

Когда рекурсивный процесс считает факториал 6, то ему нужно запомнить 5 чисел чтобы посчитать их в самом конце, когда уже никуда не деться и рекурсивно двигаться вниз больше нельзя.

Когда мы находимся в очередном вызове функции, то где-то снаружи этого вызова в памяти хранятся эти запомненные числа.



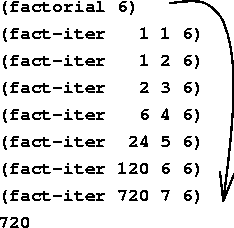
*Тут прямо физически видно, как растет использование памяти: процессу нужно запоминать все больше и больше чисел*

Рекурсивный процесс — это процесс с отложенным вычислением.

Итеративный процесс постоянно говорит «я сейчас посчитаю все что можно и продолжу» на каждом шаге рекурсии. Ему не нужно ничего запоминать вне вызова, он всегда считает все в первый возможный момент, и каждый шаг рекурсии может существовать в изоляции от прошлых, потому что вся информация передается из шага в шаг.

Когда итеративный процесс считает факториал 6, то ему не нужно запоминать числа. Нужно лишь считать и передавать результат дальше, в новый вызов.

Когда мы находимся в очередном вызове функции, снаружи этого вызова в памяти ничего не нужно запоминать.



Тут видно, что использования памяти не растет

Вся информация о вычислениях, обо всём, что запоминает программа в любой конкретный момент (вычислениях, константах, функциях), называется состоянием. Множество проблем в программировании рождается из необходимости справиться с состоянием.

Суть **итеративного процесса** — вычисление с фиксированным количеством состояний.

*Итеративный процесс с рекурсией:*

const factorial = (n) => {

const iter = (counter, acc) => {

if (counter === 1) {

return acc;

}

return iter(counter-1, counter \* acc);

}

return iter(n, 1);

}

Идея:

1. Считаем от n до 1
2. Берем число из предыдущего шага
3. Умножаем это число на текущее число
4. Передаем его в новый шаг
5. Когда counter достигает 1, число передаётся из предыдущего шага в ответ

Нам нужно с чего-то начать, потому что шаг 2 требует число из предыдущего шага, и мы начинаем с 1, потому что тогда n\*1 будет просто n:

return iter(n, 1);

Вот так выглядят вызовы iter, когда происходит вычисление 3!:

iter(3, 1); // iter(3-1, 1\*3);

iter(2, 3); // iter(2-1, 2\*3);

iter(1, 6); // counter === 1, return 6

6;

**Итеративный процесс** в целом:

1. Определить начальное состояние. В нашем случае мы делаем первый вызов iter с n и 1. Это начальное состояние.
2. ​Проверить терминальный сценарий. Мы убеждаемся, что counter это 1 и останавливаем рекурсию, когда он достигает значения 1.
3. ​Определить новое состояние. Это то, как продвигается процесс. В нашем случае мы делаем ещё один вызов iter с уменьшенным counter и умноженным accumulator. Два этих новых числа определяют новое состояние.
4. Повторить шаг 2.​

**Резюме:**

Рекурсия — это когда что-то содержит себя в своём описании.

Рекурсивный процесс — это процесс обработки данных с отложенными вычислениями.

Итеративный процесс — это процесс вычисления, когда состояние может быть описано фиксированным количеством значений.

## Контекст выполнения, стек

Теперь мы посмотрим, как работают рекурсивные вызовы. Для этого мы рассмотрим, как вообще работают функции, что происходит при вызове.

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

Контекст выполнения – это служебная информация, которая соответствует текущему запуску функции. Она включает в себя локальные переменные функции и конкретное место в коде, на котором находится интерпретатор.

Например, для вызова pow(2, 3) из примера выше будет создан контекст выполнения, который будет хранить переменные x = 2, n = 3. Мы схематично обозначим его так:

Контекст: { x: 2, n: 3, строка 1 }

Далее функция pow начинает выполняться. Вычисляется выражение n != 1 – оно равно true, ведь в текущем контексте n=3. Поэтому задействуется первая ветвь if :

function pow(x, n) {

if (n != 1) { // пока n != 1 сводить вычисление pow(x,n) к pow(x,n-1)

return x \* pow(x, n - 1);

} else {

return x;

}

}

Чтобы вычислить выражение x \* pow(x, n-1), требуется произвести запуск pow с новыми аргументами.

**При любом вложенном вызове JavaScript запоминает текущий контекст выполнения в специальной внутренней структуре данных – «**стеке контекстов**».**

Затем интерпретатор приступает к выполнению вложенного вызова.

В данном случае вызывается та же pow, однако это абсолютно неважно. Для любых функций процесс одинаков.

Для нового вызова создаётся свой контекст выполнения, и управление переходит в него, а когда он завершён – старый контекст достаётся из стека и выполнение внешней функции возобновляется.

Разберём происходящее с контекстами более подробно, начиная с вызова (\*):

function pow(x, n) {

if (n != 1) { // пока n!=1 сводить вычисление pow(..n) к pow(..n-1)

return x \* pow(x, n - 1);

} else {

return x;

}

}

alert( pow(2, 3) ); // (\*)

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, строка 1 }

Выполнение в этом контексте продолжается, пока не встретит вложенный вызов в строке 3.

pow(2, 2)

В строке 3 происходит вложенный вызов pow с аргументами x=2, n=2. Текущий контекст сохраняется в стеке, а для вложеннного вызова создаётся новый контекст (выделен жирным ниже):

Контекст: { x: 2, n: 3, строка 3 }

Контекст: { x: 2, n: 2, строка 1 }

Обратим внимание, что контекст включает в себя не только переменные, но и место в коде, так что когда вложенный вызов завершится -- можно будет легко вернуться назад.

Слово «строка» здесь условно, на самом деле, конечно, запомнено более точное место в цепочке команд.

pow(2, 1)

Опять вложенный вызов в строке 3, на этот раз – с аргументами x=2, n=1. Создаётся новый текущий контекст, предыдущий добавляется в стек:

Контекст: { x: 2, n: 3, строка 3 }

Контекст: { x: 2, n: 2, строка 3 }

Контекст: { x: 2, n: 1, строка 1 }

На текущий момент в стеке уже два старых контекста.

Выход из pow(2, 1).

При выполнении pow(2, 1), в отличие от предыдущих запусков, выражение n != 1 будет равно false, поэтому сработает вторая ветка if..else:

function pow(x, n) {

if (n != 1) {

return x \* pow(x, n - 1);

} else {

return x; // первая степень числа равна самому числу

}

}

Здесь вложенных вызовов нет, так что функция заканчивает свою работу, возвращая 2. Текущий контекст больше не нужен и удаляется из памяти, из стека восстанавливается предыдущий:

Контекст: { x: 2, n: 3, строка 3 }

Контекст: { x: 2, n: 2, строка 3 }

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, строка 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.

Обратим внимание на требования к памяти. Рекурсия приводит к хранению всех данных для неоконченных внешних вызовов в стеке, в данном случае это приводит к тому, что возведение в степень n хранит в памяти n различных контекстов.

Реализация возведения в степень 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.**

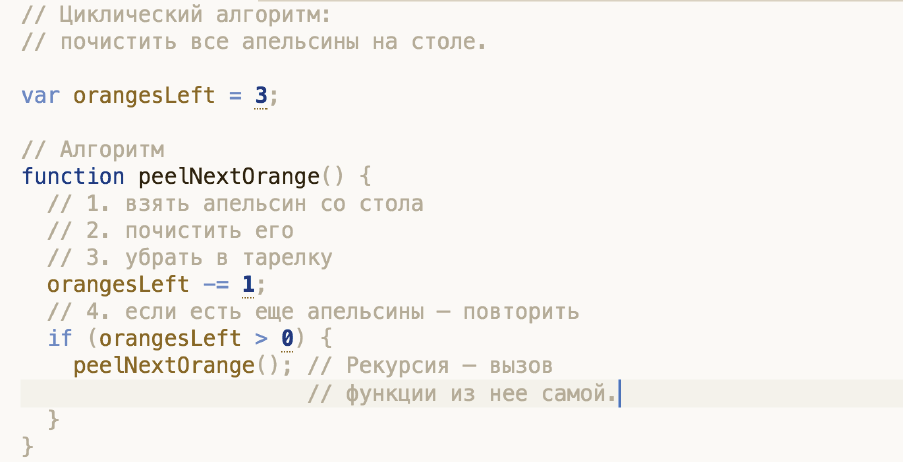
Но переделка рекурсии в цикл может быть нетривиальной, особенно когда в функции, в зависимости от условий, используются различные рекурсивные подвызовы, когда ветвление более сложное.

**Итого**

Рекурсия – это когда функция вызывает сама себя, как правило, с другими аргументами.

Существуют много областей применения рекурсивных вызовов. Здесь мы посмотрели на один из них – решение задачи путём сведения её к более простой (с меньшими аргументами), но также рекурсия используется для работы с «естественно рекурсивными» структурами данных, такими как HTML-документы, для «глубокого» копирования сложных объектов.

Есть и другие применения, с которыми мы встретимся по мере изучения JavaScript.





## Задачи

**Напишите функцию sumTo(n), которая для данного n вычисляет сумму чисел от 1 до n**

Решение через цикл:

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? Это очевидно. 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 операций сложения столько же, но нет дополнительных вычислительных затрат на организацию вложенных вызовов.

Рекурсия в данном случае работает медленнее всех.

P.P.S. Существует ограничение глубины вложенных вызовов, поэтому рекурсивный вызов sumTo(100000) выдаст ошибку.

**Вычислить факториал**

*Факториа́л числа* – это число, умноженное на «себя минус один», затем на «себя минус два» и так далее, до единицы. Обозначается n!

Определение факториала можно записать как:

n! = n \* (n - 1) \* (n - 2) \* ...\*1

Примеры значений для разных n:

1! = 1

2! = 2 \* 1 = 2

3! = 3 \* 2 \* 1 = 6

4! = 4 \* 3 \* 2 \* 1 = 24

5! = 5 \* 4 \* 3 \* 2 \* 1 = 120

Задача – написать функцию factorial(n), которая возвращает факториал числа n!, используя рекурсивный вызов.

alert( factorial(5) ); // 120

Подсказка: обратите внимание, что n! можно записать как n \* (n-1)!. Например: 3! = 3\*2! = 3\*2\*1! = 6

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) сведётся к 1\*factorial(0), будет дополнительный шаг рекурсии.

Итеративно:

function sFact(a) {

var x = 1;

for (var i = 2; i <= a; i++)

x = x \* i;

return x;

}

alert(sFact(prompt()));

Рекурсивно:

function rFact(num) {

if (num === 0)

return 1;

else

return num \* rFact(num - 1);

}

alert(rFact(prompt()));

**Реализуйте (с использованием рекурсивного процесса) функцию sequenceSum, которая находит сумму последовательности целых чисел. Последовательность задается двумя значениями: begin - начало последовательности, end - конец последовательности. Например: begin = 2 и end = 6 дают нам такую последовательность 2, 3, 4, 5, 6. Сумма такой последовательности будет: 20.**

**1-й вариант:**

const sequenceSum = (begin, end) => {

**if** (begin > end) {

**return** NaN;

} **else** **if** (begin === end) {

**return** begin;

}

**return** begin + sequenceSum(begin + 1, end);

}

2-й вариант:

const sequenceSum = (begin, end) => {

return begin == end ? begin :

begin > end ? NaN :

end + sequenceSum(begin, end - 1);

};

**Реализуйте тело функции smallestDivisor, используя итеративный процесс. Эта функция должна находить наименьший делитель заданного числа.**

**Доп. условия: число, передаваемое в функцию, больше нуля; делитель должен быть больше единицы, за исключением случая, когда аргументом является единица (наименьшим делителем которой является также единица).**

**Например, наименьший делитель числа 15 это 3.**

const smallestDivisor = (num) => {

const i = (number, divider) => {

if (number === 1) {

return number;

}

if (number % divider === 0) {

return divider;

}

return i(number, divider + 1);

};

return i(num, 2);

};

# Конструкция switch

Конструкция switch заменяет собой сразу несколько if.

Она представляет собой 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).
* Если ни один case не совпал – выполняется (если есть) вариант default.

При этом case называют *вариантами switch*.

**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' );

}

Здесь оператор switch последовательно сравнит a со всеми вариантами из case.

Сначала 3, затем – так как нет совпадения – 4. Совпадение найдено, будет выполнен этот вариант, со строки alert('В точку!') и далее, до ближайшего break, который прервёт выполнение.

**Если break нет, то выполнение пойдёт ниже по следующим case, при этом остальные проверки игнорируются.**

Пример без break:

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: // b + 1 = 1 = a;

alert( 1 );

break;

default:

alert('нет-нет, выполнится вариант выше')

}

var x = 3; //объявляем переменную по которой будем делать выборку

var myResult = ""; //объявляем переменную в которую будем записывать результат

switch (x) {

case 1: //если х равен единице то записываем

myResult = ("Икс равен единице!"); // в переменную myResult фразу "Икс равен единице!"

break;

case 2:

myResult = ("Икс равен двум!");

break;

case 3:

myResult = ("Икс равен трем!");

break;

default:

myResult = ("Икс не равен 1, 2 или 3!");

}

document.write (myResult); //Выводим результат в веб-документе

## 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!');

alert('Немного ошиблись, бывает.');

break;

default:

alert('Странный результат, очень странный');

}

При case 3 выполнение идёт со строки (\*), при case 5 – со строки (\*\*).

## 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' );

default:

alert('Неизвестное значение: ' + arg)

}

Что оно выведет при вводе числа 0? Числа 1? 2? 3?

Подумайте, выпишите свои ответы, исходя из текущего понимания работы switch и *потом* читайте дальше…

* При вводе 0 выполнится первый alert, далее выполнение продолжится вниз до первого break и выведет второй alert('Два'). Итого, два вывода alert.
* При вводе 1 произойдёт то же самое.
* При вводе 2, switch перейдет к case '2', и сработает единственный alert('Два').
* **При вводе 3, switch перейдет на default.** Это потому, что prompt возвращает строку '3', а не число. Типы разные. Оператор switch предполагает строгое равенство ===, так что совпадения не будет.

## Задачи

Напишите if..else, соответствующий следующему switch:

switch (browser) {

case 'IE':

alert( 'О, да у вас IE!' );

break;

case 'Chrome':

case 'Firefox':

case 'Safari':

case 'Opera':

alert( 'Да, и эти браузеры мы поддерживаем' );

break;

default:

alert( 'Мы надеемся, что и в вашем браузере все ок!' );

}

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!');

}

Перепишите код с использованием одной конструкции switch:

var a = +prompt('a?', '');

if (a == 0) {

alert( 0 );

}

if (a == 1) {

alert( 1 );

}

if (a == 2 || a == 3) {

alert( '2,3' );

}

var a = +prompt('a?', '');

switch (a) {

case 0:

alert( 0 );

break;

case 1:

alert( 1 );

break;

case 2:

case 3:

alert( '2,3' );

break;

}

Обратите внимание: break внизу не обязателен, но ставится по «правилам хорошего тона».

Допустим, он не стоит. Есть шанс, что в будущем нам понадобится добавить в конец ещё один case, например case 4, и мы, вполне вероятно, забудем этот break поставить. В результате выполнение case 2/case 3 продолжится на case 4 и будет ошибка.

# Циклы while, for

Циклы — это повторяющиеся блоки кода.

## Цикл while

**while** - выполнение кода пока истинно заданное условие.

Имеет вид:

while (условие) {

// код, тело цикла

}

Пока условие верно – выполняется код из тела цикла.

/ **Некурсивный факториал** / **Императивная** (лат. "imperare "командовать") **функция** - она как бы определение (трактование, характеристика) факториала. Вы приказываете чётко передвигаться по шагам — умножать это на это, пока идёт отсчёт и запоминаются какие-то числа. Она декларирует, что такое факториал / :

const factorial = (n) => {

let counter = 1;

let result = 1;

while (counter <= n) {

result =\* counter; //result = result \* counter;

counter++; //counter = counter + 1;

}

return result;

}

Например, цикл ниже выводит i пока i < 3:

var i = 0;

while (i < 3) {

alert( i );

i++;

}

Или:

varx = 1;//объявляем и инициализируем переменную, которую   
//мы будем использовать как условие цикла  
while(x <= 5) {  
 console.log(" x= " + x);  
 x = x + 1;  
}

В результате выполнения данного кода произойдет пятикратный вывод значения переменной х.

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):

var i = 3;

while (i) { // при i, равном 0, значение в скобках будет false и цикл остановится

alert( i );

i--;

}

## **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);

Или:

varx = 5**;** //установка начального значения

do {    //начало выполнения цикла

console.log(" x=" + x); //вывод данных в консоль

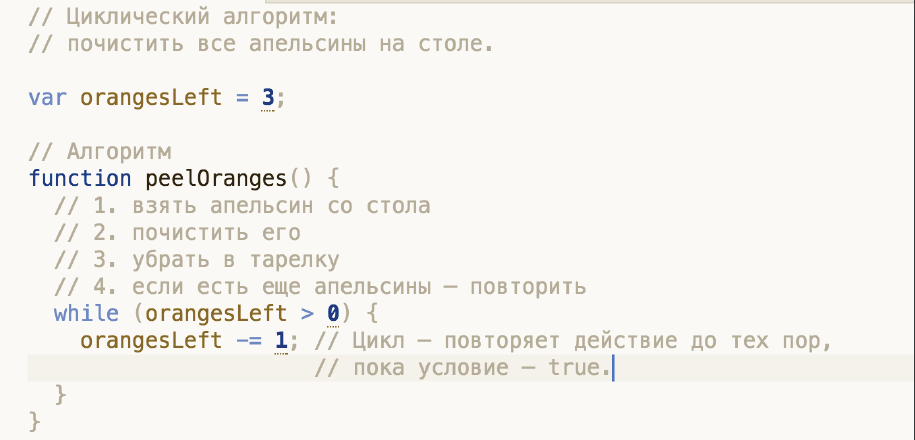
x = x - 1;           //уменьшение значения на 1

} while (x >= 1);     //проверка условия

В данном случае программа будет выводить в консоль значение переменной "х" в диапазоне от 5 до 1, результат будет "54321"

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.

Единственная разница между этими циклами — это будет ли проверяться условная конструкция до первой итерации (while) или после первой итерации (do..while).





## Цикл for

**for** - выполнение кода указанное количество раз. Чаще всего применяется цикл for.

Синтаксис команды выглядит так :

for(*переменная* ***=*** *начальное\_значение****;***//первым задается начальное значение переменной-счетчика

*переменная* ***<=*** *конечное\_значение****;***    
//после этого задается условие, которое проверяется при выполнении каждого цикла

переменная **=** переменная **+** шаг)      
//и наконец последним задается операция изменения состояния переменной-счетчика

{  
исполняемый код  
}

Т.о.:

for (начало; условие; шаг) {

// ... тело цикла ...

}

Например:

for (i = 1; i <= 5; i = i + 1**)** {  
 console.log("i = " + i);  
}

const factorial = (n) => {

let result = 1;

// initialization↓ condition↓ update↓

for (let counter = 1; counter <= n; counter++) {

result \*= counter;

}

return result;

}

Или пример цикла, который выполняет alert(i) для i от 0 до 2 включительно (до 3):

var i;

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

alert( i );

}

Здесь:

Начало: i=0.

Условие**:** i<3.

Шаг**:** i++.

Тело**:** alert(i), т.е. код внутри фигурных скобок (они не обязательны, если только одна операция)

Цикл выполняется так:

1. Начало: i=0 выполняется один-единственный раз, при заходе в цикл.
2. Условие: i<3 проверяется перед каждой итерацией и при входе в цикл, если оно нарушено, то происходит выход.
3. Тело: alert(i).
4. Шаг: i++ выполняется после *тела* на каждой итерации, но перед проверкой условия.
5. Идти на шаг 2.

Иными словами, поток выполнения: начало → (если условие → тело → шаг) → (если условие → тело → шаг) → … и так далее, пока верно условие.

В цикле также можно определить переменную:

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

alert(i); // 0, 1, 2

}

Эта переменная будет видна и за границами цикла, в частности, после окончания цикла i станет равно 3.

## Пропуск частей for

Любая часть for может быть пропущена.

Например, можно убрать начало. Цикл в примере ниже полностью идентичен приведённому выше:

var i = 0;

for (; i < 3; i++) {

alert( i ); // 0, 1, 2

}

Можно убрать и шаг:

var i = 0;

for (; i < 3;) {

alert( i );

// цикл превратился в аналог while (i<3)

}

А можно и вообще убрать всё, 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.

for **(**переменнаяinобъект) **{**  
 исполняемый код  
}

Например приведенный ниже код выведет все свойства объекта **car**

for **(**iincar**) {** console.log(car[i] + ' ');}

## Прерывание цикла: break

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

Например, следующий код подсчитывает сумму вводимых чисел до тех пор, пока посетитель их вводит, а затем – выдаёт:

var sum = 0;

while (true) {

var value = +prompt("Введите число", '');

if (!value) break; // (\*)

sum += value;

}

alert( 'Сумма: ' + sum );

Директива break в строке (\*), если посетитель ничего не ввёл, полностью прекращает выполнение цикла и передаёт управление на строку за его телом, то есть на alert.

Вообще, сочетание «бесконечный цикл + break» – отличная штука для тех ситуаций, когда условие, по которому нужно прерваться, находится не в начале-конце цикла, а посередине.

## continue

Директива continue прекращает выполнение *текущей итерации* цикла.

Она – в некотором роде «младшая сестра» директивы break: прерывает не весь цикл, а только текущее выполнение его тела, как будто оно закончилось.

Её используют, если понятно, что на текущем повторе цикла делать больше нечего.

Например, цикл ниже использует continue, чтобы не выводить чётные значения:

for (var i = 0; i < 10; i++) {

if (i % 2 == 0) continue;

alert(i);

}

for (i = 1; i < 10; i = i + 1) {  
  if (i == 5) {  
    continue;  
  }  
  console.log('i= ' + i);  
}

В данном случае continue помешает выполниться выводу в консоль при значении переменной равном 5, все остальные значения будут напечатаны.

Для чётных i срабатывает continue, выполнение тела прекращается и управление передаётся на следующий проход for.

Директива continue позволяет обойтись без скобок

Цикл, который обрабатывает только нечётные значения, мог бы выглядеть так:

for (var i = 0; i < 10; i++) {

if (i % 2) { //i % 2 >= 1, а значит значение преобразуется к true;

alert( i );

}

}

С технической точки зрения он полностью идентичен. Действительно, вместо continue можно просто завернуть действия в блок if. Однако, мы получили дополнительный уровень вложенности фигурных скобок. Если код внутри if более длинный, то это ухудшает читаемость, в отличие от варианта с continue.

**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;

Впрочем, как уже говорилось ранее, оператор вопросительный знак '?' не стоит использовать таким образом. Это – всего лишь ещё одна причина, почему для проверки условия предпочтителен if.

## Метки для break/continue

Бывает нужно выйти одновременно из нескольких уровней цикла.

Например, внутри цикла по i находится цикл по j, и при выполнении некоторого условия мы бы хотели выйти из обоих циклов сразу:

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

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

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

// если отмена ввода или пустая строка -

// завершить оба цикла

if (!input) break outer; // (\*)

}

}

alert('Готово!');

В коде выше для этого использована *метка*.

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 её на отдельную строку:

outer:

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

Вызов break outer ищет ближайший внешний цикл с такой меткой и переходит в его конец.

В примере выше это означает, что будет разорван самый внешний цикл и управление перейдёт на alert.

Директива continue также может быть использована с меткой, в этом случае управление перепрыгнет на следующую итерацию цикла с меткой.

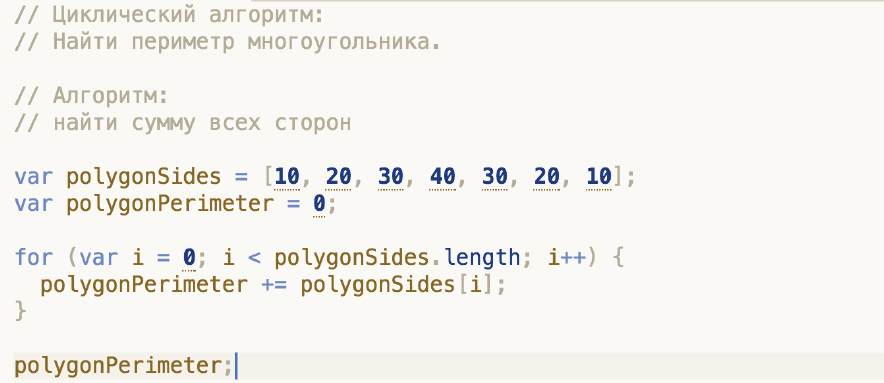
**Итого**

JavaScript поддерживает три вида циклов:

* while – проверка условия перед каждым выполнением.
* do..while – проверка условия после каждого выполнения.
* for – проверка условия перед каждым выполнением, а также дополнительные настройки.

## Цикл for и массивы

Массив —сложный тип данных, позволяющий хранить несколько связанных значений вместе. Доступ к элементам массива производится по ключу.

 160

## Задачи

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

//если будет –-i, то выведет 2, 1, 0

var i = 0;

while (++i < 5) alert( i ); //приоритет операций(инкремент, декремент выше чем у ариф. операций)

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. **От 1 до 4**

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. **От 1 до 5**

**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.

**При помощи цикла for выведите чётные числа от 2 до 10.**

for (var i = 2; i <= 10; i++) {

if (i % 2 == 0) {

alert( i );

}

}

**Напишите цикл, который предлагает prompt ввести число, большее 100. Если посетитель ввёл другое число – попросить ввести ещё раз, и так далее. Цикл должен спрашивать число пока либо посетитель не введёт число, большее 100, либо не нажмёт кнопку Cancel (ESC).**

var number;

do {

number = prompt("Введите число больше 100?", 0);

} while (num <= 100 && num != null);

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.

**Вывести простые числа**

Натуральное число, большее 1, называется *простым*, если оно ни на что не делится, кроме себя и 1.

Другими словами, n>1 – простое, если при делении на любое число от 2 до n-1 есть остаток.

**Создайте код, который выводит все простые числа из интервала от 2 до 10.** Результат должен быть: 2,3,5,7.

P.S. Код также должен легко модифицироваться для любых других интервалов.

**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

}

**Напишите функцию smallestDivisor. Она должна находить наименьший целый делитель числа. Поведение у функции должно быть таким же, как в предыдущем уроке, но реализация — код функции — должно быть другим. На этот раз реализуйте императивный итеративный процесс, что означает:**

* **не используйте рекурсию**
* **используйте переменные**
* **используйте цикл while**

const smallestDivisor = (num) => {

if (num < 1) {

return NaN;

}

if (num === 1) {

return num;

}

let divisor = 2;

while (num % divisor !== 0) {

divisor = divisor + 1;

}

return divisor;

};

**Напишите функцию isPrime. Она принимает число и возвращает true, если число является простым, и false в ином случае. Простое число — целое положительное число, имеющее ровно два различных натуральных делителя — единицу и самого себя. Например, 7 — простое число, потому что делится без остатка только на 1 и на себя. 2017 — другое простое число. Используйте цикл for и арифметические мутаторы.**

const isPrime = (number) => {

if (number < 2) {

return false;

}

for (let i = 2; i < number; i++) {

if (number % i === 0) {

return false;

}

}

return true;

};

const isPrime = (num) => {

if (num >= 0) {

switch (num) {

case 0:

case 1:

return false;

case 2:

return true;

default:

for (var i = 2; i < num; i++) {

if (num % i === 0)

return false;

}

return true;

}

} else {

return false;

}

}

# Выражения и инструкции

Выражением (**expression**) является любой корректный блок кода, который возвращает значение.

Концептуально, существуют два типа выражений: те которые **присваивают переменной значение**, и те, которые **вычисляют** **значение без его присваивания**.

Выражение x = 7  является примером выражения первого типа. Данное выражение использует *оператор* = для присваивания переменной x значения 7. Само выражение также равняется 7.

Код 3 + 4 является примером выражения второго типа. Данное выражение использует  *оператор* "+" для сложения чисел 3 и 4 без присваивания переменной полученного результата 7.

Ниже getAnswer() — это вызов функции — другое выражение. Этот вызов возвращает значение, то есть этот вызов функции выразится в значение:

const y = getAnswer();

Ниже пример выражения, которое состоит из нескольких подвыражений, и пошаговый процесс превращения каждого выражения по порядку, пока целое выражение не превратится в одно значение:

12 + square(7 + 5) + square(square(2));

12 + square(12) + square(square(2));

12 + 144 + square(square(2));

12 + 144 + square(4);

12 + 144 + 16;

156 + 16;

172;

Все выражения в JavaScript делятся на следующие категории:

* **Арифметические**: вычисляются в число, например: 3.14159 (Используют арифметические операторы).
* **Строковые**: вычисляются в текстовую строку, например: "Fred" или "234" (Используют строковые операторы).
* **Логические**: вычисляются в true или false (Используют [логические операторы](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Expressions_and_Operators#Logical_operators)).
* **Основные выражения**: Базовые ключевые слова и основные выражения в JavaScript.
* **Левосторонние выражения**: Значениям слева назначаются значения справа.

**Основные выражения**

Базовые ключевые слова и основные выражения в JavaScript.

## Оператор this

Используйте ключевое слово this для указания на текущий объект. В общем случае this указывает на вызываемый объект, которому принадлежит данный метод. Используйте this следующим образом:

this["propertyName"]

this.propertyName

Предположим, функция validate выполняет проверку свойства value некоторого объекта; задан объект, а также верхняя и нижняя граница величины данного свойства:

function validate(obj, lowval, hival){

if ((obj.value < lowval) || (obj.value > hival))

alert("Неверное значение!");

}

Вы можете вызвать функцию validate для обработчика события onChange для каждого элемента формы, используя this для указания на элемент формы, как это показано в следующем примере:

<B>Введите число от 18 до 99:</B>

<INPUT TYPE="text" NAME="age" SIZE=3

onChange="validate(this, 18, 99);">

## Оператор группировки

Оператор группировки "скобки" ( ) контролирует приоритет вычисления выражений. Например, вы можете переопределить порядок - "умножение и деление, а потом сложение и вычитание", так чтобы, например, чтобы сложение выполнялось до умножения:

var a = 1;

var b = 2;

var c = 3;

// обычный порядок

a + b \* c // 7

// выполняется, как обычно, так

a + (b \* c) // 7

// теперь поменяем порядок

// сложение до умножения

(a + b) \* c // 9

// что эквивалентно следующему

a \* c + b \* c // 9

## Упрощенный синтаксис создания массивов и генераторов

Упрощенный синтаксис - экспериментальная возможность JavaScript, которая возможно будет добавлена в будущие версии ECMAScript. Есть 2 версии синтаксиса:

[for (x of y) x]

Упрощенный синтаксис для массивов.

(for (x of y) y)

Упрощенный синтаксис для генераторов.

Упрощенные синтаксисы существуют во многих языках программирования и позволяют вам быстро собирать новый массив, основанный на существующем. Например:

[for (i of [ 1, 2, 3 ]) i\*i ];

// [ 1, 4, 9 ]

var abc = [ "A", "B", "C" ];

[for (letters of abc) letters.toLowerCase()];

// [ "a", "b", "c" ]

## Левосторонние выражения

Значениям слева назначаются значения справа.

## new

Вы можете использовать [оператор new](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/new) для создания экземпляра объекта пользовательского типа или одного из встроенных объектов. Используйте оператор new следующим образом:

var objectName = new objectType([param1, param2, ..., paramN]);

## super

[Ключевое слово](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/super) используется, чтобы вызывать функции родительского объекта. Это полезно и с [классами](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Classes) для вызова конструктора родителя, например.

super([arguments]); // вызывает конструктор родителя. super.functionOnParent([arguments]);

## Оператор расширения

[Оператор расширения](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Spread_operator) позволяет выражению расширяться в местах с множеством аргументов (для вызовов функций) или множестве элементов (для массивов).

**Пример:** Сегодня, если у вас есть массив и вы хотите создать новый с существующей частью первого, то литерального синтаксиса массива уже не достаточно, и вы должны писать императивный (без вариантов) код, используя комбинацию push, splice, concat и т.д. Но с этим оператором код становится более коротким:

var parts = ['shoulder', 'knees'];

var lyrics = ['head', ...parts, 'and', 'toes'];

Похожим образом оператор работает с вызовами функций:

function f(x, y, z) { }

var args = [0, 1, 2];

f(...args);

JavaScript различает выражения (**expressions**) и инструкции (**statements**). Инструкция — это (грубо говоря) команда, действие.

if, while, for, const — примеры инструкций. Они производят или контролируют действия, но не превращаются в значения.

# Взаимодействие с пользователем: alert, prompt, confirm

## alert

Синтаксис:

alert(сообщение)

alert выводит на экран окно с сообщением и приостанавливает выполнение скрипта, пока пользователь не нажмёт «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 разберётся с окном. В данном случае – пока не нажмёт на «OK».

## 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 ввёл посетитель – строку или специальное значение null, если ввод отменён.

Единственный браузер, который не возвращает null при отмене ввода – это Safari. При отсутствии ввода он возвращает пустую строку. Предположительно, это ошибка в браузере.

Если нам важен этот браузер, то пустую строку нужно обрабатывать точно так же, как и null, т.е. считать отменой ввода.

Как и в случае с alert, окно prompt модальное.

var years = prompt('Сколько вам лет?', ‘Например,100’);

alert('Вам ' + years + ' лет!')

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 );

# Функции

Функции  в JavaScript бывают встроенными, например alert() и пользовательскими, которые программист создает сам.

Функции в JavaScript являются объектами, и как следствие могут, например, присваиваться переменным, передаваться другим функциям, можно присваивать значения их свойствам и вызывать их методы.

• Со свойствами name, length <Число передаваемых в функцию параметров()> и prototype.

• Может использоваться как любой другой объект: храниться в переменных, других объектах, передаваться как аргумент и возвращать как значение.

• Функциям можно задавать свойства и методы (!)

• В отличие от других объектов, функцию можно вызвать

Детерминированная функция всегда возвращает одинаковое значение при определённом вводе (аргументы).

const surfaceAreaCalculator = (radius) => {

return 4 \* 3.14 \* square(radius);

}

surfaceAreaCalculator это детерминированная функция.

Недетерминированная функция *не* всегда будет возвращать одинаковое значение при определённом вводе.

Функция, которая возвращает погоду на данный момент для какой-нибудь координаты — недетрминированная: погода всегда меняется, поэтому мы не можем быть уверены, какой ответ выдаст функция.

Другой пример — генератор случайных чисел:

Math.random(); // 0.6822304980945362

Math.random(); // 0.34656303876811245

Math.random(); // 0.44983037125501646

Побочные эффекты: то, как функция меняет внешний мир.

surfaceAreaCalculator не имеет никаких побочных эффектов. Она ничего не меняет за пределами своих границ.

Функция console.log имеет побочный эффект: она что-то выводит на экран.

Другой пример:

let a = 0;

const f = () => {

a = a + 1;

return true;

}

f();

Функция f меняет значение глобальной переменной a. Эта переменная, с точки зрения функции, находится во внешнем мире, а функция f это меняет. Поэтому f имеет побочный эффект.

И f, и console.log имеют побочные эффекты, но они детерминированные. f всегда возвращает true, а console.log всегда возвращает undefined.

То, что функции возвращают, не имеет ничего общего с тем, как они влияют на внешний мир.

console.log("Hello!"); // prints "Hello!", but returns undefined

Чем меньше побочных эффектов имеет функция, тем лучше.

Когда функция детерминированная и не имеет побочных эффектов, мы называем её "чистой" функцией. Чистые функции:

* проще читать
* проще отлаживать
* проще тестировать
* не зависят от порядка, в котором они вызываются
* просто запустить параллельно (одновременно)

Чистые функции независимы от времени. Недетерминизм и побочные эффекты добавляют понятие времени. Если функция зависит от чего-то, что может случиться, а может не случиться и меняет что-то за пределами своих границ, то она неожиданно становится зависимой от времени.

Функции способ использовать код повторно.

function Name(arg:type):type нотация описания 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, ещё 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.

function printAmount(amt) {

console.log( amt.toFixed( 2 ) );

}

function formatAmount() {

return "$" + amount.toFixed( 2 );

}

var amount = 99.99;

printAmount( amount \* 2 ); // "199.98"

amount = formatAmount();

console.log( amount ); // "$99.99"

function getIt() {

return this.x;

}

var obj1 = { get : getIt, x : 1 };

var obj2 = { get : getIt, x : 2 };

obj1.x; // 1

obj2.x; // 2

obj1.get(); // 1

obj2.get(); // 2

var obj = {

base : 13,

average: function (x,y) {

return (this.base + x + y)/3;

}

};

## Объявление

Пример объявления функции:

// const <name> = (<argument>) => {

// return <expressions>;

// };

var num = (value) => {

return value + 1;

};

num(2) // 3

Если у вас функция-однострочник, то можно использовать сокращенный синтаксис:

// const <name> = (<argument>) => <expressions>;

var identity = value => value;

В коде выше мы опустили фигурные скобки и слово return, а так же скобки вокруг аргумента (это можно делать только если у функции один аргумент).

Формальными параметрами функции называются имена переменных в *определении функции*. Например у функции

const f = (a, b) => a - b; // формальные параметры — это a и b.

Фактически параметры — это то, что было *передано в функцию* в момент вызова. Например, если предыдущую функцию вызвать так f(5, z), где const z = 8, то фактическими параметрами являются 5 и z. Результатом этого вызова будет число -3, а внутри функции на момент конкретного вызова параметр a становится равным 5, а b становится равным 8.

**const** f = x => x \* x;

**const** y = 5;

console.log(f(y)); *// 25*

**const** z = 3;

console.log(f(z)); *// 9*

Как видите, нет никакой связи между именами формальных и фактических параметров. Более того, у фактических параметров вообще может не быть имен, как в примере выше, где мы сразу передали число 5 в функцию. Что имеет значение, так это **позиция**. Во время вызова функции параметры должны передаваться в правильном порядке, и только тогда функция отработает, как предполагается.

**const** f = (a, b) => a - b;

**const** x = 5;

**const** y = 8;

console.log(f(x, y)); *// -3*

console.log(f(y, x)); *// 3*

**Return**

Вызов оператора return приводит к изменению течения программы. Последующие инструкции никогда не будут выполнены:

**const** identity = (value) => {

**return** value;

**const** a = 3 + 5; *// этот код никогда не будет достигнут*

};

Или:

const func = (num) => {

return num \* num \* num;

return num \* num \* num \* num;

return num \* num; }

console.log(func(3)); //27

Строка кода называется инструкуцией, в JavaScript инструкции должны заканчиваться точкой с запятой.

Фигурные скобки создают блок. В JavaScript и других языках вы часто сталкиваетесь с блоками. Они создают группу инструкций, таким способом мы понимаем где функция начинается и заканчивается.

**const** pi = () => {

**return** 3.14;

}

**const** multi = (a, b) => {

**return** Math.round(a \* a + b \* b);

}

multi(pi(), 4);

Попробуем вычислить результат выражения multi(pi(), 4) используя подстановочную модель вычислений.

multi(pi(), 4);

*// Подстановка `pi`*

multi(3.14, 4)

*// Подстановка `multi`*

Math.round(3.14 \* 3.14 + 4 \* 4);

*// Редукции*

Math.round(9.8596 + 4 \* 4);

Math.round(9.8596 + 16);

Math.round(25.8596);

*// Подстановки внутри `round` и вычисления мы не видим,*

*// потому что это встроенная функция*

25;

или <Устаревший синтаксис чуть ниже, предпочтительным является () => {}. Кроме синтаксической разницы есть и семантическая.>

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.

При определении функции после ключевого слова может не быть имени.  Например мы можем объявить функцию из прошлого шага вот таким способом:

var printText = function(a) {document.write(a);};

Функция может быть сразу вызвана с необходимым входным параметром. В следующем примере функция сразу будет вызвана - выведет в документ фразу "Hello World!". Если бы функция возвращала какой-либо результат, он бы был записан в переменную printText.

var printText = function(a) {document.write(a);}("Hello World!");

Для того чтобы обозначить что функция является самовызываемой, ее заключают в скобки. Это не требование стандарта, однако  так делают для большей очевидности.

var printText = (function(a) {document.write(a);}("Hello World!"));

Функция, имеющая имя, может вызывать саму себя. Такие функции называются рекурсивными.

Например, в приведенном ниже примере функция с помощью рекурсивного вызова вычисляет факториал:

function factorial(x) { //Объявление функции

if (x <= 1) return 1; //Проверка условия окончания расчета

return x \* factorial(x-1); //Вызов этой же функции с уменьшенным на 1 аргументом

}

Важное замечание: в рекурсивных функциях нужно не забывать про **условие прекращения рекурсии**, иначе код "зациклится".

В примере выше, таким условием будет проверка, что x <= 1

## 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() {

// переменные i,j не будут уничтожены по окончании цикла

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

var j = i \* 2;

}

alert( i ); // i=3, последнее значение i, при нём цикл перестал работать

alert( j ); // j=4, последнее значение j, которое вычислил цикл

}

**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»*.**

В примере выше переменная userName – глобальная.

Делайте глобальными только те переменные, которые действительно имеют общее значение для вашего проекта, а нужные для решения конкретной задачи – пусть будут локальными в соответствующей функции.

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.

При use strict такой код привёл бы к ошибке, но без него 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 одна функция неожиданно поменяла глобальную переменную, которую использует другая. И поди разберись, кто и когда её поменял, не самая приятная ошибка для отладки.

## 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 + ': ' + text);

}

showMessage('Маша', 'Привет!');

showMessage('Маша', 'Как дела?');

**Параметры копируются в локальные переменные функции**.

Например, в коде ниже есть внешняя переменная from, значение которой при запуске функции копируется в параметр функции с тем же именем. Далее функция работает уже с параметром:

function showMessage(from, text) {

from = '\*\*' + from + '\*\*'; // меняем локальную переменную from

alert( from + ': ' + text );

}

var from = "Маша";

showMessage(from, "Привет"); // \*\*Маша\*\*: Привет

alert( from ); // старое значение from без изменений, в функции была изменена копия

## Аргументы по умолчанию

Функцию можно вызвать с любым количеством аргументов.

Если параметр не передан при вызове – он считается равным undefined.

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

showMessage("Маша");

При этом можно проверить, и если параметр не передан – присвоить ему 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 логическом контексте является false.

Если аргументов передано больше, чем надо, например showMessage("Маша", "привет", 1, 2, 3), то ошибки не будет. Но, чтобы получить такие «лишние» аргументы, нужно будет прочитать их из специального объекта arguments, который мы рассмотрим в главе Псевдомассив аргументов "arguments".

## 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 b2 – 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 неё доходит управление – функция завершается и значение передается обратно.

Вызовов return может быть и несколько, например:

function checkAge(age) {

if (age > 18) {

return true; (1)

} else {

return confirm('Родители разрешили?');

}

}

var age = prompt('Ваш возраст?');

if (checkAge(age)) {

alert( 'Доступ разрешен' ); // true (1)

} else {

alert( 'В доступе отказано' );

}

function divideOneTo(x) {  
  if (x != 0) {  
    return 1/x;  
  } else {  
  return "А на ноль делить нельзя!";  
  }  
}

Однако, использование ключевого слова **return**не является обязательным. Если в теле функции его нет, то после выполнения всех команд интерпретатор закончит выполнение функции. Значение, которое возвращает эта функция будет "**undefined**".

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 функции

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](http://jquery.com) есть функция $, во фреймворке [Prototype](http://prototypejs.org/) – функция $$, а в библиотеке [LoDash](http://lodash.com/) очень активно используется функция с названием из одного символа подчеркивания \_.

**Итого**

Объявление функции имеет вид:

function имя(параметры, через, запятую) {

код функции

}

* Передаваемые значения копируются в параметры функции и становятся локальными переменными.
* Параметры функции копируются в её локальные переменные.
* Можно объявить новые локальные переменные 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, но лучше, если функция использует только локальные переменные:

* 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 друг для друга меняют 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.

Следующая функция возвращает true, если параметр age больше 18. В ином случае она задаёт вопрос посредством вызова confirm и возвращает его результат.

function checkAge(age) {

if (age > 18) {

return true;

} else {

// ...

return confirm('Родители разрешили?');

}

}

Если убрать else эта функция работать точно так же.

## Задачи

function checkAge(age) {

if (age > 18) {

return true;

} else {

return confirm('Родители разрешили?');

}

}

Перепишите функцию, чтобы она делала то же самое, но без if, в одну строку.

function checkAge(age) {

return (age > 18) ? true : confirm('Родители разрешили?');

}

function checkAge(age) {

return (age > 18) || confirm('Родители разрешили?');

}

**Напишите функцию min(a,b), которая возвращает меньшее из чисел a,b.**

**Вариант решения с использованием if:**

function min(a, b) {

if (a < b) {

return a;

} else {

return b;

}

}

Вариант решения с оператором '?':

function min(a, b) {

return a < b ? a : b;

}

P.S. Случай равенства a == b здесь отдельно не рассматривается, так как при этом неважно, что возвращать.

**Создайте страницу, которая запрашивает x и n, а затем выводит результат pow(x,n).**

**P.S. В этой задаче функция обязана поддерживать только натуральные значения n, т.е. целые от 1 и выше**.

function pow(x, n) {

var result = x;

for (var i = 1; i < n; i++) {

result \*= x;

}

return result;

}

var x = prompt("x?", '');

var n = prompt("n?", '');

if (n <= 1) {

alert('Степень ' + n +

'не поддерживается, введите целую степень, большую 1'

);

} else {

alert( pow(x, n) )

}

# Функциональные выражения

В JavaScript функция является значением, таким же как строка или число.

Как и любое значение, объявленную функцию можно вывести, вот так:

function sayHi() {

alert( "Привет" );

}

alert( sayHi ); // выведет код функции

Обратим внимание на то, что в последней строке после sayHi нет скобок. То есть, функция не вызывается, а просто выводится на экран.

**Функцию можно скопировать в другую переменную:**

function sayHi() { // (1)

alert( "Привет" );

}

var func = sayHi; // (2)

func(); // Привет // (3)

sayHi = null;

sayHi(); // ошибка (4)

1. Объявление (1) как бы говорит интерпретатору "создай функцию и помести её в переменную sayHi
2. В строке (2) мы копируем функцию в новую переменную func. Ещё раз обратите внимание: после sayHi нет скобок. Если бы они были, то вызов var func = sayHi() записал бы в func результат работы sayHi() (кстати, чему он равен? правильно, undefined, ведь внутри sayHi нет return).
3. На момент (3) функцию можно вызывать и как sayHi() и как func()
4. …Однако, в любой момент значение переменной можно поменять. При этом, если оно не функция, то вызов (4) выдаст ошибку.

Обычные значения, такие как числа или строки, представляют собой *данные*. А функцию можно воспринимать как *действие*.

Это действие можно запустить через скобки (), а можно и скопировать в другую переменную, как было продемонстрировано выше.

## Объявление Function Expression

Существует альтернативный синтаксис для объявления функции, который ещё более наглядно показывает, что функция – это всего лишь разновидность значения переменной.

Он называется «Function Expression» (функциональное выражение) и выглядит так:

var f = function(параметры) {

// тело функции

};

Например:

var sayHi = function(person) {

alert( "Привет, " + person );

};

sayHi('Вася');

## Сравнение с Function Declaration

«Классическое» объявление функции, о котором мы говорили до этого, вида function имя(параметры) {...}, называется в спецификации языка «Function Declaration».

* *Function Declaration* – функция, объявленная в основном потоке кода.
* *Function Expression* – объявление функции в контексте какого-либо выражения, например присваивания.

Несмотря на немного разный вид, по сути две эти записи делают одно и то же:

// Function Declaration

function sum(a, b) {

return a + b;

}

// Function Expression

var sum = function(a, b) {

return a + b;

}

Оба этих объявления говорят интерпретатору: "объяви переменную sum, создай функцию с указанными параметрами и кодом и сохрани её в sum".

**Основное отличие между ними: функции, объявленные как Function Declaration, создаются интерпретатором до выполнения кода.**

Поэтому их можно вызвать *до* объявления, например:

sayHi("Вася"); // Привет, Вася

function sayHi(name) {

alert( "Привет, " + name );

}

А если бы это было объявление Function Expression, то такой вызов бы не сработал:

sayHi("Вася"); // ошибка!

var sayHi = function(name) {

alert( "Привет, " + name );

}

Это из-за того, что JavaScript перед запуском кода ищет в нём Function Declaration (их легко найти: они не являются частью выражений и начинаются со слова function) и обрабатывает их.

А Function Expression создаются в процессе выполнения выражения, в котором созданы, в данном случае – функция будет создана при операции присваивания sayHi = function...

Как правило, возможность Function Declaration вызвать функцию до объявления – это удобно, так как даёт больше свободы в том, как организовать свой код.

Можно расположить функции внизу, а их вызов – сверху или наоборот.

## Условное объявление функции

В некоторых случаях «дополнительное удобство» Function Declaration может сослужить плохую службу.

Например, попробуем, в зависимости от условия, объявить функцию sayHi по-разному:

var age = +prompt("Сколько вам лет?", 20);

if (age >= 18) {

function sayHi() {

alert( 'Прошу вас!' );

}

} else {

function sayHi() {

alert( 'До 18 нельзя' );

}

}

sayHi();

Function Declaration при use strict видны только внутри блока, в котором объявлены. Так как код в учебнике выполняется в режиме use strict, то будет ошибка.

А что, если использовать Function Expression?

var age = prompt('Сколько вам лет?');

var sayHi;

if (age >= 18) {

sayHi = function() {

alert( 'Прошу Вас!' );

}

} else {

sayHi = function() {

alert( 'До 18 нельзя' );

}

}

sayHi();

Или даже так:

var age = prompt('Сколько вам лет?');

var sayHi = (age >= 18) ?

function() { alert('Прошу Вас!'); } :

function() { alert('До 18 нельзя'); };

sayHi();

Оба этих варианта работают правильно, поскольку, в зависимости от условия, создаётся именно та функция, которая нужна.

## Анонимные функции

Взглянем ещё на один пример – функцию ask(question, yes, no) с тремя параметрами:

question

Строка-вопрос

yes

Функция

no

Функция

Она выводит вопрос на подтверждение question и, в зависимости от согласия пользователя, вызывает функцию yes() или no():

function ask(question, yes, no) {

if (confirm(question)) yes()

else no();

}

function showOk() {

alert( "Вы согласились." );

}

function showCancel() {

alert( "Вы отменили выполнение." );

}

// использование

ask("Вы согласны?", showOk, showCancel);

Какой-то очень простой код, не правда ли? Зачем, вообще, может понадобиться такая ask?

…Оказывается, при работе со страницей такие функции как раз очень востребованы, только вот спрашивают они не простым confirm, а выводят более красивое окно с вопросом и могут интеллектуально обработать ввод посетителя. Но это всё потом, когда перейдём к работе с интерфейсом.

Здесь же обратим внимание на то, что то же самое можно написать более коротко:

function ask(question, yes, no) {

if (confirm(question)) yes()

else no();

}

ask(

"Вы согласны?",

function() { alert("Вы согласились."); },

function() { alert("Вы отменили выполнение."); }

);

Здесь функции объявлены прямо внутри вызова ask(...), даже без присвоения им имени.

**Функциональное выражение, которое не записывается в переменную, называют анонимной функцией.**

Действительно, зачем нам записывать функцию в переменную, если мы не собираемся вызывать её ещё раз? Можно просто объявить непосредственно там, где функция нужна.

Такого рода код возникает естественно, он соответствует «духу» JavaScript.

var foo = function() {

// ..

};

var x = function bar(){

// ..

};

Первое функциональное выражение, присваиваемое переменной foo, называется *анонимным* поскольку у него нет имени.

Второе функциональное выражение *именованное* (bar), несмотря на то, что является ссылкой, также присваивается переменной x. *Выражения с именованными функциями* как правило более предпочтительны, хотя *выражения с анонимными функциями* все еще чрезвычайно употребительны.

## new Function

Существует ещё один способ создания функции, который используется очень редко, но упомянем и его для полноты картины.

Он позволяет создавать функцию полностью «на лету» из строки, вот так:

var sum = new Function('a,b', ' return a+b; ');

var result = sum(1, 2);

alert( result ); // 3

То есть, функция создаётся вызовом **new Function**(params, code):

**params**

Параметры функции через запятую в виде строки.

**code**

Код функции в виде строки.

Таким образом можно конструировать функцию, код которой неизвестен на момент написания программы, но строка с ним генерируется или подгружается динамически во время её выполнения.

Пример использования – динамическая компиляция шаблонов на JavaScript, мы встретимся с ней позже, при работе с интерфейсами.

**Итого**

Функции в JavaScript являются значениями. Их можно присваивать, передавать, создавать в любом месте кода.

* Если функция объявлена в *основном потоке кода*, то это Function Declaration.
* Если функция создана как *часть выражения*, то это Function Expression.

Между этими двумя основными способами создания функций есть следующие различия:

|  |  |  |
| --- | --- | --- |
|  | **Function Declaration** | **Function Expression** |
| Время создания | До выполнения первой строчки кода. | Когда управление достигает строки с функцией. |
| Можно вызвать до объявления | Да (т.к. создаётся заранее) | Нет |
| Условное объявление в if | Не работает | Работает |

Иногда в коде начинающих разработчиков можно увидеть много Function Expression. Почему-то, видимо, не очень понимая происходящее, функции решают создавать как var func = function(), но в большинстве случаев обычное объявление функции – лучше.

**Если нет явной причины использовать Function Expression – предпочитайте Function Declaration.**

Сравните по читаемости:

// Function Expression

var f = function() { ... }

// Function Declaration

function f() { ... }

Function Declaration короче и лучше читается. Дополнительный бонус – такие функции можно вызывать до того, как они объявлены.

Используйте Function Expression только там, где это действительно нужно и удобно.

## Выражения немедленно вызываемых функций (Immediately Invoked Function Expressions (IIFEs))

Есть еще один путь выполнить выражение с функцией, на который обычно ссылаются как на *immediately invoked function expression* (IIFE):

(function IIFE(){

console.log( "Hello!" );

})();

// "Hello!"

Или:

const result = ((a, b) => a \*\* b)(5, 2); //25

Внешние ( .. ), которые окружают выражение функции (function IIFE(){ .. }) — это всего лишь нюанс грамматики JS, необходимый для предотвращения того, чтобы это выражение воспринималось как объявление обычной функции.

Последние () в конце выражения, строка })(); — это то, что и выполняет выражение с функцией, указанное сразу перед ним.

Может показаться странным, но это не так уж чужеродно, как кажется на первый взгляд. Посмотрите на сходства между foo и IIFE тут:

function foo() { .. }

// `foo` выражение со ссылкой на функцию,

// затем `()` выполняют ее

foo();

// Выражение с функцией `IIFE`,

// затем `()` выполняют ее

(function IIFE(){ .. })();

Как видите, содержимое (function IIFE(){ .. }) до ее вызова в () фактически такое же, как включение foo до его вызова после (). В обоих случаях ссылка на функцию выполняется с помощью () сразу после них.

Так как IIFE — просто функция, а функции создают *область видимости* переменных, то использование IIFE таким образом обычно происходит, чтобы объявлять переменные, которые не будут влиять на код, окружающий IIFE снаружи:

var a = 42;

(function IIFE(){

var a = 10;

console.log( a ); // 10

})();

console.log( a ); // 42

Функции IIFE также могут возвращать значения:

var x = (function IIFE(){

return 42;

})();

x; // 42

Значение 42 возвращается из выполненной IIFE функции, а затем присваивается в x.

# 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:

alert('Привет'); alert('Мир');

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

alert('Привет')

alert('Мир')

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

var a = 2

+3

alert(a); // 5

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

alert("После этого сообщения будет ошибка")

[1, 2].forEach(alert)

Детали того, как работает код выше (массивы [...] и forEach) мы скоро изучим, здесь важно то, что при установке точки с запятой после alert он будет работать корректно.

**Поэтому в JavaScript рекомендуется точки с запятой ставить. Сейчас это, фактически, общепринятый стандарт.**

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

## 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.

**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.

## Строгий режим

Для того, чтобы интерпретатор работал в режиме максимального соответствия современному стандарту, нужно начинать скрипт директивой 'use strict';

'use strict';

...

Эта директива может также указываться в начале функций. При этом функция будет выполняться в режиме соответствия, а на внешний код такая директива не повлияет.

Одно из важных изменений в современном стандарте – все переменные нужно объявлять через var. Есть и другие, которые мы изучим позже, вместе с соответствующими возможностями языка.

## Взаимодействие с посетителем

Простейшие функции для взаимодействия с посетителем в браузере:

«prompt(вопрос[, по\_умолчанию])»

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

«confirm(вопрос)»

Задать вопрос и предложить кнопки «Ок», «Отмена». Возвращает, соответственно, true/false.

«alert(сообщение)»

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 isTeaWanted = confirm("Вы хотите чаю?");

alert( "Посетитель: " + userName );

alert( "Чай: " + isTeaWanted );

## 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

Исключение – сравнение двух строк, которое осуществляется лексикографически.

Также: значения null и undefined при == равны друг другу и не равны ничему ещё. А при операторах больше/меньше происходит приведение null к 0, а undefined к NaN.

Такое поведение может привести к неочевидным результатам, поэтому лучше всего использовать для сравнения с null/undefined оператор ===. Оператор == тоже можно, если не хотите отличать null от undefined.

Например, забавное следствие этих правил для null:

alert( null > 0 ); // false, т.к. null преобразовано к 0

alert( null >= 0 ); // true, т.к. null преобразовано к 0

alert( null == 0 ); // false, в стандарте явно указано, что null равен лишь undefined

 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( 'а' > 'Я' ); // true

## 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 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

## 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.

## 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' );

}

## Функции

Синтаксис функций в JavaScript:

// function имя(список параметров) { тело }

function sum(a, b) {

var result = a + b;

return result;

}

// использование:

alert( sum(1, 2) ); // 3

 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 копируются в локальные переменные a, b.

 Функция без return считается возвращающей undefined. Вызов return без значения также возвращает undefined:

function f() { }

alert( f() ); // undefined

## Function Declaration и Expression

Функция в JavaScript является обычным значением.

Её можно создать в любом месте кода и присвоить в переменную, вот так:

var sum = function(a, b) {

var result = a + b;

return result;

}

alert( sum(1, 2) ); // 3

Такой синтаксис, при котором функция объявляется в контексте выражения (в данном случае, выражения присваивания), называется Function Expression, а обычный синтаксис, при котором функция объявляется в основном потоке кода – Function Declaration.

Функции, объявленные через Function Declaration, отличаются от Function Expression тем, что интерпретатор создаёт их при входе в область видимости (в начале выполнения скрипта), так что они работают до объявления.

Обычно это удобно, но может быть проблемой, если нужно объявить функцию в зависимости от условия. В этом случае, а также в других ситуациях, когда хочется создать функцию «здесь и сейчас», используют Function Expression.

## Named Function Expression

Если объявление функции является частью какого-либо выражения, например var f = function... или любого другого, то это Function Expression.

В этом случае функции можно присвоить «внутреннее» имя, указав его после function. Оно будет видно только внутри этой функции и позволяет обратиться к функции изнутри себя. Обычно это используется для рекурсивных вызовов.

Например, создадим функцию для вычисления факториала как Function Expression и дадим ей имя me:

var factorial = function me(n) {

return (n == 1) ? n : n \* me(n - 1);

}

alert( factorial(5) ); // 120

alert( me ); // ошибка, нет такой переменной

Ограничение видимости для имени не работает в IE8-, но вызов с его помощью работает во всех браузерах.

**Итого**

В этой главе мы повторили основные особенности JavaScript, знание которых необходимо для обхода большинства «граблей», да и просто для написания хорошего кода.

Это, конечно, лишь основы. Дальше вы узнаете много других особенностей и приёмов программирования на этом языке.

# Качество кода

## Отладка в браузере Chrome

Перед тем, как двигаться дальше, поговорим об отладке скриптов.

Все современные браузеры поддерживают для этого «инструменты разработчика». Исправление ошибок с их помощью намного проще и быстрее.

На текущий момент самые многофункциональные инструменты – в браузере Chrome. Также очень хорош Firebug (для Firefox).

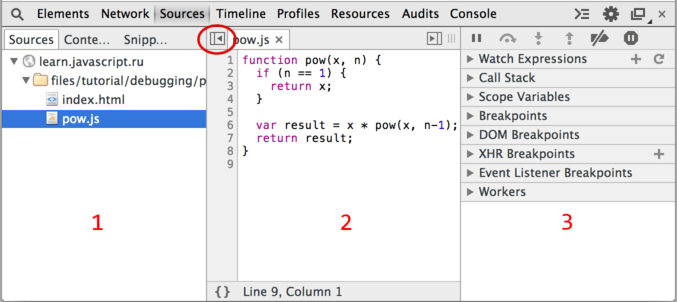
## Общий вид панели Sources

В вашей версии Chrome панель может выглядеть несколько по-иному, но что где находится, должно быть понятно.

Зайдите на [страницу с примером](http://learn.javascript.ru/article/debugging-chrome/debugging/index.html) браузером Chrome.

Откройте инструменты разработчика: F12 или в меню Инструменты > Инструменты Разработчика.

Выберите сверху Sources.



Вы видите три зоны:

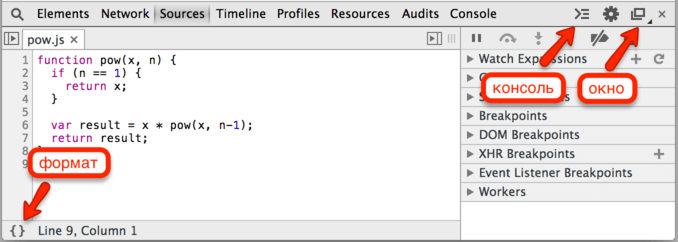
**Зона исходных файлов.** В ней находятся все подключённые к странице файлы, включая JS/CSS. Выберите pow.js, если он не выбран.

**Зона текста.** В ней находится текст файлов.

**Зона информации и контроля.** Мы поговорим о ней позже.

Обычно зона исходных файлов при отладке не нужна. Скройте её кнопкой .

## Общие кнопки управления



Три наиболее часто используемые кнопки управления:

**Формат**

Нажатие форматирует текст текущего файла, расставляет отступы. Нужна, если вы хотите разобраться в чужом коде, плохо отформатированном или сжатом.

**Консоль**

Очень полезная кнопка, открывает тут же консоль для запуска команд. Можно смотреть код и тут же запускать функции. Её нажатие можно заменить на клавишу Esc.

**Окно**

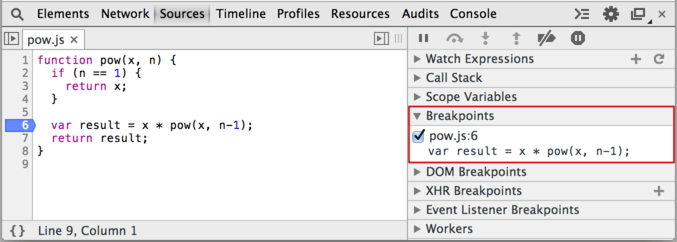
Если код очень большой, то можно вынести инструменты разработки вбок или в отдельное окно, зажав эту кнопку и выбрав соответствующий вариант из списка.

## Точки останова

Открыли файл pow.js во вкладке Sources? Кликните на 6-й строке файла pow.js, прямо на цифре 6.

Поздравляю! Вы поставили точку останова или, как чаще говорят, «брейкпойнт».

Выглядеть это должно примерно так:



Слово *Брейкпойнт* (breakpoint) – часто используемый английский жаргонизм. Это то место в коде, где отладчик будет *автоматически* останавливать выполнение JavaScript, как только оно до него дойдёт.

**В остановленном коде можно посмотреть текущие значения переменных, выполнять команды и т.п., в общем – отлаживать его.**

Вы можете видеть, что информация о точке останова появилась справа, в подвкладке Breakpoints.

Вкладка Breakpoints очень удобна, когда код большой, она позволяет:

Быстро перейти на место кода, где стоит брейкпойнт кликом на текст.

Временно выключить брейкпойнт кликом на чекбокс.

Быстро удалить брейкпойнт правым кликом на текст и выбором Remove, и так далее.

**Дополнительные возможности**

Остановку можно инициировать и напрямую из кода скрипта, командой debugger:

function pow(x, n) {

...

debugger; // <-- отладчик остановится тут

...

}

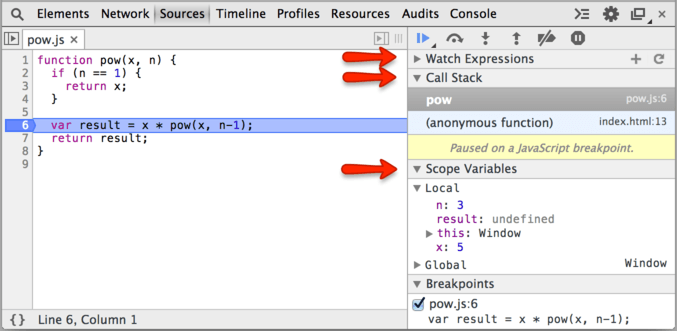
*Правый клик* на номер строки pow.js позволит создать условную точку останова (conditional breakpoint), т.е. задать условие, при котором точка останова сработает.

Это удобно, если останов нужен только при определённом значении переменной или параметра функции.

## Остановиться и осмотреться

Наша функция выполняется сразу при загрузке страницы, так что самый простой способ активировать отладчик JavaScript – перезагрузить её. Итак, нажимаем F5 (Windows, Linux) или Cmd+R (Mac).

Если вы сделали всё, как описано выше, то выполнение прервётся как раз на 6-й строке.



Обратите внимание на информационные вкладки справа (отмечены стрелками).

В них мы можем посмотреть текущее состояние:

**Watch Expressions – показывает текущие значения любых выражений.**

Можно раскрыть эту вкладку, нажать мышью + на ней и ввести любое выражение. Отладчик будет отображать его значение на текущий момент, автоматически перевычисляя его при проходе по коду.

**Call Stack – стек вызовов, все вложенные вызовы, которые привели к текущему месту кода.**

На текущий момент видно, отладчик находится в функции pow (pow.js, строка 6), вызванной из анонимного кода (index.html, строка 13).

**Scope Variables – переменные.**

На текущий момент строка 6 ещё не выполнилась, поэтому result равен undefined.

В Local показываются переменные функции: объявленные через var и параметры. Вы также можете там видеть ключевое слово this, если вы не знаете, что это такое – ничего страшного, мы это обсудим позже, в следующих главах учебника.

В Global – глобальные переменные и функции.

## Управление выполнением

Пришло время, как говорят, «погонять» скрипт и «оттрейсить» (от англ. trace – отслеживать) его работу.

Обратим внимание на панель управления справа-сверху, в ней есть 6 кнопок:

**http://learn.javascript.ru/article/debugging-chrome/manage1.png – продолжить выполнение, горячая клавиша F8.**

Продолжает выполнения скрипта с текущего момента в обычном режиме. Если скрипт не встретит новых точек останова, то в отладчик управление больше не вернётся.

Нажмите на эту кнопку.

Скрипт продолжится, далее, в 6-й строке находится рекурсивный вызов функции pow, т.е. управление перейдёт в неё опять (с другими аргументами) и сработает точка останова, вновь включая отладчик.

При этом вы увидите, что выполнение стоит на той же строке, но в Call Stack появился новый вызов.

Походите по стеку вверх-вниз – вы увидите, что действительно аргументы разные.

**http://learn.javascript.ru/article/debugging-chrome/manage2.png – сделать шаг, не заходя внутрь функции, горячая клавиша F10.**

Выполняет одну команду скрипта. Если в ней есть вызов функции – то отладчик обходит его стороной, т.е. не переходит на код внутри.

Эта кнопка очень удобна, если в текущей строке вызывается функция JS-фреймворка или какая-то другая, которая нас ну совсем не интересует. Тогда выполнение продолжится дальше, без захода в эту функцию, что нам и нужно.

Обратим внимание, в данном случае эта кнопка при нажатии всё-таки перейдёт внутрь вложенного вызова pow, так как внутри pow находится брейкпойнт, а на включённых брейкпойнтах отладчик останавливается всегда.

**http://learn.javascript.ru/article/debugging-chrome/manage3.png – сделать шаг, горячая клавиша F11.**

Выполняет одну команду скрипта и переходит к следующей. Если есть вложенный вызов, то заходит внутрь функции.

Эта кнопка позволяет подробнейшим образом пройтись по очереди по командам скрипта.

**http://learn.javascript.ru/article/debugging-chrome/manage4.png – выполнять до выхода из текущей функции, горячая клавиша Shift+F11.**

Выполняет команды до завершения текущей функции.

Эта кнопка очень удобна в случае, если мы нечаянно вошли во вложенный вызов, который нам не интересен – чтобы быстро из него выйти.

**http://learn.javascript.ru/article/debugging-chrome/manage5.png – отключить/включить все точки останова.**

Эта кнопка никак не двигает нас по коду, она позволяет временно отключить все точки останова в файле.

**http://learn.javascript.ru/article/debugging-chrome/manage6.png – включить/отключить автоматическую остановку при ошибке.**

Эта кнопка – одна из самых важных.

Нажмите её несколько раз. В старых версиях Chrome у неё три режима – нужен фиолетовый, в новых – два, тогда достаточно синего.

Когда она включена, то при ошибке в коде он автоматически остановится и мы сможем посмотреть в отладчике текущие значения переменных, при желании выполнить команды и выяснить, как так получилось.

**Процесс отладки заключается в том, что мы останавливаем скрипт, смотрим, что с переменными, переходим дальше и ищем, где поведение отклоняется от правильного.**

**Continue to here**

Правый клик на номер строки открывает контекстное меню, в котором можно запустить выполнение кода до неё (Continue to here). Это удобно, когда хочется сразу прыгнуть вперёд и breakpoint неохота ставить.

## Консоль

При отладке, кроме просмотра переменных и передвижения по скрипту, бывает полезно запускать команды JavaScript. Для этого нужна консоль.

В неё можно перейти, нажав кнопку «Console» вверху-справа, а можно и открыть в дополнение к отладчику, нажав на кнопку  или клавишей ESC.

**Самая любимая команда разработчиков: console.log(...).**

Она пишет переданные ей аргументы в консоль, например:

// результат будет виден в консоли

for (var i = 0; i < 5; i++) {

console.log("значение", i);

}

Полную информацию по специальным командам консоли вы можете получить на странице [Chrome Console API](https://developer.chrome.com/devtools/docs/console-api) и [Chrome CommandLine API](https://developer.chrome.com/devtools/docs/commandline-api). Почти все команды также действуют в Firebug (отладчик для браузера Firefox).

Консоль поддерживают все браузеры, и, хотя IE10- поддерживает далеко не все функции, но console.logработает везде. Используйте его для вывода отладочной информации по ходу работы скрипта.

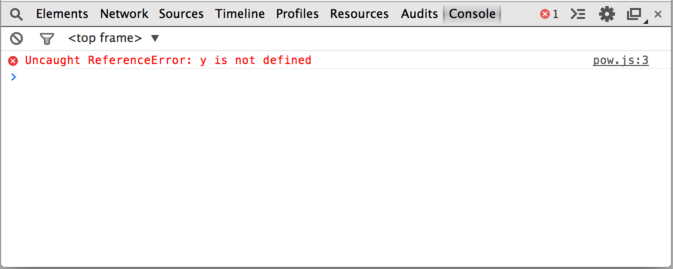
## Ошибки

Ошибки JavaScript выводятся в консоли.

Например, прервите отладку – для этого достаточно закрыть инструменты разработчика – и откройте [страницу с ошибкой](http://learn.javascript.ru/article/debugging-chrome/error/index.html).

Перейдите во вкладку Console инструментов разработчика (Ctrl+Shift+J / Cmd+Shift+J).

В консоли вы увидите что-то подобное:



Красная строка – это сообщение об ошибке.

Если кликнуть на ссылке pow.js в консоли, справа в строке с ошибкой, то мы перейдём непосредственно к месту в скрипте, где возникла ошибка.

Однако почему она возникла?

Более подробно прояснить произошедшее нам поможет отладчик. Он может «заморозить» выполнение скрипта на момент ошибки и дать нам возможность посмотреть значения переменных и стека на тот момент.

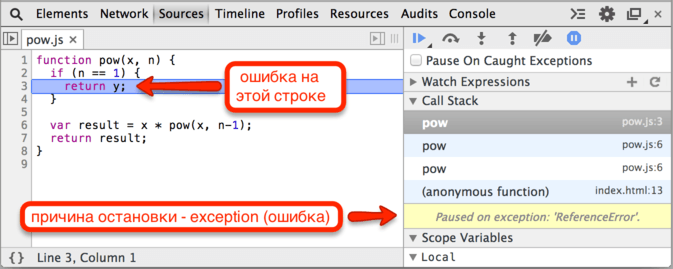
Для этого:

Перейдите на вкладку Sources.

Включите останов при ошибке, кликнув на кнопку http://learn.javascript.ru/article/debugging-chrome/manage6.png

Перезагрузите страницу.

После перезагрузки страницы JavaScript-код запустится снова и отладчик остановит выполнение на строке с ошибкой:



Можно посмотреть значения переменных. Открыть консоль и попробовать запустить что-то в ней. Поставить брейкпойнты раньше по коду и посмотреть, что привело к такой печальной картине, и так далее.

## Итого

Отладчик позволяет:

Останавливаться на отмеченном месте (breakpoint) или по команде debugger.

Выполнять код – по одной строке или до определённого места.

Смотреть переменные, выполнять команды в консоли и т.п.

В этой главе кратко описаны возможности отладчика Google Chrome, относящиеся именно к работе с кодом.

Пока что это всё, что нам надо, но, конечно, инструменты разработчика умеют много чего ещё. В частности, вкладка Elements – позволяет работать со страницей (понадобится позже), Timeline – смотреть, что именно делает браузер и сколько это у него занимает и т.п.

# Советы по стилю кода

Осваивать можно двумя путями:

[Официальная документация](https://developer.chrome.com/devtools) (на англ.)

Кликать в разных местах и смотреть, что получается. Не забывать о клике правой кнопкой мыши.

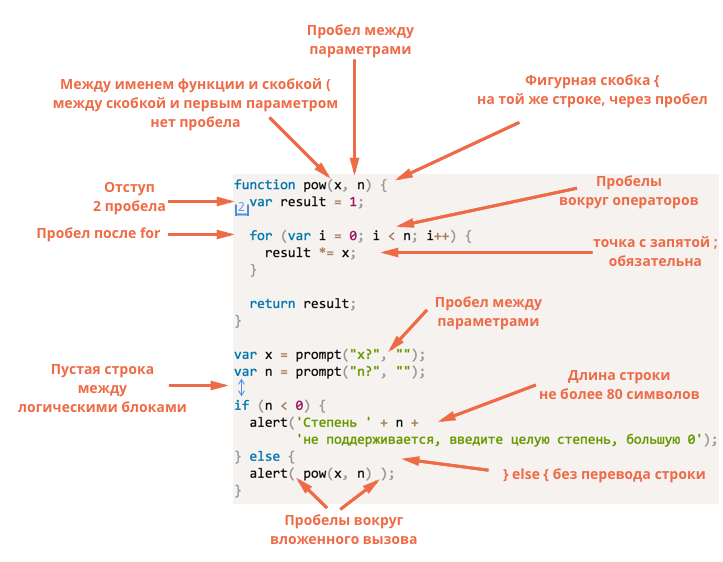
Мы ещё вернёмся к отладчику позже, когда будем работать с HTML.

## Советы по стилю кода

Код должен быть максимально читаемым и понятным. Для этого нужен *хороший стиль* написания кода. В этой главе мы рассмотрим компоненты такого стиля.

## Синтаксис

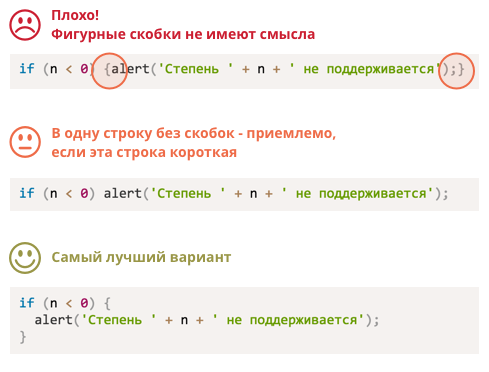
Шпаргалка с правилами синтаксиса (детально их варианты разобраны далее):



Не всё здесь однозначно, так что разберём эти правила подробнее.

## Фигурные скобки

Пишутся на той же строке, так называемый «египетский» стиль. Перед скобкой – пробел.



Если у вас уже есть опыт в разработке, и вы привыкли делать скобку на отдельной строке – это тоже вариант. В конце концов, решать вам. Но в большинстве JavaScript-фреймворков стиль именно такой.

Если условие и код достаточно короткие, например if (cond) return null, то запись в одну строку вполне читаема… Но, как правило, отдельная строка всё равно воспринимается лучше.

## Длина строки

Максимальную длину строки согласовывают в команде. Как правило, это либо 80, либо 120 символов, в зависимости от того, какие мониторы у разработчиков.

Более длинные строки необходимо разбивать для улучшения читаемости.

## Отступы

Отступы нужны двух типов:

* **Горизонтальный отступ, при вложенности – два(или четыре) пробела.**

Как правило, используются именно пробелы, т.к. они позволяют сделать более гибкие «конфигурации отступов», чем символ «Tab».

Например, выровнять аргументы относительно открывающей скобки:

show("Строки" +

" выровнены" +

" строго" +

" одна под другой");

* **Вертикальный отступ, для лучшей разбивки кода – перевод строки.**

Используется, чтобы разделить логические блоки внутри одной функции. В примере разделены инициализация переменных, главный цикл и возвращение результата:

function pow(x, n) {

var result = 1;

// <--

for (var i = 0; i < n; i++) {

result \*= x;

}

// <--

return result;

}

Вставляйте дополнительный перевод строки туда, где это сделает код более читаемым. Не должно быть более 9 строк кода подряд без вертикального отступа.

## Точка с запятой

Точки с запятой нужно ставить, даже если их, казалось бы, можно пропустить.

Есть языки, в которых точка с запятой не обязательна, и её там никто не ставит. В JavaScript перевод строки её заменяет, но лишь частично, поэтому лучше её ставить.

## Именование

Общее правило:

* Имя переменной – существительное.
* Имя функции – глагол или начинается с глагола. Бывает, что имена для краткости делают существительными, но глаголы понятнее.

Для имён используется английский язык (не транслит) и верблюжья нотация.

## Уровни вложенности

Уровней вложенности должно быть немного.

Например, [проверки в циклах можно делать через «continue»](http://learn.javascript.ru/while-for#continue), чтобы не было дополнительного уровня if(..) { ... }:

Вместо:

for (var i = 0; i < 10; i++) {

if (i подходит) {

... // <- уровень вложенности 2

}

}

Используйте:

for (var i = 0; i < 10; i++) {

if (i не подходит) continue;

... // <- уровень вложенности 1

}

Аналогичная ситуация – с if/else и return. Следующие две конструкции идентичны.

Первая:

function isEven(n) { // проверка чётности

if (n % 2 == 0) {

return true;

} else {

return false;

}

}

Вторая:

function isEven(n) { // проверка чётности

if (n % 2 == 0) {

return true;

}

return false;

}

Если в блоке if идёт return, то else за ним не нужен.

**Лучше быстро обработать простые случаи, вернуть результат, а дальше разбираться со сложным, без дополнительного уровня вложенности.**

В случае с функцией isEven можно было бы поступить и проще:

function isEven(n) { // проверка чётности

return !(n % 2);

}

…Однако, если код !(n % 2) для вас менее очевиден чем предыдущий вариант, то стоит использовать предыдущий.

Главное для нас – не краткость кода, а его простота и читаемость. Совсем не всегда более короткий код проще для понимания, чем более развёрнутый.

## Функции = Комментарии

Функции должны быть небольшими. Если функция большая – желательно разбить её на несколько.

Этому правилу бывает сложно следовать, но оно стоит того. При чем же здесь комментарии?

Вызов отдельной небольшой функции не только легче отлаживать и тестировать – сам факт его наличия является *отличным комментарием*.

Сравните, например, две функции showPrimes(n) для вывода простых чисел до n.

Первый вариант использует метку:

function showPrimes(n) {

nextPrime: for (var i = 2; i < n; i++) {

for (var j = 2; j < i; j++) {

if (i % j == 0) continue nextPrime;

}

alert( i ); // простое

}

}

Второй вариант – дополнительную функцию isPrime(n) для проверки на простоту:

function showPrimes(n) {

for (var i = 2; i < n; i++) {

if (!isPrime(i)) continue;

alert(i); // простое

}

}

function isPrime(n) {

for (var i = 2; i < n; i++) {

if ( n % i == 0) return false;

}

return true;

}

Второй вариант проще и понятнее, не правда ли? Вместо участка кода мы видим описание действия, которое там совершается (проверка isPrime).

## Функции – под кодом

Есть два способа расположить функции, необходимые для выполнения кода.

// объявить функции

function createElement() {

...

}

function setHandler(elem) {

...

}

function walkAround() {

...

}

// код, использующий функции

var elem = createElement();

setHandler(elem);

walkAround();

Сначала код, а функции внизу:

// код, использующий функции

var elem = createElement();

setHandler(elem);

walkAround();

// --- функции ---

function createElement() {

...

}

function setHandler(elem) {

...

}

function walkAround() {

...

}

…На самом деле существует еще третий «стиль», при котором функции хаотично разбросаны по коду, но это ведь не наш метод, да?

**Как правило, лучше располагать функции под кодом, который их использует.**

То есть, предпочтителен 2-й способ.

Дело в том, что при чтении такого кода мы хотим знать в первую очередь, *что он делает*, а уже затем *какие функции ему помогают.* Если первым идёт код, то это как раз дает необходимую информацию. Что же касается функций, то вполне возможно нам и не понадобится их читать, особенно если они названы адекватно и то, что они делают, понятно из названия.

## Плохие комментарии

В коде нужны комментарии.

Сразу начну с того, каких комментариев быть почти не должно.

**Должен быть минимум комментариев, которые отвечают на вопрос «что происходит в коде?»**

Что интересно, в коде начинающих разработчиков обычно комментариев либо нет, либо они как раз такого типа: «что делается в этих строках».

Серьёзно, хороший код и так понятен.

Об этом замечательно выразился Р.Мартин в книге [«Чистый код»](http://www.ozon.ru/context/detail/id/21916535/): «Если вам кажется, что нужно добавить комментарий для улучшения понимания, это значит, что ваш код недостаточно прост, и, может, стоит переписать его».

Если у вас образовалась длинная «простыня», то, возможно, стоит разбить её на отдельные функции, и тогда из их названий будет понятно, что делает тот или иной фрагмент.

Да, конечно, бывают сложные алгоритмы, хитрые решения для оптимизации, поэтому нельзя такие комментарии просто запретить. Но перед тем, как писать подобное – подумайте: «Нельзя ли сделать код понятным и без них?»

## Хорошие комментарии

А какие комментарии полезны и приветствуются?

* **Архитектурный комментарий – «как оно, вообще, устроено».**

Какие компоненты есть, какие технологии использованы, поток взаимодействия. О чём и зачем этот скрипт. Взгляд с высоты птичьего полёта. Эти комментарии особенно нужны, если вы не один, а проект большой.

Для описания архитектуры, кстати, создан специальный язык [UML](http://ru.wikipedia.org/wiki/Unified_Modeling_Language), красивые диаграммы, но можно и без этого. Главное – чтобы понятно.

* **Справочный комментарий перед функцией – о том, что именно она делает, какие параметры принимает и что возвращает.**

Для таких комментариев существует синтаксис [JSDoc](http://en.wikipedia.org/wiki/JSDoc).

/\*\*

\* Возвращает x в степени n, только для натуральных n

\*

\* @param {number} x Число для возведения в степень.

\* @param {number} n Показатель степени, натуральное число.

\* @return {number} x в степени n.

\*/

function pow(x, n) {

...

}

Такие комментарии позволяют сразу понять, что принимает и что делает функция, не вникая в код.

Кстати, они автоматически обрабатываются многими редакторами, например [Aptana](http://aptana.com/) и редакторами от [JetBrains](http://www.jetbrains.com/), которые учитывают их при автодополнении, а также выводят их в автоподсказках при наборе кода.

Кроме того, есть инструменты, например [JSDoc 3](https://github.com/jsdoc3/jsdoc), которые умеют генерировать по таким комментариям документацию в формате HTML. Более подробную информацию об этом можно также найти на сайте <http://usejsdoc.org/>.

**…Но куда более важными могут быть комментарии, которые объясняют не *что*, а *почему* в коде происходит именно это!**

Как правило, из кода можно понять, что он делает. Бывает, конечно, всякое, но, в конце концов, вы этот код *видите*. Однако гораздо важнее может быть то, чего вы *не видите*!

*Почему* это сделано именно так? На это сам код ответа не даёт.

Например:

**Есть несколько способов решения задачи. Почему выбран именно этот?**

Например, пробовали решить задачу по-другому, но не получилось – напишите об этом. Почему вы выбрали именно этот способ решения? Особенно это важно в тех случаях, когда используется не первый приходящий в голову способ, а какой-то другой.

Без этого возможна, например, такая ситуация:

* Вы открываете код, который был написан какое-то время назад, и видите, что он «неоптимален».
* Думаете: «Какой я был дурак», и переписываете под «более очевидный и правильный» вариант.
* …Порыв, конечно, хороший, да только этот вариант вы уже обдумали раньше. И отказались, а почему – забыли. В процессе переписывания вспомнили, конечно (к счастью), но результат – потеря времени на повторное обдумывание.

Комментарии, которые объясняют выбор решения, очень важны. Они помогают понять происходящее и предпринять правильные шаги при развитии кода.

**Какие неочевидные возможности обеспечивает этот код? Где ещё они используются?**

В хорошем коде должно быть минимум неочевидного. Но там, где это есть – пожалуйста, комментируйте.

**Комментарии – это важно**

Один из показателей хорошего разработчика – качество комментариев, которые позволяют эффективно поддерживать код, возвращаться к нему после любой паузы и легко вносить изменения.

## Руководства по стилю

Когда написанием проекта занимается целая команда, то должен существовать один стандарт кода, описывающий где и когда ставить пробелы, запятые, переносы строк и т.п.

Сейчас, когда есть столько готовых проектов, нет смысла придумывать целиком своё руководство по стилю. Можно взять уже готовое, к которому, по желанию, всегда можно что-то добавить.

Большинство есть на английском, сообщите мне, если найдёте хороший перевод:

* [Google JavaScript Style Guide](https://google.github.io/styleguide/javascriptguide.xml)
* [jQuery JavaScript Style Guide](http://contribute.jquery.org/style-guide/js/)
* [Airbnb JavaScript Style Guide](https://github.com/airbnb/javascript)
* [Idiomatic.JS](https://github.com/rwaldron/idiomatic.js) (есть [перевод](https://github.com/rwaldron/idiomatic.js/tree/master/translations/ru_RU))
* [Dojo Style Guide](https://dojotoolkit.org/reference-guide/1.10/developer/styleguide.html)

Для того, чтобы начать разработку, вполне хватит элементов стилей, обозначенных в этой главе. В дальнейшем, посмотрев эти руководства, вы можете выработать и свой стиль, но лучше не делать его особенно «уникальным и неповторимым», себе дороже потом будет с людьми сотрудничать.

## Автоматизированные средства проверки

Существуют средства, проверяющие стиль кода.

Самые известные – это:

* [JSLint](http://www.jslint.com/) – проверяет код на соответствие [стилю JSLint](http://www.jslint.com/lint.html), в онлайн-интерфейсе вверху можно ввести код, а внизу различные настройки проверки, чтобы сделать её более мягкой.
* [JSHint](http://www.jshint.com/) – вариант JSLint с большим количеством настроек.
* [Closure Linter](https://developers.google.com/closure/utilities/) – проверка на соответствие [Google JavaScript Style Guide](http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml).

В частности, JSLint и JSHint интегрированы с большинством редакторов, они гибко настраиваются под нужный стиль и совершенно незаметно улучшают разработку, подсказывая, где и что поправить.

Побочный эффект – они видят некоторые ошибки, например необъявленные переменные. У меня это обычно результат опечатки, которые таким образом сразу отлавливаются. Очень рекомендую поставить что-то из этого. Я использую [JSHint](http://www.jshint.com/).

**Итого**

Описанные принципы оформления кода уместны в большинстве проектов. Есть и другие полезные соглашения.

Следуя (или не следуя) им, необходимо помнить, что любые советы по стилю хороши лишь тогда, когда делают код читаемее, понятнее, проще в поддержке.

function pow(x, n) {

var result = 1;

for (var i = 0; i < n; i++) {

result \*= x;

}

return result;

}

var x = prompt("x?", "");

var n = prompt("n?", "");

if (n < 0) {

alert('Степень ' + n +

'не поддерживается, введите целую степень, большую 0');

} else {

alert( pow(x, n) );

}

# Автоматические тесты при помощи chai и mocha

<http://learn.javascript.ru/testing>

# Введение в 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». Здесь мы рассмотрим основы использования свойств и методов.

## Свойство str.length

У строки есть *свойство* length, содержащее длину:

alert( "Привет, мир!".length ); // 12

Можно и записать строку в переменную, а потом запросить её свойство:

var str = "Привет, мир!";

alert( str.length ); // 12

## Метод str.toUpperCase()

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

var hello = "Привет, мир!";

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

**Вызов метода – через круглые скобки!**

Обратите внимание, для вызова метода обязательно нужны круглые скобки.

Посмотрите, например, результат обращения к toUpperCase без скобок:

var hello = "Привет";

alert( hello.toUpperCase ); // function...

Метод – это встроенная команда («функция», мы поговорим о них позже), которую нужно вызвать для получения значения. При обращении без скобок мы получим саму эту функцию. Как правило браузер выведет её как-то так: "function toUpperCase() { ... }".

А чтобы получить результат – нужно произвести её вызов, добавив скобки:

var hello = "Привет";

alert( hello.toUpperCase() ); // ПРИВЕТ

## Метод num.toFixed(n)

Есть методы и у чисел, например 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"

## 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

**Итого**

В этой главе мы познакомились с методами и свойствами.

Почти все значения в JavaScript, кроме разве что null и undefined имеют их и предоставляют через них разный функционал.

Далее мы подробно разберём основные свойства и методы структур данных в JavaScript.

## Задачи

**Сумма четных цифр от 0 до a;**

function testWhile(a) {

var x = 0;

for( i=2; i <= a; i +=2) {

x += i;

}

return x;

}

**Дано натуральное число n. Напишите *рекурсивную* функцию, которая возвращает строку чисел от 1 до n, разделенных пробелом.**

function getNumLine(a) {

if (a == 1) return '1';

else return getNumLine(a - 1) + " " + a;

}

**В этом задании в нашу функцию testStr передаются две строки. Вам нужно вернуть из функции их суммарную длину.**

function lengthStr(a,b) {

if (typeof a == 'string' && typeof b == 'string') {

var result = a.length + b.length;

} else {

a = String(a);

b = String(b);

var result = a.length + b.length;

}

return result;

}

lengthStr(234234,234234234);

**В этом задании в нашу функцию testArray передаются два массива случайной длины заполненные случайными числами. Вам нужно сосчитать сумму всех элементов обоих массивов и возвратить ее из функции.**

function testArray(a,b) {

var sumA = 0,

sumB = 0,

i = a.length,

j = b.length;

while (i--) {

sumA += a[i];

}

while (j--) {

sumB += b[j];

}

return sumA + sumB;

}

console.log ( testArray([8, 1, 1, 7, 4, 0], [5, 8, 5, 4, 8]) )

В этом задании в нашу функцию testArray передаются две строки случайной длины и содержания. Вам нужно составить из символов этих строк один массив (каждый символ строки становится отдельным элементом массива), затем добавить первым элементом  массива текстовое значение "Иванов", и вернуть из функции все элементы в обратном порядке, преобразовав в строку. Обратите внимание, что в обратном порядке нужно переставить элементы внутри массива, а данные внутри элементов инвертировать не нужно!

function testArray(a,b) {

var str = a+b;

var mass = str.split('');

mass.splice(0, 0 , 'Иванов');

mass.reverse();

var reverseStr = mass.join('');

return reverseStr;

}

# Продвинутые операции

А начнем мы изучение этой темы с понятия "**Исключение**". Рассмотрим несколько терминов:

## Исключение

- это некоторое событие, сигнализирующее о возникновении нештатной ситуации или ошибки.

**Возбудить, (создать, бросить) исключение** - просигнализировать о такой ошибке или исключительной ситуации.

**Перехватить исключение** - значит предпринять действия по обработке исключения и восстановления нормальной работоспособности кода.

Возбуждение исключения производится оператором "**throw**", перехват - командой "**catch**" (точнее связкой операторов "try-catch-finally")

## throw

Синтаксис оператора throw выглядит следующим образом:

throw выражение;

"*выражение*" (или результат его вычисления) может представлять из себя практически любой тип данных: строку, число, булево значение, объект. Например число, представляющее код ошибки, строку, содержащую текст ошибки.

Например, приведенный ниже пример демонстрирует проверку входного параметра в рекурсивной функции при вычислении:

function testFactorial(inputData) {

  if (inputData < 0)   // Проверяем - положительное ли число

throw "Число не должно быть меньше нуля";   // Если отрицательное - "бросаем" исключение

return (inputData - 1)?(inputData \* testFactorial(inputData - 1)):inputData;

}

В этом примере, если на вход функции будет подано число меньше 0, то будет сгенерирована ошибка с текстом "Число не должно быть меньше нуля" и произойдет выход из функции.

Если ввод корректный, то будет вычислен факториал числа, поданного на вход.

Как только исключение создано, интерпретатор JavaScript прерывает нормальное выполнение кода и начинает поиск обработчика исключений - конструкции **try/catch/finally**.

Синтаксис этой конструкции выглядит следующим образом:

try {  
  // код, который нужно "попробовать"  
  // в этом коде может быть брошено исключение

} catch(exception\_variable) {  
  // в этом месте пишется код, который выполняется только в случае обнаружения

// исключения в предыдущем блоке "trу"   
  // в случае возникновения исключения, в переменную exception\_variable будет передан

// код возникшей ошибки, например аргумент оператора throw

} finally {  
  // Код в этом блоке будет выполнен всегда, независимо от результата завершения блока try:  
  // и при завершении без ошибки, и при завершении с ошибкой, и при завершении по любому оператору перехода  
  // (break, continue, return)

}

Блоки **catch** и **finally**не являются строго обязательными, однако хотя бы один из них должен присутствовать в конструкции.

Давайте рассмотрим пример с использованием функции factorial(), созданной в течение этого урока.

// Объявление функции

function testFactorial(inputData) {

  if (inputData < 0)   // Проверяем - положительное ли число

throw "Число не должно быть меньше нуля";   // Если отрицательное - "бросаем" исключение

return (inputData - 1) ? (inputData \* testFactorial(inputData - 1)) : inputData;

}

// инициируем переменную для входного параметра, зададим ее вручную в этом примере

var myNumber = -5;

// начало конструкции обработки ошибок

try {  
 document.write(testFactorial(myNumber));    //  попытка вызова функции

} catch(ex) {  
 document.write(ex); // если в функции произойдет исключение, то будет выведен текст,

    // который мы использовали в параметре оператора throw

}

В этом примере мы не использовали блок finally, он нам тут не нужен. Если вы помните, минимум один любой из блоков finally и catch может быть использован в конструкции, оба - не обязательно.

Очень важный момент - конструкции **try/catch/finally** могут быть многократно вложенными. В такой ситуации нам зачастую необходимо проверить - а что за исключение возникло, и если это не то исключение, которое мы ждали и готовы обработать, "не наше", то нужно его отправить дальше. Вариант "не перехватывать" мы реализовать не можем, однако можно перехватить, проверить и если это не то, что нужно - сгенерировать заново, как бы послать дальше.

Давайте посмотрим, как бы мы реализовали предыдущий пример с таким уточнением:

// Объявление функции

function testFactorial(inputData) {  
  if (inputData < 0)   // Проверяем - положительное ли число

throw "Число не должно быть меньше нуля";   // Если отрицательное - "бросаем" исключение

return (inputData - 1) ? (inputData \* testFactorial(inputData - 1)) : inputData;  
}

// инициируем переменную для входного параметра, зададим ее вручную в этом примере

var myNumber = -5;

// начало конструкции обработки ошибок

try {  
 document.write(testFactorial(myNumber));     //  попытка вызова функции

} catch(ex) {  
 if (ex != "Число не должно быть меньше нуля") // Проверяем - если исключение не наше

throw (ex); // то "запускаем" его дальше

document.write(ex); // вывод строки ошибки если исключение "наше"

}

В этом примере мы перед тем как вывести ошибку на экран, проверяем - та ли это ошибка, которая была сгенерирована оператором throw в нашей функции. Если нет, то генерируем ее заново - как бы отправляем гулять дальше, в поисках своего catch.

## Error

Также в процессе рассмотрения процесса обработки ошибок нам необходимо рассмотреть стандартный объект **Error**, который используется при возбуждении исключений.

В принципе командой throw может быть выброшен практически любой объект, однако стандартно при возникновении ошибок выбрасывается объект класса **Error** или его подклассы.

Синтаксис, который мы рассмотрим,  выглядит следующим образом:

new Error(*message*);

Этой командой создается новый экземпляр объекта Error, а текст "message" записывается в свойство объекта с message.

Самый простой пример вызова конструктора Error выглядит следующим образом.

try {  
  throw new Error('*Что-то пошло не так*!');  
} catch (e) {  
  console.log(e.name + ': ' + e.message);  
}

В данном примере мы создали объект типа **Error**с именем Error и текстом сообщения об ошибке "*Что-то пошло не так* !".

Например, для использования объекта в функции из предыдущего примера, нам нужно было бы написать вот так:

// Объявление функции

function testFactorial(inputData) {  
 if (inputData < 0)                                    // Проверяем - положительное ли число

throw new Error("Число не должно быть меньше нуля");   // Создаем и бросаем экземпляр объекта Error

return (inputData - 1) ? (inputData \* testFactorial(inputData - 1)) : inputData;

}

var myNumber = -5;

try {  
 document.write(testFactorial(myNumber));    //  попытка вызова функции

} catch(ex) {  
 document.write(ex.message); // вывод сообщения об ошибке

}

В данном случае мы перехватили исключение и вывели в документ его свойство message, в которое ранее поместили текст - "Число не должно быть меньше нуля".

## регулярные выражения

Регулярные выражения (**RegExp** - **regular expressions**) это объект, который может описывать шаблон символов.

Например, если вы ищете в строке подстроку, вы можете описать шаблон того, что вы ищете.

Простой шаблон может состоять даже из одного символа, большие и сложные шаблоны могут использоваться для парсинга, проверки формата введенных данных, замены и других различных целей.

Содержание шаблона регулярного выражения состоит из последовательности символов. Большая их часть (все буквы и цифры) описывают сами себя - непосредственно указывают свое присутствие. Например, регулярное выражение

/hello/

совпадет со всеми строками, в которых есть слово "hello". Другие символы обозначают не себя, а имеют некоторое модифицирующее значение.

Например, выражение

/hello$/

будет соответствовать строкам, которые ЗАКАНЧИВАЮТСЯ на слово "hello". Это обеспечивает метасимвол "$", обозначающий конец строки.

### Создание объектов RegExp

Объекты RegExp могут быть созданы **с помощью конструктора RegExp()**, или **с помощью литералов**. Но если, например, строковые литералы задаются в виде символов, заключенных в кавычки, то литералы регулярных выражений задаются заключенными в пару символов слэш  "**/**". Например, вот так:

var myPattern = /q$/; //Создание регулярного выражения с помощью литерала

В данном примере мы создали**c помощью литерала** новый объект типа RegExp и присвоили его переменной myPattern. Данный шаблон соответствует любой строке, заканчивающейся символом q.

Для создания такого же объекта **с помощью конструктора** нам нужно написать вот такое выражение:

var myPattern = new RegExp("q$"); //Создание регулярного выражения с помощью конструктора

### Составление регулярных выражений.

Первая группа это конечно же **символы**. Как мы уже говорили все алфавитные и цифровые символы обозначают сами себя. Также можно вводить некоторые не алфавитные символы с помощью последовательностей, начинающихся с обратного слэша.

**Цифры и буквы** - соответствуют сами себе

\0 - Символ NUL (Соответствует \u0000 в Unicode)  
\t - Табуляция (\u0009)  
\n - Перевод строки (\u000A)  
\v - Вертикальная табуляция (\u000B)  
\f - Перевод страницы (\u000C)  
\r - Возврат каретки (\u000D)  
\xnn - Символ из набора Latin, задаваемый шестнадцатиричным номером nn  
\unnnn - Символ Unicode,  задаваемый шестнадцатиричным номером nnnn  
\cX - Управляющий символ "Х", например \сJ эквивалентна \n

Также в регулярных выражениях используются следующие символы:

^ $ . \* + ? = ! : | \ / ( ) [ ] { }

В следующих шагах мы более подробно рассмотрим их значение и применение в комбинации с другими символами, однако сейчас нужно запомнить, что для определения смысла этих символов буквально, т.е. "самих себя", необходимо перед ними ставить символ обратного слэша.

Например, если вы хотите написать регулярное выражение, по которому будет находиться символ обратного слэша, то вы должны в выражение поставить этот символ, предваряемый таким же символом обратного слэша. В результате такое регулярное выражение будет выглядеть следующим образом:

/\\/

### Классы

Отдельные символы могут быть объединены в **классы**. Это обозначается набором символов, заключенных в квадратные скобки. Например регулярное выражение

/[0123456789]/ //соответствует любой цифре.

Или же можно указав перед набором символов знак "**^**" определить регулярное выражение, которое будет соответствовать любому символу, КРОМЕ тех, которые указаны в скобках - класс с отрицанием. Например выражение

/[^0123456789]/ //будет соответствовать любому символу КРОМЕ цифр.

Также в классах можно задавать диапазон с помощью знака дефиса "**-**", чтобы не перечислять все символы. Например все цифры можно обозначить таким выражением:  
/[0-9]/

Некоторые классы из наборов символов настолько часто используются, что для них определили специальные обозначения:

**[...]** - любой из символов, указанных в скобках  
**[^...]** - любой кроме символов, указанных в скобках   
**.** (точка) - любой символ кроме перевода строки или другого разделителя строки  
**\w** - эквивалентно [a-zA-Z0-9\_] (Любой текстовый символ ASCII)  
**\W** - эквивалентно [^a-zA-Z0-9\_] (Любой символ кроме текстовых символов ASCII)   
**\s** - любой пробельный символ из Unicode

**\S** - любой НЕпробельный символ из Unicode   
**\d** - эквивалентно [0-9] (любые цифры ASCII)  
**\D** - эквивалентно [^0-9] (все символы кроме цифр ASCII)   
**[\b]** - обозначение символа "забой"

Последовательности таких управляющих символов также можно объединить в класс, например регулярное выражение

/[\w\s]/

соответствует любому пробельному символу или символу ASCII - букве или цифре.

### Квантификация

Описанные выше шаблоны можно использовать не только для описания одиночных комбинаций символов, но и для сколь угодно многократных **повторений**. Это называют "**квантификацией**".

**Для квантификации** в регулярных выражениях есть набор специальных комбинаций, заключаемых в **фигурные скобки**. Эта комбинация в фигурных скобках должна следовать сразу за описанным шаблоном. Например комбинация   
/\d{4}/

соответствует числу, состоящему из 4-х цифр.

Давайте рассмотрим какие бывают управляющие комбинации для повторений:

**{n}** - обозначает ровно n экземпляров шаблона  
**{n,}** - обозначает n или больше экземпляров шаблона    
**{n,m}** -  обозначает не менее n и не более m экземпляров шаблона   
**?** -  обозначает ноль или один экземпляр шаблона (эквивалентно выражению {0,1} )  
**+** -  обозначает 1 или более экземпляров шаблона (эквивалентно выражению {1,} )   
**\*** -  обозначает ноль или более экземпляров шаблона (эквивалентно выражению {0,} )

Символы повторения, указанные в таблице соответствуют максимально возможному количеству соответствий. Например выражение

/х{1,}/

примеренное к строке "ххх" будет соответствовать максимальному количеству соответствий, т.е. всем трем буквам "х", встреченным в строке. Это называется "жадным" повторением. ("жадной" квантификацией).

Если же мы хотим ограничить поиск первым же вхождением, то может использовать так называемую "нежадную", или "ленивую" квантификацию. Для этого после управляющей комбинации повторений ставится символ "?". Таким образом выражение

/х{1,}?/ //будет соответствовать только первому соответствию, т.е. только первой букве "х" в строке.

**Важный момент! Признак "ленивости" действует только на тот квантификатор (подшаблон) в шаблоне, после которого стоит, все остальные квантификаторы остаются "жадными".**

### Альтернативы

Синтаксис регулярных выражений содержит специальный символ для определения **альтернативы**, т.е. можно указать больше одного варианта шаблона, соответствие которому будет проверяться.  Для разделения альтернатив используется символ "|" - вертикальная черта. Например, выражение

/ma|pa|da/ // будет соответствовать либо строке "ma" либо строке "pa" либо строке "da".

Альтернативы конечно также могут комбинироваться и с классами и с повторениями. Указанный ниже шаблон соответствует либо двум цифрам либо двум строчным буквам либо двум заглавным буквам:

/\d{2}|[a-z]{2}|[A-Z]{2}/

Необходимо обратить внимание, что альтернативы обрабатываются слева направо до первого соответствия. После нахождения первого соответствия остальные альтернативы будут игнорироваться. На практике это означает что, например, шаблон /1|12|123/ примененный к строке "123", будет соответствовать первому символу, хотя в альтернативах есть гораздо более полное соответствие.

Рассмотрим еще одну возможность, которую мы можем использовать в регулярных выражениях - **группировку**.

### Группировка

**Группировка**обозначается заключением подшаблона в **круглые скобки ( )**. При этом элементы, используемые совместно со специальными символами, например |, +, \*, ? и другие, будут рассматриваться как одно целое.

Например шаблон

/regular(expression)?/

будет соответствовать слову "regular" за которым следует необязательное слово "expression".

### Граница соответствия

Еще одна возможность регулярных выражений - указание границы соответствия. Для этого используются **якорные выражения**.

Часто нам нужно найти слово, находящееся на отдельной строке, или в начале строки. Как вариант, может понадобиться найти отдельное слово, однако просто задать шаблон, в котором слово будет обрамлено пробелами мы не можем - в выборку не попадут слова с которых начинаются строки, или которыми заканчиваются. Также граница слова может определяться любым знаком препинания в тексте и перечислять в шаблоне все возможные комбинации было бы достаточно утомительно. Давайте рассмотрим, какие специальные символы используются для определения границ:

**^** - соответствует началу строки при многострочном поиске или началу строкового выражения  
**$** - соответствует концу строки при многострочном поиске или концу строкового выражения   
**\b** - соответствует границе слова, т.е. позиции между текстовым (aA-zZ) и не-текстовым символом, либо между текстовым         символом и началом или концом строки.  
**\B** - Соответствует позиции, не являющейся границей слов.  
**(?=p)** - Позитивная опережающая проверка на последующие символы - убеждается в том, что последующие символы соответствуют шаблону "р" но не включает их в результат поиска.  
**(?!p)** - Негативная опережающая проверка на последующие символы - требует чтобы последующие символы НЕ соответствовали шаблону "р".

### Флаги

Ну и последний элемент синтаксиса регулярных выражений - **флаги**. Флаги задают глобальные правила для всего шаблона и указываются не внутри символов слэша, в которые заключен шаблон, а ПОСЛЕ них.

JavaScript поддерживает три варианта флагов:  
i - указывает на то, что поиск по шаблону должен быть не чувствительным к регистру  
g - указывает что поиск должен быть глобальным, т.е. должны быть найдены ВСЕ соответствия в строке  
m - указывает на то, что поиск должен производиться в многострочном режиме.

### Методы класса String

В данном уроке сначала мы рассмотрим методы класса String, позволяющие использовать регулярные выражения:

 search(regexp)  
 replace(regexp, newString)  
 match(regexp)  
 split(divider)

### search()

В качестве аргумента мы передаем ему регулярное выражение, а он нам в ответ возвращает номер позиции, с которой найдено соответствие шаблону, либо "-1" если соответствие не найдено.

var myString = "This is just test string";  
result = myString.search(/is/);

В указанном примере в переменной result окажется число 2 (отсчет позиций начинается с 0).

Два важных момента:  
1. Метод search() не поддерживает глобальный поиск и флаг g в составе регулярного выражения будет игнорирован.

2.  Если аргумент не является регулярным выражением, то он будет преобразован в него передачей конструктору RegExp.

### replace()

С его помощью можно выполнить операцию **поиска с заменой**.  
В качестве аргументов он принимает *регулярное выражение* и *строку замены*.

var myString = "This is just test string";  
result = myString.replace(/is/,"as");

Данный пример заменит первое найденное соответствие шаблону ("is") на подстроку "as", в результате в переменной result окажется строка  "*Th*as*is just test string*".

Нужно отметить, что этот метод поддерживает глобальный поиск и при использовании флага "g" поменяет все найденные соответствия.

var myString = "This is just test string";  
result = myString.replace(/is/g,"us");

В этом примере в переменной result окажется строка  "*Th*usus*just test string*".

Еще необходимо также отметить, что в качестве второго аргумента метода replace() может использоваться **функция**, в этом случае мы получим возможность динамического изменения строки замены. 

Если в качестве второго аргумента окажется **не регулярное выражение**, то он будет также как и у метода search() преобразован в регулярное выражение с помощью конструктора RegExp. А если второй аргумент вовсе забыть указать, то будет произведена замена всех найденных совпадений на *Undefined.*

### match().

Он принимает в качестве аргумента регулярное выражение (или преобразовывает в него аргумент подобно предыдущим методам), а в качестве результата возвращает **массив** всех найденных соответствий.

var myString = "У дедушки в деревне было 12 яблонь, 5 кустов смородины, 10 кур и 33 коровы";  
result = myString.match(/\d{2}/g);

В данном примере мы записали в регулярное выражение обозначение цифры "\d", указали что ищем её двойное повторение "{2}" и включили флаг глобального поиска "g". В результате в переменной result окажется массив [12, 10, 33].

Однако если флаг глобального поиска не будет указан, то в массив попадет только первое совпадение, оно запишется нулевым элементом. Остальными элементами массива будут подстроки, соответствующие всем подвыражениям, если таковые имеются.

["12", index: 25, input: "У дедушки в деревне было 12 яблонь, 5 кустов смородины, 10 кур и 33 коровы"]

### split()

Он разбивает строку на массив подстрок, используя в качестве разделителя содержимое аргумента, которое в том числе может быть и регулярным выражением. Например, использовав в качестве разделителя два слэша можно отделить в веб-адресе протокол от собственно наименования сайта:

var myString = "http://www.example.com/download/pictures";  
result = myString.split(/\/{2}/g); // ["http:", "www.example.com/download/pictures"]

Результатом данного выражения станет массив из двух элементов: "http:" и  "www.example.com/download/pictures"

Таким же образом, например, можно разбить на массив строку состоящую из цифр:

var myString = "1234567890987654321"  
result = myString.split(/\B/g);

Используя в качестве разделителя определение символа, не являющегося границей слова (странное решение, но почему нет) получим в переменной result массив  [1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 9, 8, 7, 6, 5, 4, 3, 2, 1].

### RegExp

Во второй части урока мы более внимательно рассмотрим объект RegExp - его свойства, методы и конструктор.

Начнем с конструктора. Как вы помните, конструктор, это метод, который принимает на вход некоторый аргумент, а на выходе создает объект необходимого вида.

Конструктор RegExp может принимать на вход строку, содержимое которой будет преобразовано в регулярное выражение. В строке должно быть содержимое регулярного выражения, т.е. тот текст, который обычно находится между двух слэшей.

var myPattern = new RegExp("q$"); // Создаем шаблон, находящий букву "q" в конце строки

Также в конструктор можно передать второй, необязательный аргумент, в котором можно указать флаги. Например:

var myPattern = new RegExp("q$","g"); // Создаем такой же шаблон, но добавляем флаг глобального поиска

Важный момент! Если в тексте присутствует символ обратного слэша "\", то его необходимо предварять таким же символом (т.е. писать "\\"), поскольку, как мы помним, обратный слэш используется в регулярных выражениях для указания управляющих символов.

**Объект RegExp в JavaScript имеет следующие свойства:**

source - собственно текст регулярного выражения  
ignoreCase - логическое значение обозначающее наличие флага "i", доступно только для чтения  
global - логическое значение обозначающее наличие флага "g",  доступно только для чтения   
multiline - логическое значение обозначающее наличие флага "m",  доступно только для чтения   
lastIndex - счетчик, указывающий, с какой позиции в строке начинать поиск

**Методов у объекта RegExp всего два :**

**exec(text)** - выполнение поиска в строке, указанной в качестве параметра, возвращает массив найденных соответствий.  
**test(text)** - проверка соответствия регулярному выражению, возвращает true\false.

**Метод exec()** выполняет регулярное выражение по отношению к строке-аргументу, результатом его работы является массив, в который попадают соответствия. Если соответствий не найдено, то результатом будет null. А если соответствие есть, то оно попадает в массив нулевым элементом, при этом свойство lastIndex объекта сместится на позицию, следующую непосредственно за найденной подстрокой.

Давайте рассмотрим на примере :

var myString = "This is just a test text"; // Задаем строку для поиска

var myPattern = /te|is/g; // Задаем шаблон - либо "te" либо "is"

result = myPattern.exec(myString);

//result будет равен "is" - первому совпадению шаблона, свойство lastIndex примет значение 4

result = myPattern.exec(myString); //result == "is" - второму совпадению шаблона, lastIndex == 7

result = myPattern.exec(myString); //result == "te" - третьему совпадению шаблона, lastIndex == 17

result = myPattern.exec(myString); //result == "te" - четвертому совпадению шаблона, lastIndex == 22

В этом примере мы четыре раза подряд вызываем метод exec(), каждый раз он сдвигает указатель начала поиска на позицию, следующую за найденным совпадением и присваивает переменной result само найденное совпадение.

**Метод test()** выполняет регулярное выражение по отношению к строке-аргументу, результатом его работы является логическое значение - true если совпадение есть, и false если нет. Свойство lastIndex объекта также как и у метода exec() сместится на позицию, следующую непосредственно за найденной подстрокой.

Важный момент! Если совпадение не найдено, то lastIndex будет смещен на позицию 0 и поиск можно будет начинать сначала.

Давайте рассмотрим на таком же примере :

var myString = "This is just a test text"; // Задаем строку для поиска

var myPattern = /te|is/g; // Задаем шаблон - либо "te" либо "is"

result = myPattern.test(myString);

// result будет равен true, поскольку будет найдено первое совпадение, свойство lastIndex примет значение 4

result = myPattern.test(myString); // result == true, lastIndex == 7

result = myPattern.test(myString); // result == true, lastIndex == 17

result = myPattern.test(myString); // result == true, lastIndex == 22

result = myPattern.test(myString); // result == false, lastIndex == 0

В этом примере мы четыре раза подряд вызываем метод test(), каждый раз он сдвигает указатель начала поиска на позицию, следующую за найденным совпадением и присваивает переменной result булево значение - true если совпадение найдено и false - если нет. В последнем запуске совпадение не найдено, поэтому указатель lastIndex получает значение 0.

### Задачи

Дата в формате DD.MM.YYYY /^([0-1]\d|2[0-3])(:[0-5]\d){2}$/ => /Все кроме(0 или 1 + 0-9 | 2 + 0-3)(: + 0-5 + 0-9)2 экземпляра в конце строки/

Почтовый индекс РФ

Доменное имя

Адрес электронной почты

Код цвета в шестнадцатеричном формате

Широта или долгота

Время в формате HH:MM:SS

/^\d{6}$/ => /Все кроме 0-9 в шестизначном числе вконце строки

/-?\d{1,3}\.\d+/ => / - 1 вхождение + 0-9(не менее 1, не больше 3) + . + 0-9 >= 1 экз.

**В этом задании в нашу функцию testRegExp первым параметром передается случайная строка(переменная s), а вторым - случайная подстрока(переменная sub\_s), которую нужно использовать в качестве шаблона регулярного выражения. Вам нужно вернуть из функции строку, в которой будут перечислены через запятую все совпадения шаблона с первой строкой.**

**Sample Input 1:**

Andsirdaarrevarariarewbutovearrmararan

ar

**Sample Output 1:**

ar,ar,ar,ar,ar,ar,ar

function testRegExp(s, sub\_s) {

var myPattern = new RegExp(sub\_s, 'g');

var massiv = s.match(myPattern);

var str = massiv.join(',');

return str;

}

# Дата и Время

Для работы с датой и временем в JavaScript используются объекты Date.

## Создание

Для создания нового объекта типа Date используется один из синтаксисов:

new Date()

Создает объект Date с текущей датой и временем:

var now = new Date();

alert( now );

**new Date(milliseconds)**

Создает объект Date, значение которого равно количеству миллисекунд (1/1000 секунды), прошедших с 1 января 1970 года GMT+0.

// 24 часа после 01.01.1970 GMT+0

var Jan02\_1970 = new Date(3600 \* 24 \* 1000);

alert( Jan02\_1970 );

**new Date(datestring)**

Если единственный аргумент – строка, используется вызов Date.parse (см. далее) для чтения даты из неё.

new Date(year, month, date [, hours, minutes, seconds, ms])

Обратите внимание - параметры, указанные в квадратных скобках являются необязательными.

год - формат YYYY  
месяц - от 0 до 11  
день - от 1 до 31  
часы - от 0 до 23  
минуты - от 0 до 59  
секунды - от 0 до 59  
миллисекунды - от 0 до 999

Дату можно создать, используя компоненты в местной временной зоне. Для этого формата обязательны только первые два аргумента. Отсутствующие параметры, начиная с hours считаются равными нулю, а date – единице.

Заметим:

* Год year должен быть из 4 цифр.
* Отсчет месяцев month начинается с нуля 0.

Например:

new Date(2011, 0, 1, 0, 0, 0, 0); // // 1 января 2011, 00:00:00

new Date(2011, 0, 1); // то же самое, часы/секунды по умолчанию равны 0

Дата задана с точностью до миллисекунд:

var date = new Date(2011, 0, 1, 2, 3, 4, 567);

alert( date ); // 1.01.2011, 02:03:04.567

Давайте рассмотрим несколько примеров создания объекта Date и покажем в комментариях - что в результате будет выводиться на странице при использовании команды **document.write(myDate)**:

(команду document.write() мы еще не рассматривали подробно, на данном этапе достаточно понимания что она выводит то, что в нее передано, в HTML-документ.)

var myDate = new Date()  //Mon May 15 2017 19:20:25 GMT+0300 (RTZ 2 (зима))

var myDate = new Date("December 14, 1975 12:10:00") //Sun Dec 14 1975 12:10:00 GMT+0300 (RTZ 2 (зима))

var myDate = new Date(1989, 6, 14)  //Fri Jul 14 1989 00:00:00 GMT+0400 (RTZ 2 (лето))

var myDate = new Date(1998, 6, 14, 11, 20, 00) //Tue Jul 14 1998 11:20:00 GMT+0400 (RTZ 2 (лето))

## Получение компонентов даты

Для доступа к компонентам даты-времени объекта Date используются следующие методы:

getDay() - возвращает день недели от 0 до 6, 0 - воскресенье, 1 - понедельник и т.д.  
getTimeZoneOffset() - возвращает смещение часового пояса  относительно UTC, в минутах с противоположным знаком.  
getYear() - возвращает значение года минус 1900, к использованию не очень рекомендуется.  
getFullYear() -  возвращает значение года.   
getMonth() - возвращает месяц, от 0 до 11  
getDate() - возвращает число месяца от 1 до 31  
getHours() - возвращает час, от 0 до 23  
getMinutes() - возвращает количество минут, от 0 до 59  
getSeconds() - возвращает количество секунд, от 0 до 59  
getMilliseconds() - возвращает количество миллисекунд, от 0 до 999  
getTime() - возвращает количество миллисекунд, прошедших с полуночи 1 января 1970г GMT.

**Все методы, указанные выше, возвращают результат для местной временной зоны.**

Существуют также UTC-варианты этих методов, возвращающие день, месяц, год и т.п. для зоны GMT+0 (UTC): getUTCFullYear(), getUTCMonth(), getUTCDay(). То есть, сразу после "get" вставляется "UTC".

Если ваше локальное время сдвинуто относительно UTC, то следующий код покажет разные часы:

// текущая дата

var date = new Date();

// час в текущей временной зоне

alert( date.getHours() );

// сколько сейчас времени в Лондоне?

// час в зоне GMT+0

alert( date.getUTCHours() );

Кроме описанных выше, существуют два специальных метода без UTC-варианта:

**getTime()**

Возвращает число миллисекунд, прошедших с 1 января 1970 года GMT+0, то есть того же вида, который используется в конструкторе new Date(milliseconds).

**getTimezoneOffset()**

Возвращает разницу между местным и UTC-временем, в минутах.

alert( new Date().getTimezoneOffset() ); // Для GMT-1 выведет 60

## Установка компонентов даты

Следующие методы позволяют устанавливать компоненты даты и времени:

setYear() - устанавливает значение года минус 1900, к использованию не очень рекомендуется.  
setFullYear(year [, month, date]) - устанавливает значение года.   
setMonth(month [, date]) - устанавливает месяц, от 0 до 11  
setDate(date) - устанавливает число месяца от 1 до 31  
setHours(hour [, min, sec, ms]) - устанавливает час, от 0 до 23  
setMinutes(min [, sec, ms]) - устанавливает количество минут, от 0 до 59  
setSeconds(sec [, ms]) - устанавливает количество секунд, от 0 до 59  
setMilliseconds(ms) - устанавливает количество миллисекунд, от 0 до 999  
setTime(milliseconds) - устанавливает количество миллисекунд, прошедших с полуночи 1 января 1970г GMT.

Все они, кроме setTime(), обладают также UTC-вариантом, например: setUTCHours().

Как видно, некоторые методы могут устанавливать несколько компонентов даты одновременно, в частности, setHours. При этом если какая-то компонента не указана, она не меняется. Например:

var today = new Date;

today.setHours(0);

alert( today ); // сегодня, но час изменён на 0

today.setHours(0, 0, 0, 0);

alert( today ); // сегодня, ровно 00:00:00.

var myDate = new Date();   // Объявляем переменную типа Date

myDate.setFullYear(2017, 4, 22); // Присваиваем ей значение даты - 22 мая 2017

А теперь попробуем сдвинуть дату на 10 дней вперед:

var myDate = new Date();   // Объявляем переменную типа Date

myDate.setFullYear(2017, 4, 22); // Присваиваем ей значение даты - 22 мая 2017

myDate.setDate(myDate.getDate() + 10); // Устанавливаем новое значение даты, получив прежнее и прибавив 10.

Обратите внимание - если прибавка дней изменит месяц и год, это будет сделано автоматически, самим методом setDate().

## Автоисправление даты

*Автоисправление* – очень удобное свойство объектов Date. Оно заключается в том, что можно устанавливать заведомо некорректные компоненты (например 32 января), а объект сам себя поправит.

var d = new Date(2013, 0, 32); // 32 января 2013 ?!?

alert(d); // ... это 1 февраля 2013!

**Неправильные компоненты даты автоматически распределяются по остальным.**

Например, нужно увеличить на 2 дня дату «28 февраля 2011». Может быть так, что это будет 2 марта, а может быть и 1 марта, если год високосный. Но нам обо всем этом думать не нужно. Просто прибавляем два дня. Остальное сделает Date:

var d = new Date(2011, 1, 28);

d.setDate(d.getDate() + 2);

alert( d ); // 2 марта, 2011

Также это используют для получения даты, отдаленной от имеющейся на нужный промежуток времени. Например, получим дату на 70 секунд большую текущей:

var d = new Date();

d.setSeconds(d.getSeconds() + 70);

alert( d ); // выведет корректную дату

Можно установить и нулевые, и даже отрицательные компоненты. Например:

var d = new Date;

d.setDate(1); // поставить первое число месяца

alert( d );

d.setDate(0); // нулевого числа нет, будет последнее число предыдущего месяца

alert( d );

var d = new Date;

d.setDate(-1); // предпоследнее число предыдущего месяца

alert( d );

## Преобразование к числу, разность дат

Когда объект Date используется в числовом контексте, он преобразуется в количество миллисекунд:

alert(+new Date) // +date то же самое, что: +date.valueOf()

**Важный побочный эффект: даты можно вычитать, результат вычитания объектов Date – их временная разница, в миллисекундах**.

Это используют для измерения времени:

var start = new Date; // засекли время

// что-то сделать

for (var i = 0; i < 100000; i++) {

var doSomething = i \* i \* i;

}

var end = new Date; // конец измерения

alert( "Цикл занял " + (end - start) + " ms" );

var currentDate = new Date(); // Объявляем переменную для текущей даты

var nextNewYear = new Date(); // Объявляем переменную для даты Нового Года

nextNewYear.setFullYear(2018, 0, 1); // Записываем значение даты для Нового Года - 1 января 2018

if (currentDate == nextNewYear) {

  alert("Поздравляем с Новым, 2018-м Годом! Ура!!!");

}

## Бенчмаркинг

Допустим, у нас есть несколько вариантов решения задачи, каждый описан функцией.

Как узнать, какой быстрее?

Для примера возьмем две функции, которые бегают по массиву:

function walkIn(arr) {

for (var key in arr) arr[key]++

}

function walkLength(arr) {

for (var i = 0; i < arr.length; i++) arr[i]++;

}

Чтобы померять, какая из них быстрее, нельзя запустить один раз walkIn, один раз walkLength и замерить разницу. Одноразовый запуск ненадежен, любая мини-помеха исказит результат.

Для правильного бенчмаркинга функция запускается много раз, чтобы сам тест занял существенное время. Это сведет влияние помех к минимуму. Сложную функцию можно запускать 100 раз, простую – 1000 раз…

Померяем, какая из функций быстрее:

var arr = [];

for (var i = 0; i < 1000; i++) arr[i] = 0;

function walkIn(arr) {

for (var key in arr) arr[key]++;

}

function walkLength(arr) {

for (var i = 0; i < arr.length; i++) arr[i]++;

}

function bench(f) {

var date = new Date();

for (var i = 0; i < 10000; i++) f(arr);

return new Date() - date;

}

alert( 'Время walkIn: ' + bench(walkIn) + 'мс' );

alert( 'Время walkLength: ' + bench(walkLength) + 'мс' );

Теперь представим себе, что во время первого бенчмаркинга bench(walkIn) компьютер что-то делал параллельно важное (вдруг) и это занимало ресурсы, а во время второго – перестал. Реальная ситуация? Конечно реальна, особенно на современных ОС, где много процессов одновременно.

**Гораздо более надёжные результаты можно получить, если весь пакет тестов прогнать много раз.**

var arr = [];

for (var i = 0; i < 1000; i++) arr[i] = 0;

function walkIn(arr) {

for (var key in arr) arr[key]++;

}

function walkLength(arr) {

for (var i = 0; i < arr.length; i++) arr[i]++;

}

function bench(f) {

var date = new Date();

for (var i = 0; i < 1000; i++) f(arr);

return new Date() - date;

}

// bench для каждого теста запустим много раз, чередуя

var timeIn = 0,

timeLength = 0;

for (var i = 0; i < 100; i++) {

timeIn += bench(walkIn);

timeLength += bench(walkLength);

}

alert( 'Время walkIn: ' + timeIn + 'мс' );

alert( 'Время walkLength: ' + timeLength + 'мс' );

**Более точное время с performance.now()**

В современных браузерах (кроме IE9-) вызов [performance.now()](https://developer.mozilla.org/en-US/docs/Web/API/performance.now) возвращает количество миллисекунд, прошедшее с начала загрузки страницы. Причём именно с самого начала, до того, как загрузился HTML-файл, если точнее – с момента выгрузки предыдущей страницы из памяти.

Так что это время включает в себя всё, включая начальное обращение к серверу.

Его можно посмотреть в любом месте страницы, даже в <head>, чтобы узнать, сколько времени потребовалось браузеру, чтобы до него добраться, включая загрузку HTML.

Возвращаемое значение измеряется в миллисекундах, но дополнительно имеет точность 3 знака после запятой (до миллионных долей секунды!), поэтому можно использовать его и для более точного бенчмаркинга в том числе.

**console.time(метка) и console.timeEnd(метка)**

Для измерения с одновременным выводом результатов в консоли есть методы:

* console.time(метка) – включить внутренний хронометр браузера с меткой.
* console.timeEnd(метка) – выключить внутренний хронометр браузера с меткой и вывести результат.

Параметр "метка" используется для идентификации таймера, чтобы можно было делать много замеров одновременно и даже вкладывать измерения друг в друга.

В коде ниже таймеры walkIn, walkLength – конкретные тесты, а таймер «All Benchmarks» – время «на всё про всё»:

var arr = [];

for (var i = 0; i < 1000; i++) arr[i] = 0;

function walkIn(arr) {

for (var key in arr) arr[key]++;

}

function walkLength(arr) {

for (var i = 0; i < arr.length; i++) arr[i]++;

}

function bench(f) {

for (var i = 0; i < 10000; i++) f(arr);

}

console.time("All Benchmarks");

console.time("walkIn");

bench(walkIn);

console.timeEnd("walkIn");

console.time("walkLength");

bench(walkLength);

console.timeEnd("walkLength");

console.timeEnd("All Benchmarks");

**При запуске этого примера нужно открыть консоль, иначе вы ничего не увидите.**

**Внимание, оптимизатор!**

Современные интерпретаторы JavaScript делают массу оптимизаций, например:

1. Автоматически выносят инвариант, то есть постоянное в цикле значение типа arr.length, за пределы цикла.
2. Стараются понять, значения какого типа хранит данная переменная или массив, какую структуру имеет объект и, исходя из этого, оптимизировать внутренние алгоритмы.
3. Выполняют простейшие операции, например сложение явно заданных чисел и строк, на этапе компиляции.
4. Могут обнаружить, что некий код, например присваивание к неиспользуемой локальной переменной, ни на что не влияет и вообще исключить его из выполнения, хотя делают это редко.

Эти оптимизации могут влиять на результаты тестов, поэтому измерять скорость базовых операций JavaScript («проводить микробенчмаркинг») до того, как вы изучите внутренности JavaScript-интерпретаторов и поймёте, что они реально делают на таком коде, не рекомендуется.

## Форматирование и вывод дат

Во всех браузерах, кроме IE10-, поддерживается новый стандарт [Ecma 402](http://www.ecma-international.org/publications/standards/Ecma-402.htm), который добавляет специальные методы для форматирования дат.

Это делается вызовом date.toLocaleString(локаль, опции), в котором можно задать много настроек. Он позволяет указать, какие параметры даты нужно вывести, и ряд настроек вывода, после чего интерпретатор сам сформирует строку.

Пример с почти всеми параметрами даты и русским, затем английским (США) форматированием:

var date = new Date(2014, 11, 31, 12, 30, 0);

var options = {

era: 'long',

year: 'numeric',

month: 'long',

day: 'numeric',

weekday: 'long',

timezone: 'UTC',

hour: 'numeric',

minute: 'numeric',

second: 'numeric'

};

alert( date.toLocaleString("ru", options) ); // среда, 31 декабря 2014 г. н.э. 12:30:00

alert( date.toLocaleString("en-US", options) ); // Wednesday, December 31, 2014 Anno Domini 12:30:00 PM

Вы сможете подробно узнать о них в статье [Intl: интернационализация в JavaScript](http://learn.javascript.ru/intl), которая посвящена этому стандарту.

**Методы вывода без локализации:**

toString(), toDateString(), toTimeString() Возвращают стандартное строчное представление, не заданное жёстко в стандарте, а зависящее от браузера. Единственное требование к нему – читаемость человеком. Метод toString возвращает дату целиком, toDateString() и toTimeString() – только дату и время соответственно.

var d = new Date();

alert( d.toString() ); // вывод, похожий на 'Wed Jan 26 2011 16:40:50 GMT+0300'

toUTCString() То же самое, что toString(), но дата в зоне UTC.

toISOString() Возвращает дату в формате ISO Детали формата будут далее. Поддерживается современными браузерами, не поддерживается IE8-.

var d = new Date();

alert( d.toISOString() ); // вывод, похожий на '2011-01-26T13:51:50.417Z'

Если хочется иметь большую гибкость и кросс-браузерность, то также можно воспользоваться специальной библиотекой, например [Moment.JS](http://momentjs.com/) или написать свою функцию форматирования.

## Разбор строки, Date.parse

Все современные браузеры, включая IE9+, понимают даты в упрощённом формате ISO 8601 Extended.

Этот формат выглядит так: YYYY-MM-DDTHH:mm:ss.sssZ, где:

* YYYY-MM-DD – дата в формате год-месяц-день.
* Обычный символ T используется как разделитель.
* HH:mm:ss.sss – время: часы-минуты-секунды-миллисекунды.
* Часть 'Z' обозначает временную зону – в формате +-hh:mm, либо символ Z, обозначающий UTC. По стандарту её можно не указывать, тогда UTC, но в Safari с этим ошибка, так что лучше указывать всегда.

Также возможны укороченные варианты, например YYYY-MM-DD или YYYY-MM или даже только YYYY.

Метод Date.parse(str) разбирает строку str в таком формате и возвращает соответствующее ей количество миллисекунд. Если это невозможно, Date.parse возвращает NaN.

Например:

var msUTC = Date.parse('2012-01-26T13:51:50.417Z'); // зона UTC

alert( msUTC ); // 1327571510417 (число миллисекунд)

С таймзоной -07:00 GMT:

var ms = Date.parse('2012-01-26T13:51:50.417-07:00');

alert( ms ); // 1327611110417 (число миллисекунд)

**Или:**

var myDate = new Date();

document.write(myDate.setTime(Date.parse("22 May 2017 11:11"))); //Выведет на экран "1495613460000"

document.write(myDate);  //Выведет на экран следующий результат: Mon May 22 2017 11:11:00 GMT+0300 (RTZ 2 (зима))

**toLocaleString()** - Возвращает объект типа String содержащий дату в длинном формате, например *"10 январь 2017г. 12:26:01"*в соответствии с региональными настройками операционной системы, в которой запущен скрипт.  Внешне разница будет, например, между странами в которых распространено 12-часовое обозначение времени, по сравнению с 24-часовым.   
**toLocaleTimeString()** - преобразовывает данные о времени в строку, используя настройки форматирования операционной системы, в которой выполняется скрипт.   
**toLocaleDateString()** - выполняет преобразование, аналогичное предыдущему, но с датой

**Формат дат для IE8-**

До появления спецификации ECMAScript 5 формат не был стандартизован, и браузеры, включая IE8-, имели свои собственные форматы дат. Частично, эти форматы пересекаются.

Например, код ниже работает везде, включая старые IE:

var ms = Date.parse("January 26, 2011 13:51:50");

alert( ms );

Вы также можете почитать о старых форматах IE в документации к методу [MSDN Date.parse](http://msdn.microsoft.com/en-us/library/k4w173wk%28v=vs.85%29.aspx).

Конечно же, сейчас лучше использовать современный формат. Если же нужна поддержка IE8-, то метод Date.parse, как и ряд других современных методов, добавляется библиотекой [es5-shim](https://github.com/kriskowal/es5-shim).

## Метод Date.now()

Метод Date.now() возвращает дату сразу в виде миллисекунд.

Технически, он аналогичен вызову +new Date(), но в отличие от него не создаёт промежуточный объект даты, а поэтому – во много раз быстрее.

Его использование особенно рекомендуется там, где производительность при работе с датами критична. Обычно это не на веб-страницах, а, к примеру, в разработке игр на JavaScript.

**Итого**

* Дата и время представлены в JavaScript одним объектом: [Date](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Date/). Создать «только время» при этом нельзя, оно должно быть с датой. Список методов Date вы можете найти в справочнике [Date](http://javascript.ru/Date) или выше.
* Отсчёт месяцев начинается с нуля.
* Отсчёт дней недели (для getDay()) тоже начинается с нуля (и это воскресенье).
* Объект Date удобен тем, что автокорректируется. Благодаря этому легко сдвигать даты.
* При преобразовании к числу объект Date даёт количество миллисекунд, прошедших с 1 января 1970 UTC. Побочное следствие – даты можно вычитать, результатом будет разница в миллисекундах.
* Для получения текущей даты в миллисекундах лучше использовать Date.now(), чтобы не создавать лишний объект Date (кроме IE8-)
* Для бенчмаркинга лучше использовать performance.now() (кроме IE9-), он в 1000 раз точнее.

## Задачи

**Создайте дату**

Создайте объект Date для даты: 20 февраля 2012 года, 3 часа 12 минут.

Временная зона – местная. Выведите его на экран.

Дата в местной временной зоне создается при помощи new Date.

Месяцы начинаются с нуля, так что февраль имеет номер 1. Параметры можно указывать с точностью до минут:

var d = new Date(2012, 1, 20, 3, 12);

alert( d );

**Имя дня недели**

Создайте функцию getWeekDay(date), которая выводит текущий день недели в коротком формате „пн“, „вт“, … „вс“.

Например:

var date = new Date(2012,0,3); // 3 января 2012

alert( getWeekDay(date) ); // Должно вывести 'вт'

function getWeekDay(date) {

var days = ['вс', 'пн', 'вт', 'ср', 'чт', 'пт', 'сб'];

return days[date.getDay()];

}

В современных браузерах можно использовать и toLocaleString:

var date = new Date(2014, 0, 3); // 3 января 2014

alert( date.toLocaleString('ru', {weekday: 'short'}) ); // 'Пт'

**День недели в европейской нумерации**

Напишите функцию, getLocalDay(date) которая возвращает день недели для даты date.

День нужно возвратить в европейской нумерации, т.е. понедельник имеет номер 1, вторник номер 2, …, воскресенье – номер 7.

var date = new Date(2012, 0, 3); // 3 янв 2012

alert( getLocalDay(date) ); // вторник, выведет 2

Решение – в использовании встроенной функции getDay. Она полностью подходит нашим целям, но для воскресенья возвращает 0 вместо 7:

function getLocalDay(date) {

var day = date.getDay();

if (day == 0) { // день 0 становится 7

day = 7;

}

return day;

}

Если удобнее, чтобы день недели начинался с нуля, то можно возвращать в функции day - 1, тогда дни будут от 0 (пн) до 6(вс).

**День указанное количество дней назад**

Создайте функцию getDateAgo(date, days), которая возвращает число, которое было days дней назад от даты date.

Например, для 2 января 2015:

var date = new Date(2015, 0, 2);

alert( getDateAgo(date, 1) ); // 1, (1 января 2015)

alert( getDateAgo(date, 2) ); // 31, (31 декабря 2014)

alert( getDateAgo(date, 365) ); // 2, (2 января 2014)

P.S. Важная деталь: в процессе вычислений функция не должна менять переданный ей объект date.

Из даты date нужно вычесть указанное количество дней. Это просто:

function getDateAgo(date, days) {

date.setDate(date.getDate() - days);

return date.getDate();

}

Ситуацию осложняет то, что исходный объект даты не должен меняться. Это разумное требование, оно позволит избежать сюрпризов.

Для того чтобы ему соответствовать, создадим копию объекта даты:

function getDateAgo(date, days) {

var dateCopy = new Date(date);

dateCopy.setDate(date.getDate() - days);

return dateCopy.getDate();

}

**Последний день месяца?**

Напишите функцию getLastDayOfMonth(year, month), которая возвращает последний день месяца.

Параметры:

* year – 4-значный год, например 2012.
* month – месяц от 0 до 11.

Например, getLastDayOfMonth(2012, 1) = 29 (високосный год, февраль).

Создадим дату из следующего месяца, но день не первый, а «нулевой» (т.е. предыдущий):

function getLastDayOfMonth(year, month) {

var date = new Date(year, month + 1, 0);

return date.getDate();

}

**Сколько секунд уже прошло сегодня?**

Напишите функцию getSecondsToday() которая возвращает, сколько секунд прошло с начала сегодняшнего дня.

Например, если сейчас 10:00 и не было перехода на зимнее/летнее время, то:

getSecondsToday() == 36000 // (3600 \* 10)

Функция должна работать в любой день, т.е. в ней не должно быть конкретного значения сегодняшней даты.

Для вывода достаточно сгенерировать объект Date, соответствующий началу дня, т.е. «сегодня» 00 часов 00 минут 00 секунд, и вычесть его из текущей даты.

Полученная разница – это как раз количество миллисекунд от начала дня, которое достаточно поделить на 1000, чтобы получить секунды:

function getSecondsToday() {

var now = new Date();

// создать объект из текущей даты, без часов-минут-секунд

var today = new Date(now.getFullYear(), now.getMonth(), now.getDate());

var diff = now - today; // разница в миллисекундах

return Math.floor(diff / 1000); // перевести в секунды

}

alert( getSecondsToday() );

Альтернативное решение – получить часы/минуты/секунды и преобразовать их все в секунды:

function getSecondsToday() {

var d = new Date();

return d.getHours() \* 3600 + d.getMinutes() \* 60 + d.getSeconds();

};

**Сколько секунд - до завтра?**

Напишите функцию getSecondsToTomorrow() которая возвращает, сколько секунд осталось до завтра.

Например, если сейчас 23:00, то:

getSecondsToTomorrow() == 3600

P.S. Функция должна работать в любой день, т.е. в ней не должно быть конкретного значения сегодняшней даты.

Для получения оставшихся до конца дня миллисекунд нужно из «завтра 00 ч 00 мин 00 сек» вычесть текущее время.

Чтобы сгенерировать «завтра» – увеличим текущую дату на 1 день:

function getSecondsToTomorrow() {

var now = new Date();

// создать объект из завтрашней даты, без часов-минут-секунд

var tomorrow = new Date(now.getFullYear(), now.getMonth(), now.getDate()+1);

var diff = tomorrow - now; // разница в миллисекундах

return Math.round(diff / 1000); // перевести в секунды

}

**Вывести дату в формате дд.мм.гг**

Напишите функцию formatDate(date), которая выводит дату date в формате дд.мм.гг:

Например:

var d = new Date(2014, 0, 30); // 30 января 2014

alert( formatDate(d) ); // '30.01.14'

P.S. Обратите внимание, ведущие нули должны присутствовать, то есть 1 января 2001 должно быть 01.01.01, а не 1.1.1.

Получим компоненты один за другим.

1. День можно получить как date.getDate(). При необходимости добавим ведущий ноль:

var dd = date.getDate();

if (dd < 10) dd = '0' + dd;

1. date.getMonth() возвратит месяц, начиная с нуля. Увеличим его на 1:

var mm = date.getMonth() + 1; // месяц 1-12

if (mm < 10) mm = '0' + mm;

1. date.getFullYear() вернет год в 4-значном формате. Чтобы сделать его двузначным – воспользуемся оператором взятия остатка '%':

var yy = date.getFullYear() % 100;

if (yy < 10) yy = '0' + yy;

Заметим, что год, как и другие компоненты, может понадобиться дополнить нулем слева, причем возможно что yy == 0 (например, 2000 год). При сложении со строкой 0+'0' == '00', так что будет все в порядке.

Полный код:

function formatDate(date) {

var dd = date.getDate();

if (dd < 10) dd = '0' + dd;

var mm = date.getMonth() + 1;

if (mm < 10) mm = '0' + mm;

var yy = date.getFullYear() % 100;

if (yy < 10) yy = '0' + yy;

return dd + '.' + mm + '.' + yy;

}

**Относительное форматирование даты**

Напишите функцию formatDate(date), которая форматирует дату date так:

* Если со времени date прошло менее секунды, то возвращает "только что".
* Иначе если со времени date прошло менее минуты, то "n сек. назад".
* Иначе если прошло меньше часа, то "m мин. назад".
* Иначе полная дата в формате "дд.мм.гг чч:мм".

Например:

function formatDate(date) { /\* ваш код \*/ }

alert( formatDate(new Date(new Date - 1)) ); // "только что"

alert( formatDate(new Date(new Date - 30 \* 1000)) ); // "30 сек. назад"

alert( formatDate(new Date(new Date - 5 \* 60 \* 1000)) ); // "5 мин. назад"

alert( formatDate(new Date(new Date - 86400 \* 1000)) ); // вчерашняя дата в формате "дд.мм.гг чч:мм"

Для того, чтобы узнать время от date до текущего момента – используем вычитание дат.

function formatDate(date) {

var diff = new Date() - date; // разница в миллисекундах

if (diff < 1000) { // прошло менее 1 секунды

return 'только что';

}

var sec = Math.floor(diff / 1000); // округлить diff до секунд

if (sec < 60) {

return sec + ' сек. назад';

}

var min = Math.floor(diff / 60000); // округлить diff до минут

if (min < 60) {

return min + ' мин. назад';

}

// форматировать дату, с учетом того, что месяцы начинаются с 0

var d = date;

d = [

'0' + d.getDate(),

'0' + (d.getMonth() + 1),

'' + d.getFullYear(),

'0' + d.getHours(),

'0' + d.getMinutes()

];

for (var i = 0; i < d.length; i++) {

d[i] = d[i].slice(-2);

}

return d.slice(0, 3).join('.') + ' ' + d.slice(3).join(':');

}

**В этом задании в нашу функцию testDateTime передаются две строки вида "03 November 2017 04:17".   
Вам нужно превратить строки в даты, сравнить их. Для той, что больше получить день недели и вернуть его из функции.**

**Название дня недели должно быть по-русски, с большой буквы, например: "Понедельник".**

function testDateTime(a, b) {

var myDate = new Date( a );

var myDate2 = new Date( b );

var numDate = +myDate;

var numDate2 = +myDate2;

var numDay = myDate.getDay();

var numDay2 = myDate2.getDay();

var week = ['Воскресенье', 'Понедельник', 'Вторник', 'Среда', 'Четверг', 'Пятница', 'Суббота'];

var myDateDay = week[numDay];

var myDate2Day = week[numDay2];

return (numDate > numDate2) ? myDateDay : myDate2Day;

}

# Замыкания, область видимости

В этой главе мы продолжим рассматривать, как работают переменные, и, как следствие, познакомимся с замыканиями. От глобального объекта мы переходим к работе внутри функций.

В данной теме мы разберем основы такого понятия как **замыкания**.

Если в двух словах - **замыкание** это такая функция, которая была объявлена внутри другой функции.

Есть еще одно условие - эта функция должна иметь доступ к переменным функции, внутри которой она была объявлена.

Таким образом такая функция (**замыкание**) **имеет доступ к данным внутри себя и внутри родительской функции**.

Также внутренняя функция может обращаться не только к переменным, но и к входным параметрам своей внешней функции.

Давайте рассмотрим практический пример.

function greetPirate(pirateName) { // Объявление родительской функции

var greeting = "Hello ";  
  
  function checkCaptain() { // Объявление замыкания

if (pirateName != "Jack Sparrow")   
      return greeting + pirateName;  
    else   
      return greeting + "CAPTAIN " + pirateName + "!";  
 }  
  
  return checkCaptain();  
}

const createPrinter = () => {

const name = "King";

const printName = () => {

console.log(name);

}

return printName;

}

const myPrinter = createPrinter();

myPrinter(); //"King"

В этом примере мы написали функцию, которая выдает приветствие пирату - добавляет "Hello " к имени, которое подается на вход. Внутри это функции мы создали замыкание (checkCaptain), которое проверяет входной параметр родительской функции - если имя пирата - "Jack Sparrow", то замыкание добавляет к имени слово "CAPTAIN" и для капитана Джека Воробья выводит персональное приветствие "Hello CAPTAIN Jack Sparrow".

В результате при вызове мы получим следующий результат:

console.log(greetPirate("Mad Dog"));             // Выведет в консоль "Hello Mad Dog"

console.log(greetPirate("Jack Sparrow"));     // Выведет в консоль "Hello CAPTAIN Jack Sparrow!"

var getAnswer = function() {

var answer = 42;

return answer;

};

getAnswer(); // 42 будет каждый раз при вызову заново создавать answer

С замыканием на answer <при запуске inner answer уже не будет существовать>:

var getAnswer = (function() {

var answer = 42;

return function inner(){

return answer;

};

}())

Замыкает переменную text:

function greeting(name) {

var text = "Hello " + name;

var greet = function() {

console.log(text);

};

return greet();

}

a1 = greeting("Maxim"); // Hello Maxim;

a2 = greeting("Julia"); // Hello Julia;

У замыканий есть два важных свойства.

**1. Замыкание может обращаться к переменным своей внешней функции даже после ее окончания выполнения.**

На практике это означает что даже после того как выполнение внешней функции завершено, внутренняя все еще может быть вызвана и имеет доступ к переменным внешней функции.

Давайте рассмотрим пример:

function pirate() {  
    var pirateName = "noname";  
    return {  
        getName: function() {  
            return pirateName;  
        },  
        setName: function(newName) {  
            pirateName = newName;  
        }  
    }  
}

Мы описали функцию с двумя замыканиями: одно возвращает значение переменной из вызывающей функции, второе - изменяет его. Посмотрим, что получится при практическом использовании:

**var newPirate = pirate();**console.log(newPirate.getName());    // Выводим текущее содержимое переменной - там изначальный "noname"

newPirate.setName("Jack Sparrow"); //Изменяем значение переменной на "Jack Sparrow"

console.log(newPirate.getName());    //Выводим текущее содержимое переменной - получаем "Jack Sparrow"

В данном примере мы видим что замыкания получили доступ к переменной внешней функции после ее завершения.

**2. Замыкания хранят не содержимое переменных внешней функции, а ссылки на эти переменные.**

Давайте в этом контексте рассмотрим классический пример замыкания, описываемый в большинстве источников - счетчик.

function makeCounter(initialValue) {   
  var currentState = initialValue;  
  return function () {  
    currentState = currentState + 1;  
    return currentState;  
  }  
}

В данном случае мы описали функцию, внутри которой находится замыкание, увеличивающее каждый раз счетчик и возвращающее его.

Давайте посмотрим, что получится если вызывать ее много раз пордряд.

var myCounter = makeCounter(5); // Создаем экземпляр счетчика и устанавливаем его начальное значение = 5

console.log(myCounter()); // В консоль будет выведено значение 6

console.log(myCounter()); // В консоль будет выведено значение 7

console.log(myCounter()); // В консоль будет выведено значение 8

## Глобальный объект

*Глобальными* называют переменные и функции, которые не находятся внутри какой-то функции. То есть, иными словами, если переменная или функция не находятся внутри конструкции function, то они – «глобальные».

**В JavaScript все глобальные переменные и функции являются свойствами специального объекта, который называется *«глобальный объект»* (global object).**

В браузере этот объект явно доступен под именем window. Объект window одновременно является глобальным объектом и содержит ряд свойств и методов для работы с окном браузера, но нас здесь интересует только его роль как глобального объекта.

В других окружениях, например Node.JS, глобальный объект может быть недоступен в явном виде, но суть происходящего от этого не изменяется, поэтому далее для обозначения глобального объекта мы будем использовать "window".

**Присваивая или читая глобальную переменную, мы, фактически, работаем со свойствами window.**

Например:

var a = 5; // объявление var создаёт свойство window.a

alert( window.a ); // 5

Создать переменную можно и явным присваиванием в window:

window.a = 5;

alert( a ); // 5

**Порядок инициализации**

Выполнение скрипта происходит в две фазы:

1. На первой фазе происходит инициализация, подготовка к запуску.

Во время инициализации скрипт сканируется на предмет объявления функций вида [Function Declaration](http://learn.javascript.ru/function-declaration-expression), а затем – на предмет объявления переменных var. Каждое такое объявление добавляется в window.

**Функции, объявленные как Function Declaration, создаются сразу работающими, а переменные – равными undefined.**

1. На второй фазе – собственно, выполнение.

Присваивание (=) значений переменных происходит, когда поток выполнения доходит до соответствующей строчки кода, до этого они undefined.

В коде ниже указано содержание глобального объекта на момент инициализации и далее последовательно по коду:

// На момент инициализации, до выполнения кода:

// window = { f: function, a: undefined, g: undefined }

var a = 5;

// window = { f: function, a: 5, g: undefined }

function f(arg) { /\*...\*/ }

// window = { f: function, a: 5, g: undefined } без изменений, f обработана ранее

var g = function(arg) { /\*...\*/ };

// window = { f: function, a: 5, g: function }

Кстати, тот факт, что к началу выполнения кода переменные и функции *уже* содержатся в window, можно легко проверить, выведя их:

alert("a" in window); // true, т.к. есть свойство window.a

alert(a); // равно undefined, присваивание будет выполнено далее

alert(f); // function ..., готовая к выполнению функция

alert(g); // undefined, т.к. это переменная, а не Function Declaration

var a = 5;

function f() { /\*...\*/ }

var g = function() { /\*...\*/ };

**Присвоение переменной без объявления**

В старом стандарте JavaScript переменную можно было создать и без объявления var:

a = 5;

alert( a ); // 5

Такое присвоение, как и var a = 5, создает свойство window.a = 5. Отличие от var a = 5 – в том, что переменная будет создана не на этапе входа в область видимости, а в момент присвоения.

Сравните два кода ниже.

Первый выведет undefined, так как переменная была добавлена в window на фазе инициализации:

alert( a ); // undefined

var a = 5;

Второй код выведет ошибку, так как переменной ещё не существует:

alert( a ); // error, a is not defined

a = 5;

Это, конечно, для общего понимания, мы всегда объявляем переменные через var.

**Конструкции for, if... не влияют на видимость переменных**

Фигурные скобки, которые используются в for, while, if, в отличие от объявлений функции, имеют «декоративный» характер.

В JavaScript нет разницы между объявлением вне блока:

var i;

{

i = 5;

}

…И внутри него:

i = 5;

{

var i;

}

**Также нет разницы между объявлением в цикле и вне его:**

for (var i = 0; i < 5; i++) { }

Идентичный по функциональности код:

var i;

for (i = 0; i < 5; i++) { }

В обоих случаях переменная будет создана до выполнения цикла, на стадии инициализации, и ее значение будет сохранено после окончания цикла.

**Не важно, где и сколько раз объявлена переменная**

Объявлений var может быть сколько угодно:

var i = 10;

for (var i = 0; i < 20; i++) {

...

}

var i = 5;

**Все var будут обработаны один раз, на фазе инициализации.**

На фазе исполнения объявления var будут проигнорированы: они уже были обработаны. Зато будут выполнены присваивания.

**Ошибки при работе с window в IE8-**

В старых IE есть две забавные ошибки при работе с переменными в window:

1. Переопределение переменной, у которой такое же имя, как и id элемента, приведет к ошибке:

<div id="a">...</div>

<script>

a = 5; // ошибка в IE8-! Правильно будет "var a = 5"

alert( a ); // никогда не сработает

</script>

А если сделать через var, то всё будет хорошо.

Это была реклама того, что надо везде ставить var.

1. Ошибка при рекурсии через функцию-свойство window. Следующий код «умрет» в IE8-:

<script>

// рекурсия через функцию, явно записанную в window

window.recurse = function(times) {

if (times !== 0) recurse(times - 1);

}

recurse(13);

</script>

Проблема здесь возникает из-за того, что функция напрямую присвоена в window.recurse = .... Ее не будет при обычном объявлении функции.

**Этот пример выдаст ошибку только в настоящем IE8!** Не IE9 в режиме эмуляции. Вообще, режим эмуляции позволяет отлавливать где-то 95% несовместимостей и проблем, а для оставшихся 5% вам нужен будет настоящий IE8 в виртуальной машине.

**Итого**

В результате инициализации, к началу выполнения кода:

1. Функции, объявленные как Function Declaration, создаются полностью и готовы к использованию.
2. Переменные объявлены, но равны undefined. Присваивания выполнятся позже, когда выполнение дойдет до них.

### Задачи

**Window и переменная**

Каков будет результат кода?

if ("a" in window) {

var a = 1;

}

alert( a );

Ответ: 1.

Посмотрим, почему.

На стадии подготовки к выполнению, из var a создается window.a:

// window = {a:undefined}

if ("a" in window) { // в if видно что window.a уже есть

var a = 1; // поэтому эта строка сработает

}

alert( a ); // В результате a становится 1.

**Window и переменная 2**

Каков будет результат (перед a нет var)?

if ("a" in window) {

a = 1;

}

alert( a );

Ответ: ошибка. // <-- error!

Переменной a нет, так что условие "a" in window не выполнится. В результате на последней строчке – обращение к неопределенной переменной.

**Window и переменная 3**

Каков будет результат (перед a нет var, а ниже есть)?

if ("a" in window) {

a = 1;

}

var a;

alert( a );

Ответ: 1.

Переменная a создается до начала выполнения кода, так что условие "a" in windowвыполнится и сработает a = 1.

## Лексическое окружение

Все переменные внутри функции – это свойства специального внутреннего объекта LexicalEnvironment, который создаётся при её запуске.

Мы будем называть этот объект «лексическое окружение» или просто «объект переменных».

При запуске функция создает объект LexicalEnvironment, записывает туда аргументы, функции и переменные. Процесс инициализации выполняется в том же порядке, что и для глобального объекта, который, вообще говоря, является частным случаем лексического окружения.

В отличие от window, объект LexicalEnvironment является внутренним, он скрыт от прямого доступа.

**Пример**

Посмотрим пример, чтобы лучше понимать, как это работает:

function sayHi(name) {

var phrase = "Привет, " + name;

alert( phrase );

}

sayHi('Вася');

При вызове функции:

1. До выполнения первой строчки её кода, на стадии инициализации, интерпретатор создает пустой объект LexicalEnvironment и заполняет его.

В данном случае туда попадает аргумент name и единственная переменная phrase:

function sayHi(name) {

// LexicalEnvironment = { name: 'Вася', phrase: undefined }

var phrase = "Привет, " + name;

alert( phrase );

}

sayHi('Вася');

1. Функция выполняется.

Во время выполнения происходит присвоение локальной переменной phrase, то есть, другими словами, присвоение свойству LexicalEnvironment.phrase нового значения:

function sayHi(name) {

// LexicalEnvironment = { name: 'Вася', phrase: undefined }

var phrase = "Привет, " + name;

// LexicalEnvironment = { name: 'Вася', phrase: 'Привет, Вася'}

alert( phrase );

}

sayHi('Вася');

1. В конце выполнения функции объект с переменными обычно выбрасывается и память очищается. В примерах выше так и происходит. Через некоторое время мы рассмотрим более сложные ситуации, при которых объект с переменными сохраняется и после завершения функции.

### Тонкости спецификации

Если почитать спецификацию ECMA-262, то мы увидим, что речь идёт о двух объектах: VariableEnvironment и LexicalEnvironment.

Но там же замечено, что в реализациях эти два объекта могут быть объединены. Так что мы избегаем лишних деталей и используем везде термин LexicalEnvironment, это достаточно точно позволяет описать происходящее.

Более формальное описание находится в спецификации ECMA-262, секции 10.2-10.5 и 13.

### До[ступ ко внешним переменным](http://learn.javascript.ru/closures#доступ-ко-внешним-переменным)

Из функции мы можем обратиться не только к локальной переменной, но и к внешней:

var userName = "Вася";

function sayHi() {

alert( userName ); // "Вася"

}

**Интерпретатор, при доступе к переменной, сначала пытается найти переменную в текущем LexicalEnvironment, а затем, если её нет – ищет во внешнем объекте переменных. В данном случае им является window.**

Такой порядок поиска возможен благодаря тому, что ссылка на внешний объект переменных хранится в специальном внутреннем свойстве функции, которое называется [[Scope]]. Это свойство закрыто от прямого доступа, но знание о нём очень важно для понимания того, как работает JavaScript.

**При создании функция получает скрытое свойство [[Scope]], которое ссылается на лексическое окружение, в котором она была создана.**

В примере выше таким окружением является window, так что создаётся свойство:

sayHi.[[Scope]] = window

Это свойство никогда не меняется. Оно всюду следует за функцией, привязывая её, таким образом, к месту своего рождения.

При запуске функции её объект переменных LexicalEnvironment получает ссылку на «внешнее лексическое окружение» со значением из [[Scope]].

Если переменная не найдена в функции – она будет искаться снаружи.

Именно благодаря этой механике в примере выше alert(userName) выводит внешнюю переменную. На уровне кода это выглядит как поиск во внешней области видимости, вне функции.

Если обобщить:

* Каждая функция при создании получает ссылку [[Scope]] на объект с переменными, в контексте которого была создана.
* При запуске функции создаётся новый объект с переменными LexicalEnvironment. Он получает ссылку на внешний объект переменных из [[Scope]].
* При поиске переменных он осуществляется сначала в текущем объекте переменных, а потом – по этой ссылке.

Выглядит настолько просто, что непонятно – зачем вообще говорить об этом [[Scope]], об объектах переменных. Сказали бы: «Функция читает переменные снаружи» – и всё. Но знание этих деталей позволит нам легко объяснить и понять более сложные ситуации, с которыми мы столкнёмся далее.

### Всегда текущее значение

Значение переменной из внешней области берётся всегда текущее. Оно может быть уже не то, что было на момент создания функции.

Например, в коде ниже функция sayHi берёт phrase из внешней области:

var phrase = 'Привет';

function sayHi(name) {

alert(phrase + ', ' + name);

}

sayHi('Вася'); // Привет, Вася (\*)

phrase = 'Пока';

sayHi('Вася'); // Пока, Вася (\*\*)

На момент первого запуска (\*), переменная phrase имела значение 'Привет', а ко второму (\*\*)изменила его на 'Пока'.

Это естественно, ведь для доступа к внешней переменной функция по ссылке [[Scope]] обращается во внешний объект переменных и берёт то значение, которое там есть на момент обращения.

### Вложенные функции

Внутри функции можно объявлять не только локальные переменные, но и другие функции.

К примеру, вложенная функция может помочь лучше организовать код:

function sayHiBye(firstName, lastName) {

alert( "Привет, " + getFullName() );

alert( "Пока, " + getFullName() );

function getFullName() {

return firstName + " " + lastName;

}

}

sayHiBye("Вася", "Пупкин"); // Привет, Вася Пупкин ; Пока, Вася Пупкин

Здесь, для удобства, создана вспомогательная функция getFullName().

Вложенные функции получают [[Scope]] так же, как и глобальные. В нашем случае:

getFullName.[[Scope]] = объект переменных текущего запуска sayHiBye

Благодаря этому getFullName() получает снаружи firstName и lastName.

Заметим, что если переменная не найдена во внешнем объекте переменных, то она ищется в ещё более внешнем (через [[Scope]] внешней функции), то есть, такой пример тоже будет работать:

var phrase = 'Привет';

function say() {

function go() {

alert( phrase ); // найдёт переменную снаружи

}

go();

}

say();

### Возврат функции

Рассмотрим более «продвинутый» вариант, при котором внутри одной функции создаётся другая и возвращается в качестве результата.

В разработке интерфейсов это совершенно стандартный приём, функция затем может назначаться как обработчик действий посетителя.

Здесь мы будем создавать функцию-счётчик, которая считает свои вызовы и возвращает их текущее число.

В примере ниже makeCounter создает такую функцию:

function makeCounter() {

var currentCount = 1;

return function() { // (\*\*)

return currentCount++;

};

}

var counter = makeCounter(); // (\*)

// каждый вызов увеличивает счётчик и возвращает результат

alert( counter() ); // 1

alert( counter() ); // 2

alert( counter() ); // 3

// создать другой счётчик, он будет независим от первого

var counter2 = makeCounter();

alert( counter2() ); // 1

Как видно, мы получили два независимых счётчика counter и counter2, каждый из которых незаметным снаружи образом сохраняет текущее количество вызовов.

Где? Конечно, во внешней переменной currentCount, которая у каждого счётчика своя.

Если подробнее описать происходящее:

1. В строке (\*) запускается makeCounter(). При этом создаётся LexicalEnvironment для переменных текущего вызова. В функции есть одна переменная var currentCount, которая станет свойством этого объекта. Она изначально инициализуется в undefined, затем, в процессе выполнения, получит значение 1:

function makeCounter() {

// LexicalEnvironment = { currentCount: undefined }

var currentCount = 1;

// LexicalEnvironment = { currentCount: 1 }

return function() { // [[Scope]] -> LexicalEnvironment (\*\*)

return currentCount++;

};

}

var counter = makeCounter(); // (\*)

1. В процессе выполнения makeCounter() создаёт функцию в строке (\*\*). При создании эта функция получает внутреннее свойство [[Scope]] со ссылкой на текущий LexicalEnvironment.
2. Далее вызов makeCounter() завершается и функция (\*\*) возвращается и сохраняется во внешней переменной counter (\*).

На этом создание «счётчика» завершено.

Итоговым значением, записанным в переменную counter, является функция:

function() { // [[Scope]] -> {currentCount: 1}

return currentCount++;

};

Возвращённая из makeCounter() функция counter помнит (через [[Scope]]) о том, в каком окружении была создана.

Это и используется для хранения текущего значения счётчика.

Далее, когда-нибудь, функция counter будет вызвана. Мы не знаем, когда это произойдёт. Может быть, прямо сейчас, но, вообще говоря, совсем не факт.

Эта функция состоит из одной строки: return currentCount++, ни переменных ни параметров в ней нет, поэтому её собственный объект переменных, для краткости назовём его LE – будет пуст.

Однако, у неё есть свойство [[Scope]], которое указывает на внешнее окружение. Чтобы увеличить и вернуть currentCount, интерпретатор ищет в текущем объекте переменных LE, не находит, затем идёт во внешний объект, там находит, изменяет и возвращает новое значение:

function makeCounter() {

var currentCount = 1;

return function() {

return currentCount++;

};

}

var counter = makeCounter(); // [[Scope]] -> {currentCount: 1}

alert( counter() ); // 1, [[Scope]] -> {currentCount: 1}

alert( counter() ); // 2, [[Scope]] -> {currentCount: 2}

alert( counter() ); // 3, [[Scope]] -> {currentCount: 3}

**Переменную во внешней области видимости можно не только читать, но и изменять.**

В примере выше было создано несколько счётчиков. Все они взаимно независимы:

var counter = makeCounter();

var counter2 = makeCounter();

alert( counter() ); // 1

alert( counter() ); // 2

alert( counter() ); // 3

alert( counter2() ); // 1, счётчики независимы

Они независимы, потому что при каждом запуске makeCounter создаётся свой объект переменных LexicalEnvironment, со своим свойством currentCount, на который новый счётчик получит ссылку [[Scope]].

### Свойства функции

Функция в JavaScript является объектом, поэтому можно присваивать свойства прямо к ней, вот так:

function f() {}

f.test = 5;

alert( f.test );

Свойства функции не стоит путать с переменными и параметрами. Они совершенно никак не связаны. Переменные доступны только внутри функции, они создаются в процессе её выполнения. Это – использование функции «как функции».

А свойство у функции – доступно отовсюду и всегда. Это – использование функции «как объекта».

Если хочется привязать значение к функции, то можно им воспользоваться вместо внешних переменных.

В качестве демонстрации, перепишем пример со счётчиком:

function makeCounter() {

function counter() {

return counter.currentCount++;

};

counter.currentCount = 1;

return **counter**;

}

var counter = makeCounter();

alert( counter() ); // 1

alert( counter() ); // 2

При запуске пример работает также.

Принципиальная разница – во внутренней механике и в том, что свойство функции, в отличие от переменной из замыкания – общедоступно, к нему имеет доступ любой, у кого есть объект функции.

Например, можно взять и поменять счётчик из внешнего кода:

var counter = makeCounter();

alert( counter() ); // 1

counter.currentCount = 5;

alert( counter() ); // 5

### Статические переменные

Иногда свойства, привязанные к функции, называют «статическими переменными».

В некоторых языках программирования можно объявлять переменную, которая сохраняет значение между вызовами функции. В JavaScript ближайший аналог – такое вот свойство функции.

**Итого: замыкания**

Замыкание – это функция вместе со всеми внешними переменными, которые ей доступны.

Таково стандартное определение, которое есть в Wikipedia и большинстве серьёзных источников по программированию. То есть, замыкание – это функция + внешние переменные.

Тем не менее, в JavaScript есть небольшая терминологическая особенность.

**Обычно, говоря «замыкание функции», подразумевают не саму эту функцию, а именно внешние переменные.**

Иногда говорят «переменная берётся из замыкания». Это означает – из внешнего объекта переменных.

**Что это такое – «понимать замыкания?»**

Иногда говорят «Вася молодец, понимает замыкания!». Что это такое – «понимать замыкания», какой смысл обычно вкладывают в эти слова?

«Понимать замыкания» в JavaScript означает понимать следующие вещи:

1. Все переменные и параметры функций являются свойствами объекта переменных LexicalEnvironment. Каждый запуск функции создает новый такой объект. На верхнем уровне им является «глобальный объект», в браузере – window.
2. При создании функция получает системное свойство [[Scope]], которое ссылается на LexicalEnvironment, в котором она была создана.
3. При вызове функции, куда бы её ни передали в коде – она будет искать переменные сначала у себя, а затем во внешних LexicalEnvironment с места своего «рождения».

В следующих главах мы углубим это понимание дополнительными примерами, а также рассмотрим, что происходит с памятью.

### Задачи

**В какую переменную будет присвоено значение?**

Каков будет результат выполнения этого кода?

var value = 0;

function f() {

if (1) {

value = true;

} else {

var value = false;

}

alert( value );

}

f();

Изменится ли внешняя переменная value ?

P.S. Какими будут ответы, если из строки var value = false убрать var?

**Результатом будет true**, т.к. var обработается и переменная будет создана до выполнения кода.

Соответственно, присвоение value=true сработает на локальной переменной, и alert выведет true.

**Внешняя переменная не изменится.**

P.S. Если var нет, то в функции переменная не будет найдена. Интерпретатор обратится за ней в window и изменит её там.

**Так что без var результат будет также true, но внешняя переменная изменится.**

**var window**

Каков будет результат выполнения этого кода? Почему?

function test() {

alert( window );

var window = 5;

alert( window );

}

test();

Результатом будет undefined, затем 5.

Такой результат получился потому, что window – это глобальная переменная, но ничто не мешает объявить такую же локальную.

Директива var window обработается до начала выполнения кода функции и будет создана локальная переменная, т.е. свойство LexicalEnvironment.window:

LexicalEnvironment = {

window: undefined

}

Когда выполнение кода начнется и сработает alert, он выведет уже локальную переменную, которая на тот момент равна undefined.

Затем сработает присваивание, и второй alert выведет уже 5.

**Вызов "на месте"**

важность: 4

Каков будет результат выполнения кода? Почему?

var a = 5

(function() {

alert(a)

})()

P.S. *Подумайте хорошо! Здесь все ошибаются!* P.P.S. *Внимание, здесь подводный камень! Ок, вы предупреждены.*

Результат – **ошибка**. Попробуйте:

var a = 5

(function() {

alert(a)

})()

Дело в том, что после var a = 5 нет точки с запятой.

JavaScript воспринимает этот код как если бы перевода строки не было:

var a = 5(function() {

alert(a)

})()

То есть, он пытается вызвать *функцию* 5, что и приводит к ошибке.

Если точку с запятой поставить, все будет хорошо:

var a = 5;

(function() {

alert(a)

})()

Это один из наиболее частых и опасных подводных камней, приводящих к ошибкам тех, кто *не*ставит точки с запятой.

**Перекрытие переменной**

Если во внутренней функции есть своя переменная с именем currentCount – можно ли в ней получить currentCount из внешней функции?

function makeCounter() {

var currentCount = 1;

return function() {

var currentCount;

// можно ли здесь вывести currentCount из внешней функции (равный 1)?

};

}

Нет, нельзя.

Локальная переменная полностью перекрывает внешнюю.

**Глобальный счётчик**

Что выведут эти вызовы, если переменная currentCount находится вне makeCounter?

var currentCount = 1;

function makeCounter() {

return function() {

return currentCount++;

};

}

var counter = makeCounter();

var counter2 = makeCounter();

alert( counter() ); // ?

alert( counter() ); // ?

alert( counter2() ); // ?

alert( counter2() ); // ?

Выведут **1,2,3,4.**

Здесь внутренняя функция будет искать – и находить currentCount каждый раз в самом внешнем объекте переменных: глобальном объекте window.

В результате все счётчики будут разделять единое, глобальное текущее значение.

## [[Scope]] для new Function

**Присвоение [[Scope]] для new Function**

Есть одно исключение из общего правила присвоения [[Scope]], которое мы рассматривали в предыдущей главе.

При создании функции с использованием new Function, её свойство [[Scope]] ссылается не на текущий LexicalEnvironment, а на window.

**Пример**

Следующий пример демонстрирует как функция, созданная new Function, игнорирует внешнюю переменную a и выводит глобальную вместо неё:

var a = 1;

function getFunc() {

var a = 2;

var func = new Function('', 'alert(a)');

return func;

}

getFunc()(); // 1, из window

Сравним с обычным поведением:

var a = 1;

function getFunc() {

var a = 2;

var func = function() { alert(a); };

return func;

}

getFunc()(); // 2, из LexicalEnvironment функции getFunc

**Почему так сделано?**

**Продвинутые знания**

Содержимое этой секции содержит продвинутую информацию теоретического характера, которая прямо сейчас не обязательна для дальнейшего изучения JavaScript.

Эта особенность new Function, хоть и выглядит странно, на самом деле весьма полезна.

Представьте себе, что нам действительно нужно создать функцию из строки кода. Текст кода этой функции неизвестен на момент написания скрипта (иначе зачем new Function), но станет известен позже, например получен с сервера или из других источников данных.

Предположим, что этому коду надо будет взаимодействовать с внешними переменными основного скрипта.

Но проблема в том, что JavaScript при выкладывании на «боевой сервер» предварительно сжимается минификатором – специальной программой, которая уменьшает размер кода, убирая из него лишние комментарии, пробелы, что очень важно – переименовывает локальные переменные на более короткие.

То есть, если внутри функции есть var userName, то минификатор заменит её на var a (или другую букву, чтобы не было конфликта), предполагая, что так как переменная видна только внутри функции, то этого всё равно никто не заметит, а код станет короче. И обычно проблем нет.

…Но если бы new Function могла обращаться к внешним переменным, то при попытке доступа к userName в сжатом коде была бы ошибка, так как минификатор переименовал её.

**Получается, что даже если бы мы захотели использовать локальные переменные в new Function, то после сжатия были бы проблемы, так как минификатор переименовывает локальные переменные.**

Описанная особенность new Function просто-таки спасает нас от ошибок.

Ну а если внутри функции, создаваемой через new Function, всё же нужно использовать какие-то данные – без проблем, нужно всего лишь предусмотреть соответствующие параметры и передавать их явным образом, например так:

var sum = new Function('a, b', ' return a + b; ');

var a = 1, b = 2;

alert( sum(a, b) ); // 3

**Итого**

* Функции, создаваемые через new Function, имеют значением [[Scope]] не внешний объект переменных, а window.
* Следствие – такие функции не могут использовать замыкание. Но это хорошо, так как бережёт от ошибок проектирования, да и при сжатии JavaScript проблем не будет. Если же внешние переменные реально нужны – их можно передать в качестве параметров.

## Локальные переменные для объекта

Замыкания можно использовать сотнями способов. Иногда люди сами не замечают, что использовали замыкания – настолько это просто и естественно.

В этой главе мы рассмотрим дополнительные примеры использования замыканий и задачи на эту тему.

### Счётчик-объект

Ранее мы сделали счётчик.

Напомню, как он выглядел:

function makeCounter() {

var currentCount = 1;

return function() {

return currentCount++;

};

}

var counter = makeCounter();

// каждый вызов возвращает результат, увеличивая счётчик

alert( counter() ); // 1

alert( counter() ); // 2

alert( counter() ); // 3

Счётчик получился вполне рабочий, но вот только возможностей ему не хватает. Хорошо бы, чтобы можно было сбрасывать значение счётчика или начинать отсчёт с другого значения вместо 1 или… Да много чего можно захотеть от простого счётчика и, тем более, в более сложных проектах.

**Чтобы добавить счётчику возможностей – перейдём с функции на полноценный объект:**

function makeCounter() {

var currentCount = 1;

return { // возвратим объект вместо функции

getNext: function() {

return currentCount++;

},

set: function(value) {

currentCount = value;

},

reset: function() {

currentCount = 1;

}

};

}

var counter = makeCounter();

alert( counter.getNext() ); // 1

alert( counter.getNext() ); // 2

counter.set(5);

alert( counter.getNext() ); // 5

Теперь функция makeCounter возвращает не одну функцию, а объект с несколькими методами:

* getNext() – получить следующее значение, то, что раньше делал вызов counter().
* set(value) – поставить значение.
* reset() – обнулить счётчик.

Все они получают ссылку [[Scope]] на текущий (внешний) объект переменных. Поэтому вызов любого из этих методов будет получать или модифицировать одно и то же внешнее значение currentCount.

### Объект счётчика + функция

Изначально, счётчик делался функцией во многом ради красивого вызова: counter(), который увеличивал значение и возвращал результат.

К сожалению, при переходе на объект короткий вызов пропал, вместо него теперь counter.getNext(). Но он ведь был таким простым и удобным…

Поэтому давайте вернём его!

function makeCounter() {

var currentCount = 1;

// возвращаемся к функции

function counter() {

return currentCount++;

}

// ...и добавляем ей методы!

counter.set = function(value) {

currentCount = value;

};

counter.reset = function() {

currentCount = 1;

};

return counter;

}

var counter = makeCounter();

alert( counter() ); // 1

alert( counter() ); // 2

counter.set(5);

alert( counter() ); // 5

Красиво, не правда ли? Получился полноценный объект, который можно вдобавок ещё и вызывать.

Этот трюк часто используется при разработке JavaScript-библиотек. Например, популярная библиотека [jQuery](http://jquery.com/) предоставляет глобальную переменную с именем [jQuery](http://api.jquery.com/jQuery/) (доступна также под коротким именем $), которая с одной стороны является функцией и может вызываться как jQuery(...), а с другой – у неё есть различные методы, например jQuery.type(123) возвращает тип аргумента.

Далее вы найдёте различные задачи на понимание замыканий. Рекомендуется их сделать самостоятельно.

### Задачи

**Сумма через замыкание**

Напишите функцию sum, которая работает так: sum(a)(b) = a+b.

Да, именно так, через двойные скобки (это не опечатка). Например:

sum(1)(2) = 3

sum(5)(-1) = 4

Чтобы вторые скобки в вызове работали – первые должны возвращать функцию.

Эта функция должна знать про a и уметь прибавлять a к b. Вот так:

function sum(a) {

return function(b) {

return a + b; // возьмет a из внешнего LexicalEnvironment

};

}

**Функция - строковый буфер**

В некоторых языках программирования существует объект «строковый буфер», который аккумулирует внутри себя значения. Его функционал состоит из двух возможностей:

1. Добавить значение в буфер.
2. Получить текущее содержимое.

**Задача – реализовать строковый буфер на функциях в JavaScript, со следующим синтаксисом:**

* Создание объекта: var buffer = makeBuffer();.
* Вызов makeBuffer должен возвращать такую функцию buffer, которая при вызове buffer(value)добавляет значение в некоторое внутреннее хранилище, а при вызове без аргументов buffer() – возвращает его.

Вот пример работы:

function makeBuffer() { /\* ваш код \*/ }

var buffer = makeBuffer();

// добавить значения к буферу

buffer('Замыкания');

buffer(' Использовать');

buffer(' Нужно!');

// получить текущее значение

alert( buffer() ); // Замыкания Использовать Нужно!

Буфер должен преобразовывать все данные к строковому типу:

var buffer = makeBuffer();

buffer(0);

buffer(1);

buffer(0);

alert( buffer() ); // '010'

Решение не должно использовать глобальные переменные.

Текущее значение текста удобно хранить в замыкании, в локальной переменной makeBuffer:

function makeBuffer() {

var text = '';

return function(piece) {

if (arguments.length == 0) { // вызов без аргументов

return text;

}

text += piece;

};

};

var buffer = makeBuffer();

// добавить значения к буферу

buffer('Замыкания');

buffer(' Использовать');

buffer(' Нужно!');

alert( buffer() ); // 'Замыкания Использовать Нужно!'

var buffer2 = makeBuffer();

buffer2(0);

buffer2(1);

buffer2(0);

alert( buffer2() ); // '010'

Начальное значение text = '' – пустая строка. Поэтому операция text += piece прибавляет piece к строке, автоматически преобразуя его к строковому типу, как и требовалось в условии.

**Строковый буфер с очисткой**

Добавьте буферу из решения задачи [Функция - строковый буфер](http://learn.javascript.ru/task/stringbuffer) метод buffer.clear(), который будет очищать текущее содержимое буфера:

function makeBuffer() {

...ваш код...

}

var buffer = makeBuffer();

buffer("Тест");

buffer(" тебя не съест ");

alert( buffer() ); // Тест тебя не съест

buffer.clear();

alert( buffer() ); // ""

function makeBuffer() {

var text = '';

function buffer(piece) {

if (arguments.length == 0) { // вызов без аргументов

return text;

}

text += piece;

};

buffer.clear = function() {

text = "";

}

return buffer;

};

var buffer = makeBuffer();

buffer("Тест");

buffer(" тебя не съест ");

alert( buffer() ); // Тест тебя не съест

buffer.clear();

alert( buffer() ); // ""

**Сортировка**

У нас есть массив объектов:

var users = [{

name: "Вася",

surname: 'Иванов',

age: 20

}, {

name: "Петя",

surname: 'Чапаев',

age: 25

}, {

name: "Маша",

surname: 'Медведева',

age: 18

}];

Обычно сортировка по нужному полю происходит так:

// по полю name (Вася, Маша, Петя)

users.sort(function(a, b) {

return a.name > b.name ? 1 : -1;

});

// по полю age (Маша, Вася, Петя)

users.sort(function(a, b) {

return a.age > b.age ? 1 : -1;

});

Мы хотели бы упростить синтаксис до одной строки, вот так:

users.sort(byField('name'));

users.forEach(function(user) {

alert( user.name );

}); // Вася, Маша, Петя

users.sort(byField('age'));

users.forEach(function(user) {

alert( user.name );

}); // Маша, Вася, Петя

То есть, вместо того, чтобы каждый раз писать в sort function... – будем использовать byField(...)

Напишите функцию byField(field), которую можно использовать в sort для сравнения объектов по полю field, чтобы пример выше заработал.

var users = [{

name: "Вася",

surname: 'Иванов',

age: 20

}, {

name: "Петя",

surname: 'Чапаев',

age: 25

}, {

name: "Маша",

surname: 'Медведева',

age: 18

}];

function byField(field) {

return function(a, b) {

return a[field] > b[field] ? 1 : -1;

}

}

users.sort(byField('name'));

users.forEach(function(user) { // forEach перебор

alert( user.name );

});

users.sort(byField('age'));

users.forEach(function(user) { // forEach перебор

alert( user.name );

});

**[Фильтрация через функцию](http://learn.javascript.ru/closures-usage" \l "фильтрация-через-функцию)**

важность: 5

1. Создайте функцию filter(arr, func), которая получает массив arr и возвращает новый, в который входят только те элементы arr, для которых func возвращает true.
2. Создайте набор «готовых фильтров»: inBetween(a,b) – «между a,b», inArray([...]) – "в массиве [...]". Использование должно быть таким:

* filter(arr, inBetween(3,6)) – выберет только числа от 3 до 6,
* filter(arr, inArray([1,2,3])) – выберет только элементы, совпадающие с одним из значений массива.

Пример, как это должно работать:

/\* .. ваш код для filter, inBetween, inArray \*/

var arr = [1, 2, 3, 4, 5, 6, 7];

alert(filter(arr, function(a) {

return a % 2 == 0

})); // 2,4,6

alert( filter(arr, inBetween(3, 6)) ); // 3,4,5,6

alert( filter(arr, inArray([1, 2, 10])) ); // 1,2

**Функция фильтрации**

function filter(arr, func) {

var result = [];

for (var i = 0; i < arr.length; i++) {

var val = arr[i];

if (func(val)) {

result.push(val);

}

}

return result;

}

var arr = [1, 2, 3, 4, 5, 6, 7];

alert(filter(arr, function(a) {

return a % 2 == 0;

})); // 2, 4, 6

**Фильтр inBetween**

function filter(arr, func) {

var result = [];

for (var i = 0; i < arr.length; i++) {

var val = arr[i];

if (func(val)) {

result.push(val);

}

}

return result;

}

function inBetween(a, b) {

return function(x) {

return x >= a && x <= b;

};

}

var arr = [1, 2, 3, 4, 5, 6, 7];

alert( filter(arr, inBetween(3, 6)) ); // 3,4,5,6

**Фильтр inArray**

function filter(arr, func) {

var result = [];

for (var i = 0; i < arr.length; i++) {

var val = arr[i];

if (func(val)) {

result.push(val);

}

}

return result;

}

function inArray(arr) {

return function(x) {

return arr.indexOf(x) != -1;

};

}

var arr = [1, 2, 3, 4, 5, 6, 7];

alert( filter(arr, inArray([1, 2, 10])) ); // 1,2

**Армия функций**

Следующий код создает массив функций-стрелков shooters. По замыслу, каждый стрелок должен выводить свой номер:

function makeArmy() {

var shooters = [];

for (var i = 0; i < 10; i++) {

var shooter = function() { // функция-стрелок

alert( i ); // выводит свой номер

};

shooters.push(shooter);

}

return shooters;

}

var army = makeArmy();

army[0](); // стрелок выводит 10, а должен 0

army[5](); // стрелок выводит 10...

// .. все стрелки выводят 10 вместо 0,1,2...9

Почему все стрелки́ выводят одно и то же? Поправьте код, чтобы стрелки работали как задумано. Предложите несколько вариантов исправления.

**Что происходит в этом коде**

Функция makeArmy делает следующее:

1. Создаёт пустой массив shooters:

var shooters = [];

1. В цикле заполняет массив элементами через shooters.push. При этом каждый элемент массива – это функция, так что в итоге после цикла массив будет таким:

shooters = [

function () { alert(i); },

function () { alert(i); },

function () { alert(i); },

function () { alert(i); },

function () { alert(i); },

function () { alert(i); },

function () { alert(i); },

function () { alert(i); },

function () { alert(i); },

function () { alert(i); }

];

Этот массив возвращается из функции.

1. Вызов army[5]() – это получение элемента массива (им будет функция), и тут же – её запуск.

**Почему ошибка**

Вначале разберемся, почему все стрелки выводят одно и то же значение.

В функциях-стрелках shooter отсутствует переменная i. Когда такая функция вызывается, то iона берет из внешнего LexicalEnvironment.

Чему же будет равно это значение i?

К моменту вызова army[0](), функция makeArmy уже закончила работу. Цикл завершился, последнее значение было i=10.

В результате все функции shooter получают из внешнего лексического окружения это, одно и то же, последнее, значение i=10.

Попробуйте исправить проблему самостоятельно.

**Исправление (3 варианта)**

Есть несколько способов исправить ситуацию.

1. **Первый способ исправить код – это привязать значение непосредственно к функции-стрелку:**

function makeArmy() {

var shooters = [];

for (var i = 0; i < 10; i++) {

var shooter = function me() {

alert( me.i );

};

shooter.i = i;

shooters.push(shooter);

}

return shooters;

}

var army = makeArmy();

army[0](); // 0

army[1](); // 1

В этом случае каждая функция хранит в себе свой собственный номер.

Кстати, обратите внимание на использование Named Function Expression, вот в этом участке:

...

var shooter = function me() {

alert( me.i );

};

...

Если убрать имя me и оставить обращение через shooter, то работать не будет:

for (var i = 0; i < 10; i++) {

var shooter = function() {

alert( shooter.i ); // вывести свой номер (не работает!)

// потому что откуда функция возьмёт переменную shooter?

// ..правильно, из внешнего объекта, а там она одна на всех

};

shooter.i = i;

shooters.push(shooter);

}

Вызов alert(shooter.i) при вызове будет искать переменную shooter, а эта переменная меняет значение по ходу цикла, и к моменту вызова она равна последней функции, созданной в цикле.

Если использовать Named Function Expression, то имя жёстко привязывается к конкретной функции, и поэтому в коде выше me.i возвращает правильный i.

1. **Другое, более продвинутое решение – использовать дополнительную функцию для того, чтобы «поймать» текущее значение i**:

function makeArmy() {

var shooters = [];

for (var i = 0; i < 10; i++) {

var shooter = (function(x) {

return function() {

alert( x );

};

})(**i**);

shooters.push(shooter);

}

return shooters;

}

var army = makeArmy();

army[0](); // 0

army[1](); // 1

Посмотрим выделенный фрагмент более внимательно, чтобы понять, что происходит:

var shooter = (function(x) {

return function() {

alert( x );

};

})(i);

Функция shooter создана как результат вызова промежуточного функционального выражения function(x), которое объявляется – и тут же выполняется, получая x = i.

Так как function(x) тут же завершается, то значение x больше не меняется. Оно и будет использовано в возвращаемой функции-стрелке.

Для красоты можно изменить название переменной x на i, суть происходящего при этом не изменится:

var shooter = (function(i) {

return function() {

alert( i );

};

})(i);

**Кстати, обратите внимание – скобки вокруг function(i) не нужны**, можно и так:

var shooter = function(i) { // без скобок вокруг function(i)

return function() {

alert( i );

};

}(i);

Скобки добавлены в код для лучшей читаемости, чтобы человек, который просматривает его, не подумал, что var shooter = function, а понял что это вызов «на месте», и присваивается его результат.

1. **Еще один забавный способ – обернуть весь цикл во временную функцию**:

function makeArmy() {

var shooters = [];

for (var i = 0; i < 10; i++)(function(i) {

var shooter = function() {

alert( i );

};

shooters.push(shooter);

})(i);

return shooters;

}

var army = makeArmy();

army[0](); // 0

army[1](); // 1

Вызов (function(i) { ... }) обернут в скобки, чтобы интерпретатор понял, что это Function Expression.

Плюс этого способа – в большей читаемости. Фактически, мы не меняем создание shooter, а просто обертываем итерацию в функцию.

## Модули через замыкания

Приём программирования «модуль» имеет громадное количество вариаций. Он немного похож на счётчик, который мы рассматривали ранее, использует аналогичный приём, но на уровне выше.

Его цель – скрыть внутренние детали реализации скрипта. В том числе: временные переменные, константы, вспомогательные мини-функции и т.п.

### Зачем нужен модуль?

Допустим, мы хотим разработать скрипт, который делает что-то полезное на странице.

Умея работать со страницей, мы могли бы сделать много чего, но так как пока этого не было (скоро научимся), то пусть скрипт просто выводит сообщение:

Файл hello.js

// глобальная переменная нашего скрипта

var message = "Привет";

// функция для вывода этой переменной

function showMessage() {

alert( message );

}

// выводим сообщение

showMessage();

У этого скрипта есть свои внутренние переменные и функции.

В данном случае это message и showMessage.

Предположим, что мы хотели бы распространять этот скрипт в виде библиотеки. Каждый, кто хочет, чтобы посетителям выдавалось «Привет» – может просто подключить этот скрипт. Достаточно скачать и подключить, например, как внешний файл hello.js – и готово.

**Если подключить подобный скрипт к странице «как есть», то возможен конфликт с переменными, которые она использует.**

То есть, при подключении к такой странице он её «сломает»:

<script>

var message = "Пожалуйста, нажмите на кнопку";

</script>

<script src="hello.js"></script>

<button>Кнопка</button>

<script>

// ожидается сообщение из переменной выше...

alert( message ); // но на самом деле будет выведено "Привет"

</script>

Автор страницы ожидает, что библиотека "hello.js" просто отработает, без побочных эффектов. А она вместе с этим переопределила message в "Привет".

Если же убрать скрипт hello.js, то страница будет выводить правильное сообщение.

Зная внутреннее устройство hello.js нам, конечно, понятно, что проблема возникла потому, что переменная message из скрипта hello.js перезаписала объявленную на странице.

### Приём проектирования «Модуль»

Чтобы проблемы не было, всего-то нужно, чтобы у скрипта была *своя собственная область видимости*, чтобы его переменные не попали на страницу.

Для этого мы завернём всё его содержимое в функцию, которую тут же запустим.

Файл hello.js, оформленный как модуль:

(function() {

// глобальная переменная нашего скрипта

var message = "Привет";

// функция для вывода этой переменной

function showMessage() {

alert( message );

}

// выводим сообщение

showMessage();

})();

Этот скрипт при подключении к той же странице будет работать корректно.

Будет выводиться «Привет», а затем «Пожалуйста, нажмите на кнопку».

**Зачем скобки вокруг функции?**

В примере выше объявление модуля выглядит так:

(function() {

alert( "объявляем локальные переменные, функции, работаем" );

// ...

}());

В начале и в конце стоят скобки, так как иначе была бы ошибка.

Вот, для сравнения, неверный вариант:

function() {

// будет ошибка

}();

Ошибка при его запуске произойдет потому, что браузер, видя ключевое слово function в основном потоке кода, попытается прочитать Function Declaration, а здесь имени нет.

Впрочем, даже если имя поставить, то работать тоже не будет:

function work() {

// ...

}(); // syntax error

**Дело в том, что «на месте» разрешено вызывать *только* Function Expression.**

Общее правило таково:

* Если браузер видит function в основном потоке кода – он считает, что это Function Declaration.
* Если же function идёт в составе более сложного выражения, то он считает, что это Function Expression.

Для этого и нужны скобки – показать, что у нас Function Expression, который по правилам JavaScript можно вызвать «на месте».

Можно показать это другим способом, например поставив перед функцией оператор:

+function() {

alert('Вызов на месте');

}();

!function() {

alert('Так тоже будет работать');

}();

### Экспорт значения

Приём «модуль» используется почти во всех современных библиотеках.

Ведь что такое библиотека? Это полезные функции, ради которых её подключают, плюс временные переменные и вспомогательные функции, которые библиотека использует внутри себя.

Посмотрим, к примеру, на библиотеку [Lodash](http://lodash.com/), хотя могли бы и [jQuery](http://jquery.com/), там почти то же самое.

Если её подключить, то появится специальная переменная lodash (короткое имя \_), которую можно использовать как функцию, и кроме того в неё записаны различные полезные свойства, например:

* \_.defaults(src, dst1, dst2...) – копирует в объект src те свойства из объектов dst1, dst2 и других, которых там нет.
* \_.cloneDeep(obj) – делает глубокое копирование объекта obj, создавая полностью независимый клон.
* \_.size(obj) – возвращает количество свойств в объекте, полиморфная функция: можно передать массив или даже 1 значение.

Есть и много других функций, подробнее описанных в [документации](https://lodash.com/docs).

Пример использования:

<p>Подключим библиотеку</p>

<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"></script>

<p>Функция <code>\_.defaults()</code> добавляет отсутствующие свойства.</p>

<script>

var user = {

name: 'Вася'

};

\_.defaults(user, {

name: 'Не указано',

employer: 'Не указан'

});

alert( user.name ); // Вася

alert( user.employer ); // Не указан

alert( \_.size(user) ); // 2

</script>

Здесь нам не важно, какие функции или методы библиотеки используются, нас интересует именно как описана эта библиотека, как в ней применяется приём «модуль».

Вот примерная выдержка из исходного файла:

;(function() {

// lodash - основная функция для библиотеки

function lodash(value) {

// ...

}

// вспомогательная переменная

var version = '2.4.1';

// ... другие вспомогательные переменные и функции

// код функции size, пока что доступен только внутри

function size(collection) {

return Object.keys(collection).length;

}

// присвоим в lodash size и другие функции, которые нужно вынести из модуля

lodash.size = size

// lodash.defaults = ...

// lodash.cloneDeep = ...

// "экспортировать" lodash наружу из модуля

window.\_ = lodash; // в оригинальном коде здесь сложнее, но смысл тот же

}());

Внутри внешней функции:

1. Происходит что угодно, объявляются свои локальные переменные, функции.
2. В window выносится то, что нужно снаружи.

Технически, мы могли бы вынести в window не только lodash, но и вообще все объекты и функции. На практике, как раз наоборот, всё прячут внутри модуля, глобальную область во избежание конфликтов хранят максимально чистой.

**Зачем точка с запятой в начале?**

В начале кода выше находится точка с запятой ; – это не опечатка, а особая «защита от дураков».

Если получится, что несколько JS-файлов объединены в один (и, скорее всего, сжаты минификатором, но это не важно), и программист забыл поставить точку с запятой, то будет ошибка.

Например, первый файл a.js:

var a = 5

Второй файл lib.js:

(function() {

// без точки с запятой в начале

})()

После объединения в один файл:

var a = 5

// библиотека

(function() {

// ...

})();

При запуске будет ошибка, потому что интерпретатор перед скобкой сам не вставит точку с запятой. Он просто поймёт код как var a = 5(function ...), то есть пытается вызвать число 5как функцию.

Таковы правила языка, и поэтому рекомендуется явно ставить точку с запятой. В данном случае автор lodash ставит ; перед функцией, чтобы предупредить эту ошибку.

### Экспорт через return

Можно оформить модуль и чуть по-другому, например передать значение через return:

var lodash = (function() {

var version;

function assignDefaults() { ... }

return {

defaults: function() { }

}

})();

Здесь, кстати, скобки вокруг внешней function() { ... } не обязательны, ведь функция и так объявлена внутри выражения присваивания, а значит – является Function Expression.

Тем не менее, лучше их ставить, для улучшения читаемости кода, чтобы было сразу видно, что это не простое присвоение функции.

**Итого**

Модуль при помощи замыканий – это оборачивание пакета функционала в единую внешнюю функцию, которая тут же выполняется.

Все функции модуля будут иметь доступ к другим переменным и внутренним функциям этого же модуля через замыкание.

Например, defaults из примера выше имеет доступ к assignDefaults.

Но снаружи программист, использующий модуль, может обращаться напрямую только к тем переменным и функциям, которые экспортированы. Благодаря этому будут скрыты внутренние аспекты реализации, которые нужны только разработчику модуля.

Можно придумать и много других вариаций такого подхода. В конце концов, «модуль» – это всего лишь функция-обёртка для скрытия переменных.

## Управление памятью в JavaScript

Управление памятью в JavaScript обычно происходит незаметно. Мы создаём примитивы, объекты, функции… Всё это занимает память.

Что происходит с объектом, когда он становится «не нужен»? Возможно ли «переполнение» памяти? Для ответа на эти вопросы – залезем «под капот» интерпретатора.

### Управление памятью в JavaScript

Главной концепцией управления памятью в JavaScript является принцип достижимости (англ. reachability).

1. Определённое множество значений считается достижимым изначально, в частности:

* Значения, ссылки на которые содержатся в стеке вызова, то есть – все локальные переменные и параметры функций, которые в настоящий момент выполняются или находятся в ожидании окончания вложенного вызова.
* Все глобальные переменные.

Эти значения гарантированно хранятся в памяти. Мы будем называть их корнями.

1. **Любое другое значение сохраняется в памяти лишь до тех пор, пока доступно из корня по ссылке или цепочке ссылок.**

Для очистки памяти от недостижимых значений в браузерах используется автоматический [Сборщик мусора](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)) (англ. Garbage collection, GC), встроенный в интерпретатор, который наблюдает за объектами и время от времени удаляет недостижимые.

Самая простая ситуация здесь с примитивами. При присвоении они копируются целиком, ссылок на них не создаётся, так что если в переменной была одна строка, а её заменили на другую, то предыдущую можно смело выбросить.

Именно объекты требуют специального «сборщика мусора», который наблюдает за ссылками, так как на один объект может быть много ссылок из разных переменных и, при перезаписи одной из них, объект может быть всё ещё доступен из другой.

Далее мы посмотрим ряд примеров, которые помогут в этом разобраться.

### Достижимость и наличие ссылок

Есть одно упрощение для работы с памятью: «значение остаётся в памяти, пока на него есть хотя бы одна ссылка».

Но такое упрощение будет верным лишь в одну сторону.

* **Верно – в том плане, что если ссылок на значение нет, то память из-под него очищается.**

Например, была создана ссылка в переменной, и эту переменную тут же перезаписали:

var user = {

name: "Вася"

};

user = null;

Теперь объект { name: "Вася" } более недоступен. Память будет освобождена.

* **Неверно – в другую сторону: наличие ссылки не гарантирует, что значение останется в памяти.**

Такая ситуация возникает с объектами, которые ссылаются друг на друга:

var vasya = {};

var petya = {};

vasya.friend = petya;

petya.friend = vasya;

vasya = petya = null;

Несмотря на то, что объекты vasya и petya ссылаются друг на друга через ссылку friend, то есть можно сказать, что на каждый из них есть ссылка, последняя строка делает эти объекты в совокупности недостижимыми.

Поэтому они будут удалены из памяти.

Здесь как раз и играет роль «достижимость» – оба этих объекта становятся недостижимы из корней, в первую очередь, из глобальной области, стека.

[Сборщик мусора](https://ru.wikipedia.org/wiki/%D0%A1%D0%B1%D0%BE%D1%80%D0%BA%D0%B0_%D0%BC%D1%83%D1%81%D0%BE%D1%80%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) отслеживает такие ситуации и очищает память.

### Алгоритм сборки мусора

Сборщик мусора идёт от корня по ссылкам и запоминает все найденные объекты. По окончанию – он смотрит, какие объекты в нём отсутствуют и удаляет их.

Например, рассмотрим пример объекта «семья»:

function marry(man, woman) {

woman.husband = man;

man.wife = woman;

return {

father: man,

mother: woman

}

}

var family = marry({

name: "Василий"

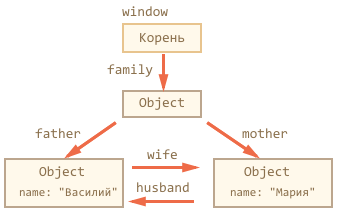
}, {

name: "Мария"

});

Функция marry принимает два объекта, даёт им ссылки друг на друга и возвращает третий, содержащий ссылки на оба.

Получившийся объект family можно изобразить так:



Здесь стрелочками показаны ссылки, а вот свойство name ссылкой не является, там хранится примитив, поэтому оно внутри самого объекта.

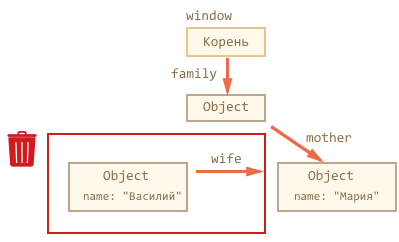
Чтобы запустить сборщик мусора, удалим две ссылки:

delete family.father;

delete family.mother.husband;

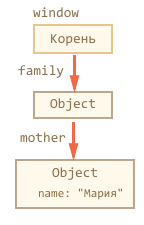
Обратим внимание, удаление только одной из этих ссылок ни к чему бы не привело. Пока до объекта можно добраться из корня window, объект остаётся жив.

А если две, то получается, что от бывшего family.father ссылки выходят, но в него – ни одна не идёт:



**Совершенно неважно, что из объекта выходят какие-то ссылки, они не влияют на достижимость этого объекта.**

Бывший family.father стал недостижимым и будет удалён вместе со своими данными, которые также более недоступны из программы.

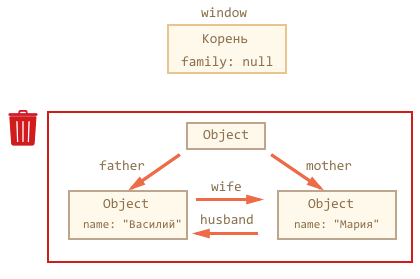


А теперь – рассмотрим более сложный случай. Что будет, если удалить главную ссылку family?

Исходный объект – тот же, что и в начале, а затем:

window.family = null;

Результат:



Как видим, объекты в конструкции всё ещё связаны между собой. Однако, поиск от корня их не находит, они не достижимы, и значит сборщик мусора удалит их из памяти.

### Оптимизации

Проблема описанного алгоритма – в больших задержках. Если объектов много, то на поиск всех достижимых уйдёт довольно много времени. А ведь выполнение скрипта при этом должно быть остановлено, уже просканированные объекты не должны поменяться до окончания процесса. Получаются небольшие, но неприятные паузы-зависания в работе скрипта.

Поэтому современные интерпретаторы применяют различные оптимизации.

Самая частая – это деление объектов на два вида «старые» и «новые». Для каждого типа выделяется своя область памяти. Каждый объект создаётся в «новой» области и, если прожил достаточно долго, мигрирует в старую. «Новая» область обычно небольшая. Она очищается часто. «Старая» – редко.

На практике получается эффективно, обычно большинство объектов создаются и умирают почти сразу, к примеру, служа локальными переменными функции:

function showTime() {

alert( new Date() ); // этот объект будет создан и умрёт сразу

}

Если вы знаете низкоуровневые языки программирования, то более подробно об организации сборки мусора в V8 можно почитать, например, в статье [A tour of V8: Garbage Collection](http://jayconrod.com/posts/55/a-tour-of-v8-garbage-collection).

### Замыкания

Объекты переменных, о которых шла речь ранее, в главе про замыкания, также подвержены сборке мусора. Они следуют тем же правилам, что и обычные объекты.

Объект переменных внешней функции существует в памяти до тех пор, пока существует хоть одна внутренняя функция, ссылающаяся на него через свойство [[Scope]].

Например:

Обычно объект переменных удаляется по завершении работы функции. Даже если в нём есть объявление внутренней функции:

function f() {

var value = 123;

function g() {} // g видна только изнутри

}

f();

В коде выше value и g являются свойствами объекта переменных. Во время выполнения f() её объект переменных находится в текущем стеке выполнения, поэтому жив. По окончанию, он станет недостижимым и будет убран из памяти вместе с остальными локальными переменными.

…А вот в этом случае лексическое окружение, включая переменную value, будет сохранено:

function f() {

var value = 123;

function g() {}

return g;

}

var g = f(); // функция g будет жить и сохранит ссылку на объект переменных

В скрытом свойстве g.[[Scope]] находится ссылка на объект переменных, в котором была создана g. Поэтому этот объект переменных останется в памяти, а в нём – и value.

Если f() будет вызываться много раз, а полученные функции будут сохраняться, например, складываться в массив, то будут сохраняться и объекты LexicalEnvironment с соответствующими значениями value:

function f() {

var value = Math.random();

return function() { return value; };

}

// 3 функции, каждая ссылается на свой объект переменных,

// каждый со своим значением value

var arr = [f(), f(), f()];

Объект LexicalEnvironment живёт ровно до тех пор, пока на него существуют ссылки. В коде ниже после удаления ссылки на g умирает:

function f() {

var value = 123;

function g() {}

return g;

}

var g = f(); // функция g жива

// а значит в памяти остается соответствующий объект переменных f()

g = null; // ..а вот теперь память будет очищена

### Оптимизация в V8 и её последствия

Современные JS-движки делают оптимизации замыканий по памяти. Они анализируют использование переменных и в случае, когда переменная из замыкания абсолютно точно не используется, удаляют её.

В коде выше переменная value никак не используется. Поэтому она будет удалена из памяти.

**Важный побочный эффект в V8 (Chrome, Opera) состоит в том, что удалённая переменная станет недоступна и при отладке!**

Попробуйте запустить пример ниже с открытой консолью Chrome. Когда он остановится, в консоли наберите alert(value).

function f() {

var value = Math.random();

function g() {

debugger; // выполните в консоли alert( value ); Нет такой переменной!

}

return g;

}

var g = f();

g();

Как вы могли увидеть – нет такой переменной! Недоступна она изнутри g. Интерпретатор решил, что она нам не понадобится и удалил.

Это может привести к забавным казусам при отладке, вплоть до того что вместо этой переменной будет другая, внешняя:

var value = "Сюрприз";

function f() {

var value = "самое близкое значение";

function g() {

debugger; // выполните в консоли alert( value ); Сюрприз!

}

return g;

}

var g = f();

g();

**Ещё увидимся**

Об этой особенности важно знать. Если вы отлаживаете под Chrome/Opera, то наверняка рано или поздно с ней встретитесь!

Это не глюк отладчика, а особенность работы V8, которая, возможно, будет когда-нибудь изменена. Вы всегда сможете проверить, не изменилось ли чего, запустив примеры на этой странице.

### Влияние управления памятью на скорость

На создание новых объектов и их удаление тратится время. Это важно иметь в виду в случае, когда важна производительность.

В качестве примера рассмотрим рекурсию. При вложенных вызовах каждый раз создаётся новый объект с переменными и помещается в стек. Потом память из-под него нужно очистить. Поэтому рекурсивный код будет всегда медленнее использующего цикл, но насколько?

Пример ниже тестирует сложение чисел до данного через рекурсию по сравнению с обычным циклом:

function sumTo(n) { // обычный цикл 1+2+...+n

var result = 0;

for (var i = 1; i <= n; i++) {

result += i;

}

return result;

}

function sumToRec(n) { // рекурсия sumToRec(n) = n+SumToRec(n-1)

return n == 1 ? 1 : n + sumToRec(n - 1);

}

var timeLoop = performance.now();

for (var i = 1; i < 1000; i++) sumTo(1000); // цикл

timeLoop = performance.now() - timeLoop;

var timeRecursion = performance.now();

for (var i = 1; i < 1000; i++) sumToRec(1000); // рекурсия

timeRecursion = performance.now() - timeRecursion;

alert( "Разница в " + (timeRecursion / timeLoop) + " раз" );

Различие в скорости на таком примере может составлять, в зависимости от интерпретатора, 2-10 раз.

Вообще, этот пример – не показателен. Ещё раз обращаю ваше внимание на то, что такие искусственные «микротесты» часто врут. Правильно их делать – отдельная наука, которая выходит за рамки этой главы. Но и на практике ускорение в 2-10 раз оптимизацией по количеству объектов (и вообще, любых значений) – отнюдь не миф, а вполне достижимо.

В реальной жизни в большинстве ситуаций такая оптимизация несущественна, просто потому что «JavaScript и так достаточно быстр». Но она может быть эффективной для «узких мест» кода.

## Устаревшая конструкция "with"

Конструкция with позволяет использовать в качестве области видимости для переменных произвольный объект.

В современном JavaScript от этой конструкции отказались. С use strict она не работает, но её ещё можно найти в старом коде, так что стоит познакомиться с ней, чтобы если что – понимать, о чём речь.

Синтаксис:

with(obj) {

...код...

}

Любое обращение к переменной внутри with сначала ищет её среди свойств obj, а только потом – вне with.

**Пример**

В примере ниже переменная будет взята не из глобальной области, а из obj:

var a = 5;

var obj = {

a: 10

};

with(obj) {

alert( a ); // 10, из obj

}

Попробуем получить переменную, которой в obj нет:

var b = 1;

var obj = {

a: 10

};

with(obj) {

alert( b ); // 1, из window

}

Здесь интерпретатор сначала проверяет наличие obj.b, не находит и идет вне with.

Особенно забавно выглядит применение вложенных with:

var obj = {

weight: 10,

size: {

width: 5,

height: 7

}

};

with(obj) {

with(size) { // size будет взят из obj

alert( width \* height / weight ); // width,height из size, weight из obj

}

}

Свойства из разных объектов используются как обычные переменные… Магия! Порядок поиска переменных в выделенном коде: size => obj => window.

### Изменения переменной

При использовании with, как и во вложенных функциях – переменная изменяется в той области, где была найдена.

Например:

var obj = {

a: 10

}

with(obj) {

a = 20;

}

alert( obj.a ); // 20, переменная была изменена в объекте

### Почему отказались от with?

Есть несколько причин.

1. В современном стандарте JavaScript отказались от with, потому что конструкция with подвержена ошибкам и непрозрачна.

Проблемы возникают в том случае, когда в with(obj) присваивается переменная, которая по замыслу должна быть в свойствах obj, но ее там нет.

Например:

var obj = {

weight: 10

};

with(obj) {

weight = 20; // (1)

size = 35; // (2)

}

alert( obj.size ); //undefined

alert( window.size ); //35

В строке (2) присваивается свойство, отсутствующее в obj. В результате интерпретатор, не найдя его, создает новую глобальную переменную window.size.

Такие ошибки редки, но очень сложны в отладке, особенно если size изменилась не в window, а где-нибудь во внешнем LexicalEnvironment.

1. Еще одна причина – алгоритмы сжатия JavaScript не любят with. Перед выкладкой на сервер JavaScript сжимают. Для этого есть много инструментов, например [Closure Compiler](http://code.google.com/intl/ru-RU/closure/compiler/) и [UglifyJS](https://github.com/mishoo/UglifyJS). Обычно они переименовывают локальные переменные в более короткие имена, но не свойства объектов. С конструкцией with до запуска кода непонятно – откуда будет взята переменная. Поэтому выходит, что, на всякий случай (если это свойство), лучше её не переименовывать. Таким образом, качество сжатия кода страдает.
2. Ну и, наконец, производительность – усложнение поиска переменной из-за with влечет дополнительные накладные расходы.

Современные движки применяют много внутренних оптимизаций, ряд которых не может быть применен к коду, в котором есть with.

Вот, к примеру, запустите этот код в современном браузере. Производительность функции fastсущественно отличается slow с пустым(!) with. И дело тут именно в with, т.к. наличие этой конструкции препятствует оптимизации.

var i = 0;

function fast() {

i++;

}

function slow() {

with(i) {}

i++;

}

var time = performance.now();

while (i < 1000000) fast();

alert( "Без with: " + (performance.now() - time) );

var time = performance.now();

i = 0;

while (i < 1000000) slow();

alert( "С with: " + (performance.now() - time) );

### Замена with

Вместо with рекомендуется использовать временную переменную, например:

/\* вместо

with(elem.style) {

top = '10px';

left = '20px';

}

\*/

var s = elem.style;

s.top = '10px';

s.left = '0';

Это не так элегантно, но убирает лишний уровень вложенности и абсолютно точно понятно, что будет происходить и куда присвоятся свойства.

**Итого**

* Конструкция with(obj) { ... } использует obj как дополнительную область видимости. Все переменные, к которым идет обращение внутри блока, сначала ищутся в obj.
* Конструкция with устарела и не рекомендуется по ряду причин. Избегайте её.

### Задачи

**With + функция**

Какая из функций будет вызвана?

function f() {

alert(1)

}

var obj = {

f: function() {

alert(2)

}

};

with(obj) {

f();

}

Вторая (2), т.к. при обращении к любой переменной внутри with – она ищется прежде всего в объекте.

Соответственно, будет выведено 2:

**With + переменные**

Что выведет этот код?

var a = 1;

var obj = {

b: 2

};

with(obj) {

var b;

alert( a + b );

}

Выведет 3.

**Конструкция with не создаёт области видимости,** её создают только функции. Поэтому объявление var b внутри конструкции работает также, как если бы оно было вне её.

Код в задаче эквивалентен такому:

var a = 1;

var b;

var obj = {

b: 2

}

with(obj) {

alert( a + b );

}

# Область видимости (Don`t know JS)

## Область видимости

Имя переменной должно быть уникальным в рамках одной и той же области видимости — не может быть двух различных переменных a, расположенных рядом друг с другом. Но одно и тоже имя переменной a может появляться в разных областях видимости.

function one() {

// эта `a` принадлежит только функции `one()`

var a = 1;

console.log( a );

}

function two() {

// эта `a` принадлежит только функции `two()`

var a = 2;

console.log( a );

}

one(); // 1

two(); // 2

Если одна область вложена в другую, для кода внутри самой внутренней области доступны переменные из окружающей области:

function outer() {

var a = 1;

function inner() {

var b = 2;

// здесь у нас есть доступ и к `a`, и к `b`

console.log( a + b ); // 3

}

inner();

// здесь у нас есть доступ только к `a`

console.log( a ); // 1

}

outer();

Правила лексической области видимости говорят, что код в одной области может иметь доступ к переменным как её самой, так и к переменным любой области снаружи этой области.

Таким образом, код внутри функции inner() имеет доступ к обеим переменным a и b, но у кода в outer() есть доступ только к a — у него нет доступа к b потому что эта переменная внутри inner().

## Поднятие переменной (hoisting)

Где бы ни появлялось var внутри области видимости, это объявление принадлежит всей области видимости и доступно везде в ней.

Метафорически это поведение называется *поднятие (hoisting)*, когда объявление var концептуально "перемещается" на вершину своей объемлющей области видимости. Технически этот процесс более точно объясняется тем, как компилируется код, но сейчас опустим эти подробности.

Пример:

var a = 2;

foo(); // работает, так как определение `foo()`

// "всплыло"

function foo() {

a = 3;

console.log( a ); // 3

var a; // определение "всплыло"

// наверх `foo()`

}

console.log( a ); // 2

Или:

var a = 2;

(function() {

console.log(a);

var a = 3;

})() //undefined

**Равносильно этому:**

var a = 2;

(function() {

var a;

console.log(a);

a = 3;

})();

Еще примеры:

function foo(b) {

a = b \* 10; // если переменная объявлена без var => она становится глобальной.

return a;

}

foo(2); // 20

function bar() {

console.log(a);

}

bar(); // 20

a //20

если перед a будет стоять var, то запустив функцию bar() => ReferenceError (переменная не найдена)

## Вложенные области видимости

Когда вы объявляете переменную, она доступна везде в ее области видимости, также как и в более нижних/внутренних областях видимости. Например:

function foo() {

var a = 1;

function bar() {

var b = 2;

function baz() {

var c = 3;

console.log( a, b, c ); // 1 2 3

}

baz();

console.log( a, b ); // 1 2

}

bar();

console.log( a ); // 1

}

foo();

Заметьте, что c не доступна внутри bar(), потому что она объявлена только внутри внутренней области видимости baz() и b не доступна в foo() по той же причине.

Если вы попытаетесь получить доступ к значению переменной в области видимости, где она уже недоступна, вы получите ReferenceError. Если вы попытаетесь установить значение переменной, которая еще не объявлена, у вас либо закончится тем, что переменная создастся в самой верхней глобальной области видимости (плохо!), либо получите ошибку в зависимости от "строгого режима" .

В дополнение к созданию объявлений переменных на уровне функций, ES6 *позволяет* вам объявлять переменные, принадлежащие отдельным блокам (пара { .. }), используя ключевое слово let. Кроме некоторых едва уловимых деталей, правила области видимости будут вести себя точно также, как мы видели в функциях:

function foo() {

var a = 1;

if (a >= 1) {

let b = 2;

while (b < 5) {

let c = b \* 2;

b++;

console.log( a + c );

}

}

}

foo(); // 5 7 9

Из-за использования let вместо var, b будет принадлежать только оператору if и следовательно не всей области видимости функции foo(). Точно также c принадлежит только циклу while. Блочная область видимости очень полезна для управления областями ваших переменных более точно, что может сделать ваш код более легким в обслуживании в долгосрочной перспективе.

## Замыкание

*Замыкание* — одно из самых важных и зачастую наименее понятных концепций в JavaScript. Я не буду вдаваться в подробности сейчас, а вместо этого направляю вас в книгу *Область видимости и замыкания* этой серии.

function makeAdder(x) {

// параметр `x` - внутренная переменная

// внутренняя функция `add()` использует `x`, поэтому

// у нее есть "замыкание" на нее

function add(y) {

return y + x;

};

return add;

}

Ссылка на внутреннюю функцию add(..), которая возвращается с каждым вызовом внешней makeAdder(..), умеет запоминать какое значение x было передано в makeAdder(..). Теперь, давайте используем makeAdder(..):

// `plusOne` получает ссылку на внутреннюю функцию `add(..)`

// с замыканием на параметре `x`

// внешней `makeAdder(..)`

var plusOne = makeAdder( 1 );

// `plusTen` получает ссылку на внутреннюю функцию `add(..)`

// с замыканием на параметре `x`

// внешней `makeAdder(..)`

var plusTen = makeAdder( 10 );

plusOne( 3 ); // 4 <-- 1 + 3

plusOne( 41 ); // 42 <-- 1 + 41

plusTen( 13 ); // 23 <-- 10 + 13

Теперь подробней о том, как работает этот код:

1. Когда мы вызываем makeAdder(1), мы получаем обратно ссылку на ее внутреннюю add(..), которая запоминает x как 1. Мы назвали эту ссылку на функцию plusOne(..).
2. Когда мы вызываем makeAdder(10), мы получаем обратно ссылку на ее внутреннюю add(..), которая запоминает x как 10. Мы назвали эту ссылку на функцию plusTen(..).
3. Когда мы вызываем plusOne(3), она прибавляет 3 (свою внутреннюю y) к 1 (которая запомнена в x) и мы получаем в качестве результата 4.
4. Когда мы вызываем plusTen(13), она прибавляет 13 (свою внутреннюю y) к 10 (которая запомнена в x), и мы получаем в качестве результата 23.

Еще пример:

function foo(a) {

var c= a \* 2;

var b = function bar() {

var d = c – 5;

return d;

};

return b;

}

var e1 = foo(10); //ссылка на функцию

var e2 = foo(20); // ссылка на функцию

var g = e1(); //15

var h = e2(); //35

## Модули

Самое распространенное использование замыкания в JavaScript — это модульный шаблон. Модули позволяют определять частные детали реализации (переменные, функции), которые скрыты от внешнего мира, а также публичное API, которое *доступно* снаружи.

Представим:

function User(){

var username, password;

function doLogin(user,pw) {

username = user;

password = pw;

// сделать остальную часть работы по логину

}

var publicAPI = {

login: doLogin

};

return publicAPI;

}

// создать экземпляр модуля `User`

var fred = User();

fred.login( "fred", "12Battery34!" );

Функция User() служит как внешняя область видимости, которая хранит переменные username и password, а также внутреннюю функцию doLogin(). Всё это частные внутренние детали этого модуля User, которые недоступны из внешнего мира.

**Предупреждение:** Мы не вызываем тут new User() намеренно, несмотря на тот факт, что это будет более естественно для большинства читателей. User() — просто функция, а не класс, поэтому она вызывается обычным образом. Использование new было бы неуместным и еще и тратой попусту ресурсов.

При выполнении User() создается *экземпляр* модуля User и создается целая новая область видимости и также совершенно новая копия каждой из этих внутренних переменных/функций. Мы присваиваем этот экземпляр в fred. Если мы запустим User() снова, мы получим новый экземпляр, целиком отдельный от fred.

У внутренней функции doLogin() есть замыкание на username и password, что значит, что она сохранит свой доступ к ним даже после того, как функция User() завершит свое выполнение.

publicAPI — это объект с одним свойством/методом, login, который является ссылкой на внутреннюю функцию doLogin(). Когда вы возвращаем publicAPI из User(), он становится экземпляром, который мы назвали fred.

На данный момент, внешняя функция User() закончила выполнение. Как правило, вы думаете, что внутренние переменные, такие как username и password, при этом исчезают. Но они никуда не деваются, потому что есть замыкание в функции login(), хранящее их.

Вот поэтому мы можем вызвать fred.login(..), что тоже самое, что вызвать внутреннюю doLogin(..) и у нее все еще будет доступ ко внутренним переменным username и password.

Есть большой шанс, что кратко бросив взгляд на замыкание и шаблон модуля, кое-что останется неясным. Ничего страшного! Понадобится некоторая работа, чтобы намотать всё это на ус.

## Идентификатор this

Еще одна очень часто неверно понимаемая концепция в JavaScript — это идентификатор this. И опять таки, есть пара глав по нему в книге this и прототипы объектов этой серии, поэтому здесь мы только кратко его рассмотрим.

При том что может часто казаться, что этот this связан с "объектно-ориентированным шаблонами," в JS this — это другой механизм.

Если у функции есть внутри ссылка this, эта ссылка this обычно указывает на объект. Но на какой объект она указывает зависит от того, как эта функция была вызвана.

Важно представлять, что this не ссылается на саму функцию, учитывая, что это самое распространенное неверное представление.

Вот краткая иллюстрация:

function foo() {

console.log( this.bar );

}

var bar = "global";

var obj1 = {

bar: "obj1",

foo: foo

};

var obj2 = {

bar: "obj2"

};

//--------

foo(); // "global"

obj1.foo(); // "obj1"

foo.call( obj2 ); // "obj2"

new foo(); // undefined

Есть четыре правила того, как устанавливается this и они показаны в этих четырех последних строках кода.

foo() заканчивается установкой this в глобальный объект в нестрогом режиме. В строгом режиме, this будет undefined и вы получите ошибку при доступе к свойству bar, поэтому "global" — это значение для this.bar.

obj1.foo() устанавливает this в объект obj1.

foo.call(obj2) устанавливает this в объект obj2.

new foo() устанавливает this в абсолютно новый пустой объект.

Резюме: чтобы понять на что указывает this, вы должны проверить как вызывалась на самом деле функция. Это будет один из тех четырех путей, только что показанных, и это то, что поможет потом ответить на вопрос что будет в this.

## Прототипы

Механизм прототипов в JavaScript довольно сложен. Здесь мы только взглянем на него немного. Когда вы ссылаетесь на свойство объекта, то если это свойство не существует, JavaScript автоматически использует ссылку на внутренний прототип этого объекта, чтобы найти другой объект, чтобы поискать свойство там. Можете думать об этом почти как о резервном варианте когда свойство отсутствует.

Связывание ссылки на внутренний прототип от объекта к его резервному варианту происходит в момент когда объект создается. Простейший путь проиллюстрировать это — с помощью вызова встроенной функции Object.create(..).

Пример:

var foo = {

a: 42

};

// создаем `bar` и связываем его с `foo`

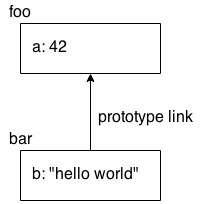
var bar = Object.create( foo );

bar.b = "hello world";

bar.b; // "hello world"

bar.a; // 42 <-- делегируется в `foo`

Следующая картинка поможет визуально показать объекты foo и bar и их связь:

[](https://github.com/azat-io/you-dont-know-js-ru/blob/master/up%20&%20going/fig6.png)

Свойство a в действительности не существует в объекте bar, но поскольку bar прототипно связан с foo, JavaScript автоматически прибегает к поиску a в объекте foo, где оно и находится.

Такая связь может показаться странной возможностью языка. Самый распространенный способ ею пользоваться, и я бы поспорил, что неправильно пытаться эмулировать механизм "классов" "наследование".

Но более естественный путь применения прототипов — шаблон, называемый "делегирование поведения", когда вы намеренно проектируете свои связанные объекты так, чтобы они могли *делегировать* от одного к другому части необходимого поведения.

Есть две основные техники, которыми можно пользоваться, чтобы "привнести" более новые возможности JavaScript в старые браузеры: полифиллинг (polyfilling) и транспиляция (transpiling).

## Полифиллинг (polyfilling)

Слово "polyfill" — изобретенный термин (Реми Шарпом) (<https://remysharp.com/2010/10/08/what-is-a-polyfill>), используется для указания на взятие определения новой возможности и генерации кода, эквивалентного этому поведению, но с возможностью запуска в более старых окружениях JS.

Например, ES6 определяет функцию, называемую Number.isNaN(..) для обеспечения точной безошибочной проверки на значения NaN, отмечая как устаревшую исходную функцию isNaN(..). Но очень легко заполифиллить эту функцию, чтобы вы могли пользоваться ею в вашем коде независимо от того, поддерживает браузер ES6 или нет.

Пример:

if (!Number.isNaN) {

Number.isNaN = function isNaN(x) {

return x !== x;

};

}

Оператор if защищает против применения полифильного определения в браузерах с ES6, где функция уже есть. Если она еще не существует, мы определяем Number.isNaN(..).

**Примечание:** Проверка, которую мы тут выполняем, использует преимущество причудливости значения NaN, которое заключается в том, что оно является единственным значением во всем языке, которое не равно самому себе. Поэтому значение NaN — единственное, которое может сделать условие x !== x истинным.

Не все новые возможностью полностью полифильны. Иногда большая часть поведения может быть сполифиллена, но еще есть пока что небольшие отступления. Вы должны быть очень, очень осторожны реализуя полифиллинг сами, следя за тем, чтобы придерживаться спецификации настолько строго, насколько возможно.

Или даже лучше используйте уже проверенный набор полифиллов, которому вы можете доверять, вроде тех, что предоставляются ES5-Shim (<https://github.com/es-shims/es5-shim>) и ES6-Shim (<https://github.com/es-shims/es6-shim>).

## Транспиляция (Transpiling)

Не существует возможности полифиллить новый синтаксис, который был добавлен в язык. Новый синтаксис вызовет ошибку в старом движке JS как нераспознанный/невалидный.

Поэтому лучшим выбором будет использовать утилиту, которая конвертирует ваш более новый код в эквивалент более старого. Этот процесс обычно называют "транспиляцией", как объединение терминов трансформация и компиляция (transforming + compiling).

По большому счету, ваш исходный код написан в новом синтаксисе, но то, что вы развертываете в браузере — это транспилированный код со старым синтаксисом. Вы обычно вставляете транспилятор в ваш процесс сборки, примерно так же как linter или minifier.

Вы могли бы удивиться, а почему идете на неприятности, чтобы писать только в новом синтаксисе, чтобы потом транспилировать его в старый код? Почему бы просто не писать напрямую в старом синтаксисе?

Есть несколько важных причин, чтобы вы позаботились о транспиляции:

* Новый синтаксис, добавленный в язык, разрабатывается, чтобы заставить ваш код быть более читаемым и обслуживаемым. Старые эквиваленты часто намного более запутанны. Следует предпочитать писать с помощью более нового и ясного синтаксиса, не только для себя, но и для всех остальных членов команды разработки.
* Если вы транспилируете только для старых браузеров, но используете новый синтаксис в новейших браузерах, вы получаете преимущество оптимизации производительности браузера с помощью нового синтаксиса. Это также позволяет разработчикам браузеров делать код более приближенным к жизни для проверки их реализаций и оптимизаций.
* Использование нового синтаксиса как можно раньше позволяет ему быть протестированным более тесно в реальном мире, что обеспечивает более ранние отзывы в комитет JavaScript (TC39). Если проблемы обнаружены достаточно рано, их можно изменить/устранить до того, как эти ошибки дизайна языка станут постоянными.

Вот быстрый пример транспиляции. ES6 добавляет возможность, называемую "значения параметров по умолчанию". Это выглядит примерно так:

function foo(a = 2) {

console.log( a );

}

foo(); // 2

foo( 42 ); // 42

Просто, правда? Еще и полезно! Но это как раз новый синтаксис, который будет считаться невалидным в до-ES6 движках. Так что же транспилятор сделает с этим кодом, чтобы заставить его работать в более старых движках?

function foo() {

var a = arguments[0] !== (void 0) ? arguments[0] : 2;

console.log( a );

}

Как видите, он проверяет, что значение arguments[0] — void 0 (т.е. undefined) и если да, то предоставляет значение по умолчанию 2, иначе он присваивает то, что было передано.

В дополнение к тому, что теперь можно использовать привлекательный синтаксис даже в старых браузерах, транспилированный код фактически делает заданное поведение яснее.

Возможно вы даже не представляли себе просто глядя на версию ES6, что undefined — единственное значение, которое не может быть явно задано значением по умолчанию для параметра, но транспилированный код показывает это гораздо лучше.

Последняя важная деталь, чтобы сделать акцент на транспиляторах — то, что о них следует думать как о стандартной части экосистемы и процесса разработки на JS. JS будет продолжать эволюционировать, намного быстрее, чем прежде, поэтому каждые несколько месяцев будут добавляться новый синтаксис и новые возможности.

Если вы по умолчанию используете транспилятор, вы всегда сможете переключиться на новый синтаксис, когда бы ни захотели, нежели всегда ждать годы, чтобы сегодняшние браузеры вышли из употребления.

Есть довольно много отличных транспиляторов для выбора. Вот несколько из них на момент написания этого текста:

* Babel ([https://babeljs.io](https://babeljs.io/)) (бывший 6to5): Транспилирует из ES6+ в ES5
* Traceur (<https://github.com/google/traceur-compiler>): Транспилирует из ES6, ES7 и далее в ES5

**Не-JavaScript**

На данный момент, мы рассмотрели только вещи, касающиеся самого языка JS. Реальность такова, что большая часть JS написана для запуска и взаимодействия с такими средами как браузеры. Добрая часть вещей, которые вы пишете в своем коде, строго говоря, не контролируется напрямую JavaScript. Возможно это звучит несколько странно.

Самый распространенный не-JavaScript JavaScript, с которым вы столкнетесь — это DOM API. Например:

var el = document.getElementById( "foo" );

Переменная document существует как глобальная переменная, когда ваш код выполняется в браузере. Она не обеспечивается ни движком JS, ни особенно не контролируется спецификацией JavaScript. Она принимает форму чего-то очень ужасного похожего на обычный JS объект, но не является им на самом деле. Это — специальный объект, часто называемый "хост-объектом."

Более того, метод getElementById(..) в document выглядит как обычная функция JS, но это всего лишь кое-как открытый интерфейс к встроенному методу, предоставляемому DOM из вашего браузера. В некоторых (нового поколения) браузерах этот слой может быть на JS, но традиционно DOM и его поведение реализовано на чем-то вроде C/C++.

Еще один пример с вводом/выводом (I/O).

Всеобщее любимое всплывающее окно alert(..) в пользовательском окне браузера. alert(..) предоставляется вашей программе на JS браузером, а не самим движком JS. Вызов, который вы делаете, отправляет сообщение во внутренности браузера и они обрабатывают отрисовку и отображение окна с сообщением.

То же происходит и с console.log(..) — ваш браузер предоставляет такие механизмы и подключает их к средствам разработчика.

Эта книга и вся эта серия фокусируется на языке JavaScript. Поэтому вы не увидите какого-либо подробного раскрытия деталей об этих не-JavaScript механизмах JavaScript. Как бы то ни было, вам не нужно забывать о них, поскольку они будут в каждой программе на JS, которую вы напишете!

## Глава 1: Что такое область видимости?

*Область видимости* - определенный набор правил для хранения переменных в некоем месте и для обнаружения эти переменных позднее.

Но где и как правила этих *областей видимости* устанавливаются?

**Теория компиляторов**

Это может быть самоочевидно или это может удивлять, в зависимости от вашего уровня взаимодействия с различных языками, но несмотря на тот факт, что JavaScript подпадает под общую категорию "динамических" или "интерпретируемых" языков, на самом деле он является компилируемым языком. Он *не* компилируется заранее, как многие традиционно компилируемые языки, и результаты компиляции не являются переносимыми среди различных распределенных систем.

Но, тем ни менее, среда исполнения JavaScript выполняет много тех же шагов, что и любой традиционный компилятор языка, хоть и более сложными способами, чем мы обычно можем представить.

В традиционном процессе языковой компиляции, часть кода вашей программы обычно проходит три шага *до* того, как будет выполнена, в общих чертах называемых "компиляцией":

1. **Разбиение на** лексемы **(Tokenizing/Lexing)**: разбиение строки символов на имеющие смысл (для языка) части, называемые лексемами. Например, представьте программу: var a = 2;. Эта программа, вполне вероятно, будет разбита на следующие лексемы: var, a, =, 2 и ;. Пробел может быть сохранен или не сохранен как лексема в зависимости от того имеет он смысл или нет.

**Примечание:** Разница между tokenizing и lexing — едва различима и теоретическая, но она сосредотачивается на том, идентифицируются ли эти лексемы как *без состояния* или *с состоянием*. Проще говоря, если токенизатор используется, чтобы вызывать правила парсинга с сохранением состояния для выяснения следует ли считать отдельной лексемой или только частью другой лексемы, *это* будет **lexing**.

1. **Парсинг:** берет поток (массив) лексем и превращает его в дерево вложенных элементов, которые сообща представляют грамматическую структуру программы. Это дерево называется "AST" (**A**bstract **S**yntax **T**ree, дерево абстрактного синтаксиса).

Такое дерево для var a = 2; может начинаться с узла верхнего уровня с названием VariableDeclaration, с дочерним узлом Identifier (чье значение равно a) и еще одним дочерним узлом AssignmentExpression, у которого тоже есть дочерний узел NumericLiteral (чье значение равно 2).

1. **Генерация кода:** процесс взятия AST и превращения его в исполняемый код. Эта часть сильно зависит от языка, платформы назначения и т.п..

Итак, вместо того, чтобы увязать в деталях, мы просто опустим их и скажем, что есть способ взять наше вышеописанное AST для var a = 2; и превратить его в набор машинных инструкций, чтобы в действительности *создать* переменную с именем a (включая выделение памяти и т.д.), а затем сохранить значение в a.

**Примечание:** подробности того, как движок управляет системными ресурсами, глубже, чем мы будем "копать", поэтому мы всего лишь примем на веру, что движок умеет создавать и сохранять переменные, когда необходимо.

Движок JavaScript гораздо сложнее, чем *только* эти три шага, как и большинство других языковых компиляторов. Например, в процессе парсинга и генерации кода безусловно есть шаги по оптимизации быстродействия выполнения, включая удаление лишних элементов.

Поэтому здесь я лишь очерчиваю границы. Но я думаю, что вы скоро увидите почему *эти* детали, которые мы *обязательно* рассмотрим, хоть и на более высоком уровне, связаны.

Для начала, движки JavaScript не балуют роскошью (как другие языковые компиляторы) затрат массы времени на оптимизацию, так как компиляция JavaScript не происходит на шаге сборки заранее, как в других языках.

Для JavaScript, компиляция во многих случаях происходит за всего лишь микросекунды (или меньше!) перед выполнением кода. Чтобы гарантировать высочайшее быстродействие, движки JS используют все виды уловок (такие как JIT, который компилирует лениво и даже перекомпилирует на ходу), которые вне "области" нашего обсуждения тут.

Давайте скажем, простоты ради, что любой код JavaScript должен быть скомпилирован до (обычно *прямо* перед!) его выполнения. Поэтому, компилятор JS возьмет программу var a = 2; и *сперва* скомпилирует ее, а потом будет готов выполнить ее, обычно сразу же.

**Понимание области видимости**

Путь, которым мы будем приближаться к обучению области видимости, это думать о процессе в терминах диалога. Но, *кто* ведет этот диалог?

**Действующие лица**

Встречайте действующих лиц, которые взаимодействуют, чтобы обработать программу var a = 2;. Чтобы мы могли понять о чем их диалоги мы немного подслушаем их:

1. Движок: отвечает за компиляцию от начала до конца и выполнение нашей JavaScript программы.
2. Компилятор: один из друзей *Движка*, выполняет всю грязную работу по парсингу и генерации кода (см. предыдущий раздел).
3. Область видимости: еще один друг *Движка*; собирает и обслуживает список поиска всех объявленных идентификаторов (переменных), и следит за исполнением строгого набора правил о том, как эти идентификаторы доступны для кода, выполняемого в текущий момент.

Для *полного понимания* как работает JavaScript, вам необходимо начать *думать* как *Движок* (и его друзья), задавать вопросы, которые задают они и отвечать на них также.

**Туда и обратно**

Когда вы видите программу var a = 2; вы вероятнее всего подумаете о ней как об одном операторе. Но наш новый друг *Движок* видит это не так. На самом деле, *Движок* видит два отдельных оператора, один, который *Компилятор* обработает во время компиляции, а другой, который *Движок* обработает во время выполнения.

Так давайте же разберем по полочкам как *Движок* и его друзья поступят с программой var a = 2;.

Первая вещь, которую сделает *Компилятор* с этой программой, выполнит разбиение на лексемы, которые он затем распарсит в дерево. Но когда *Компилятор* доберется до генерации кода, он будет интерпретировать программу несколько по-другому нежели предполагалось.

Обоснованным предположением будет то, что *Компилятор* породит код, который можно кратко представить следующим псевдо-кодом: "Выделить память для переменной, пометить ее как a, затем поместить значение 2 в эту переменную." К сожалению, это не совсем точно.

*Компилятор* вместо этого сделает следующее:

1. Встретив var a, *Компилятор* просит *Область видимости* посмотреть существует ли уже переменная a в коллекции указанной области видимости. Если да, то *Компилятор* игнорирует это объявление переменной и двигается дальше. В противном случае, *Компилятор* просит *Область видимости* объявить новую переменную в коллекции указанной области видимости.
2. Затем *Компилятор* генерирует код для *Движка* для последующего выполнения, чтобы обработать присваивание a = 2. Код, который *Движок* запускает, сначала спрашивает *Область видимости* есть ли переменная с именем a, доступная в коллекции текущей области видимости. Если да, то *Движок* использует эту переменную. Если нет, то *Движок* ищет *в другом месте* (см. раздел *Вложенная область видимости* ниже).

Если *Движок* в итоге находит переменную, он присваивает ей значение 2. Если нет, то *Движок* вскинет руки и выкрикнет "ошибка!"!

Подводя итоги: для присваивания значения переменной выполняется два отдельных действия: первое, *Компилятор* объявляет переменную (если не была объявлена до этого в текущей области видимости), и второе, когда выполняет код, *Движок* ищет эту переменную в *Области видимости* и присваивает ей значение, если находит.

**Компилятор расскажет**

Нам нужно еще немного компиляторной терминологии, чтобы двинуться дальше.

Когда *Движок* выполняет код, который *Компилятор* генерирует на шаге (2), он должен поискать переменную a, чтобы увидеть была ли она объявлена и этот поиск принимает во внимание *Область видимости*. Но тип поиска, который выполняет *Движок*, влияет на результат поиска.

В нашем случае, он говорит, что *Движок* выполнит "LHS"-поиск переменной a. Другой тип поиска называется "RHS".

Держу пари, что вы можете угадать что значат "L" и "R". Эти термины означают "Left-hand Side" (левая сторона) и "Right-hand Side" (правая сторона).

Сторона... чего? **Операции присваивания.**

Иными словами, LHS-поиск выполняется, когда переменная появляется с левой стороны операции присваивания, а RHS-поиск выполняется, когда переменная появляется с правой стороны операции присваивания.

На самом деле, давайте будем более точны. RHS-поиск неотличим, для наших целей, от простого поиска значения некоторой переменной, тогда как LHS-поиск пытается найти сам контейнер переменной, чтобы он мог присвоить значение. Таким образом, RHS не *обязательно* означает "правая сторона присваивания" по существу, он просто более точно означает "не левая сторона".

Прикинувшись немного поверхностным на минуту, вы можете подумать, что "RHS" вместо этого значит "retrieve his/her source (value)" (получить его/ее исходное значение), представляя, что RHS означает "иди и возьми значение из...".

Давайте копнем немного глубже в этом направлении.

Когда я говорю:

console.log( a );

Ссылка на a — это RHS-ссылка, потому что здесь ничего не присваивается в a. Напротив, мы выполняем поиск, чтобы извлечь значение a, для того, чтобы передать значение в console.log(..).

Для сравнения:

a = 2;

Ссылка на a здесь — это LHS-ссылка, так как мы не заботимся здесь о том, каково текущее значение, мы просто хотим найти эту переменную как цель для операции присваивания = 2.

**Примечание:** LHS и RHS, означающие "левая/правая сторона присваивания", не обязательно буквально означают "левая/правая сторона операции присваивания =". Есть еще несколько способов, которыми производится присваивание, и поэтому лучше концептуально думать о нем как: "кто является целью присваивания (LHS)" и "кто источник присваивания (RHS)".

Представьте такую программу, в которой есть обе ссылки LHS и RHS:

function foo(a) {

console.log( a ); // 2

}

foo( 2 );

Последняя строка, которая активизирует foo(..) как вызов функции, требует RHS-ссылку на foo, что значит, "сходи и найди значение foo и дай его мне". Более того, (..) означает, что значение foo должно быть выполнено, поэтому это скорее всего функция!

Здесь есть едва уловимое, но важное присваивание. **Вы обнаружили его?**

Вы наверное упустили неявное a = 2 в этом коде. Это происходит, когда значение 2 передается как аргумент в функцию foo(..), в этом случае значение 2 **присваивается** параметру a. Чтобы (неявно) присвоить значение параметру a, выполняется LHS-поиск.

Также есть и RHS-ссылка на значение a и это результирующее значение передается в console.log(..). console.log(..) нужна ссылка для выполнения. Для объекта console это RHS-поиск, затем происходит разрешение имени свойства чтобы убедиться существует ли метод, называемый log.

Наконец, мы можем осмыслить, что есть LHS/RHS-обмен передаваемым значением 2 (путем RHS-поиска переменной a) в log(..). Внутри родной реализации log(..), мы можем предположить, что у нее есть параметры, у первого из которых (возможно называющегося arg1) есть поиск LHS-ссылки, до присваивания ему 2.

**Примечание:** У вас может появиться соблазн представлять объявление функции function foo(a) {... как обычное объявление переменной и присваивание, такое как var foo и foo = function(a){.... Делая так, будет соблазн думать об объявлении этой функции как подразумевающей LHS-поиск.

Однако, едва заметная, но важная разница есть в том, что *Компилятор* обрабатывает как объявление, так и определение значения во время генерации кода, благодаря чему, когда *Движок* выполняет код, не требуется никакой обработки чтобы "присвоить" значение функции в foo. Следовательно, неуместно думать об объявлении функции как о присваивании с помощью LHS-поиска тем способом, который мы здесь обсуждаем.

**Беседа Движка и Области видимости**

function foo(a) {

console.log( a ); // 2

}

foo( 2 );

Давайте представим вышеуказанный обмен между ними (который обрабатывает этот код) как беседу. Беседа может пойти примерно так:

***Движок***: Эй, *Область видимости*, у меня есть RHS-ссылка на foo. Когда-нибудь слышала о такой?

***Область видимости***: Ну разумеется, слышала. *Компилятор* объявил ее всего секунду назад. Это функция. Пожалуйста!

***Движок***: Отлично, спасибо! Хорошо, я выполняю foo.

***Движок***: Эй, *Область видимости*, у меня есть LHS-ссылка на a, слышала что-нибудь о ней?

***Область видимости***: Ну разумеется, слышала. *Компилятор* объявил ее как формальный параметр в foo только что. Пожалуйста!

***Движок***: Отзывчива как всегда, *Область видимости*. Снова спасибо. А теперь присвоим 2 в a.

***Движок***: Эй, *Область видимости*, извини, что беспокою тебя снова. Мне нужен RHS-поиск console. Когда-нибудь слышала о таком имени?

***Область видимости***: Нет проблем, *Движок*, это то, чем я весь день и занимаюсь. Да, у меня есть console. Она встроенная. Пожалуйста!

***Движок***: Идеально. Ищу log(..). Превосходно, это функция.

***Движок***: Эй, *Область видимости*. Можешь помочь мне с RHS-ссылкой на a? Думаю, я ее помню, но просто хочу лишний раз проверить.

***Область видимости***: Ты прав, *Движок*. Та же ссылка, не изменилась. Пожалуйста!

***Движок***: Круто! Передаю значение a, которое равно 2, в log(..).

...

### Вложенная область видимости

Мы говорили, что *Область видимости* — это набор правил поиска переменных по их идентификатору. Однако, обычно бывает более одной *Области видимости*.

Также как блок или функция вкладывается внутрь другого блока или функции, области видимости вкладываются внутрь других областей. Поэтому, если переменную не найти в ближайшей области видимости, *Движок* заглядывает в следующую внешнюю по отношению к этой область видимости, продолжая так до тех пор, пока не найдет или пока не достигнет самой внешней (т.е. глобальной) области.

Пример:

function foo(a) {

console.log( a + b );

}

var b = 2;

foo( 2 ); // 4

RHS-ссылка на b не может быть разрешена внутри функции foo, но она может быть разрешена в *Области видимости*, окружающей ее (в этом случае, глобальной).

Поэтому, еще раз пересмотрев беседы между *Движком* и *Областью видимости*, мы возможно услышим:

***Движок***: "Эй, *Область видимости* foo, что-нибудь слышала о b? У меня есть RHS-ссылка на нее".

***Область видимости***: "Не-а, никогда не слышала о такой. Попробуй что-нибудь другое!"

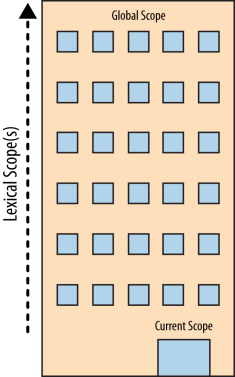
***Движок***: "Эй, *Область видимости* снаружи foo! О, ты еще и глобальная *Область видимости*, круто. Когда-нибудь слышала о b? У меня есть RHS-ссылка на нее."

***Область видимости***: "Да-да, конечно есть. Пожалуйста!"

Простые правила просмотра вложенных *Областей видимости*: *Движок* начинает в текущей выполняемой *Области видимости*, ищет в ней переменную, затем если не находит, продолжает поиск уровнем выше и так далее. Если достигнута самая внешняя глобальная область видимости, поиск останавливается, независимо от того, нашел он переменную или нет.

**Берем за основу метафоры**

Для визуализации процесса разрешения во вложенных *Областях видимости*, я хочу, чтобы вы подумали об этом высоком здании.

[](https://github.com/azat-io/you-dont-know-js-ru/blob/master/scope%20&%20closures/fig1.png)

Здание символизирует набор правил вложенных *Областей видимости* нашей программы. Первый этаж здания представляет вашу текущую выполняемую *Область видимости*, где бы вы ни были. Верхний уровень здания — это глобальная *Область видимости*.

Вы разрешаете LHS- и RHS-ссылки ища на вашем текущем этаже, а если вы не нашли что искали, поднимаетесь на лифте на следующий этаж, ища там, затем на следующий и так далее. Как только вы попадаете на верхний этаж(глобальная *Область видимости*), вы либо находите то, что искали, либо не находите. Но в любом случае вы должны остановиться.

### Ошибки

Почему имеет значение называть поиск LHS или RHS?

Потому что эти два типа поиска ведут себя по-разному в обстановке, когда переменная еще не была объявлена (не была найдена ни в одной просмотренной *Области видимости*).

Представьте:

function foo(a) {

console.log( a + b );

b = a;

}

foo( 2 );

Когда происходит RHS-поиск b первый раз, она не будет найдена. Это как бы "необъявленная" переменная, так как она не была найдена в этой области видимости.

Если RHS-поиск не сможет когда-либо найти переменную, в любой из вложенных *Областей видимости*, это приведет к возврату *Движком* ошибки ReferenceError. Важно отметить, что эта ошибка имеет тип ReferenceError.

Напротив, если *Движок* выполняет LHS-поиск и достигает верхнего этажа (глобальной *Области видимости*) и не находит ничего, и если программа не запущена в "строгом режиме", то затем глобальная *Область видимости* создаст новую переменную с таким именем **в глобальной области видимости** и передаст ее обратно *Движку*.

*"Нет, до этого не было ни одной и я любезно создала ее для тебя."*

"Строгий режим", который был добавлен в ES5, имеет ряд разных отличий от обычного/нестрогого/ленивого режима. Одно такое отличие — это то, что он запрещает автоматическое/неявное создание глобальных переменных. В этом случае, не было бы никакой переменной в глобальной *Области видимости*, чтобы передать обратно от LHS-поиска, и *Движок* выбросит ReferenceError аналогично случаю с RHS.

Теперь, если переменная найдена в ходе RHS-поиска, но вы пытаетесь сделать что-то с ее значением, что невозможно, например, пытаетесь выполнить как функцию не-функциональное значение или ссылаетесь на свойство значения null или undefined, то *Движок* выдаст другой вид ошибки, называемый TypeError.

ReferenceError — это сбой разрешения имени, связанный с *Областью видимости*, тогда как TypeError подразумевает, что разрешение имени в *Области видимости* было успешным, но была попытка выполнения нелегального/невозможного действия с результатом.

**Обзор**

Область видимости — это набор правил, которые определяют где и как переменная (идентификатор) могут быть найдены. Этот поиск может осуществляться для целей присваивания значения переменной, которая является LHS (left-hand-side) ссылкой, или может осуществляться для целей извлечения ее значения, которое является RHS (right-hand-side) ссылкой.

LHS-ссылки являются результатом операции присваивания. Присваивания, связанные с *Областью видимости*, могут происходить либо с помощью операции =, либо передачей аргументов (присваиванием) параметрам функции.

JavaScript *Движок* перед выполнением сначала компилирует код, и пока он это делает, он разбивает операторы, подобные var a = 2; на два отдельных шага:

1. Первый, var a, чтобы объявить ее в *Область видимости*. Это выполняется в самом начале, до исполнения кода.
2. Позже, a = 2 ищет переменную (LHS-ссылку) и присваивает ей значение, если находит.

Оба поиска ссылок LHS и RHS начинаются в текущей выполняющейся *Области видимости* и если нужно (т.е. они не нашли что искали в ней), они работают с их более высокими вложенными *Областями видимости*, с одной областью (этажом) за раз, ища идентификатор, пока не доберутся до глобальной (верхний этаж) и не остановятся, вне зависимости от результата поиска.

Невыполненные RHS-ссылки приводят к выбросу ReferenceError. Невыполненные LHS-ссылки приводят к автоматической, неявносозданной переменной с таким именем (если не включен "Строгий режим"), либо к ReferenceError (если включен "Строгий режим").

**Ответы к тесту**

function foo(a) {

var b = a;

return a + b;

}

var c = foo( 2 );

1. Определите все LHS-поиски (их 3!).

**c = .., a = 2 (неявное присваивание параметру) и b = ..**

1. Определите все RHS-поиски (их 4!).

**foo(2.., = a; ,  a + .. и .. + b.**

## Глава 2: Лексическая область видимости

В главе 1, мы определили "область видимости" как набор правил, которые регулируют как *Движок* может искать переменную по ее имени идентификатора и найти ее либо в текущей *Области видимости*, либо в любой из *вложенных Областей видимости*, в которой она содержится.

Есть две преобладающих модели того, как работает область видимости. Первая из них, безусловно самая распространенная, используется необъятным большинством языков программирования. Она называется **Лексическая область видимости** и мы изучим ее в деталях. Другая модель, которая все еще используется некоторыми языками (такими как скриптовый Bash, некоторые режимы в Perl и т.д.), называется **Динамическая область видимости**.

Динамическая область видимости рассматривается в приложении A. Я упоминаю ее здесь только чтобы показать контраст с лексической областью действия, которая является моделью области видимости, используемой в JavaScript.

**Время разбора на лексемы**

Как мы уже обсудили в главе 1, первая традиционная фаза работы стандартного компилятора языка называется разбиение на лексемы (lexing или tokenizing). Если вы припомните, то процесс разбиения на лексемы анализирует символы строки исходного кода и дает семантическое значение лексемам как результат некоторого парсинга с состоянием.

Это и есть та концепция, которая предоставляет основу понимания что такое лексическая область видимости и откуда происходит ее название.

Определяя ее отчасти через саму себя, лексическая область видимости — это область видимости, которая определена во время разбора на лексемы. Иными словами, лексическая область видимости основана на том, где переменные и блоки области видимости были созданы вами во время написания и таким образом (в основном) навечно зафиксированы на момент, когда лексический анализатор обрабатывал ваш код.

**Примечание:** Совсем скоро мы увидим, что есть некоторые способы обмануть лексическую область действия, тем самым меняя ее после того, как лексический анализатор уже прошелся по ней, но к ним относятся неодобрительно. Считается лучшей практикой обращаться с лексической областью видимости как, по сути дела с чисто лексической и следовательно полностью относящейся к моменту написания кода по своей природе.

Давайте рассмотрим этот код:

function foo(a) {

var b = a \* 2;

function bar(c) {

console.log( a, b, c );

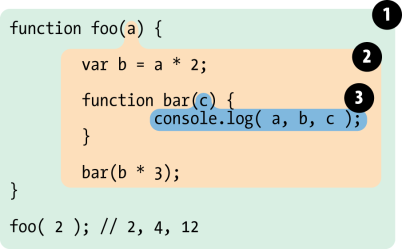
}

bar(b \* 3);

}

foo( 2 ); // 2 4 12

В этом примере кода есть три вложенных области видимости. Удобно представлять эти области видимости как зоны одна внутри другой.

[](https://github.com/azat-io/you-dont-know-js-ru/blob/master/scope%20&%20closures/fig2.png)

**Зона 1** заключает в себе глобальную область видимости, у которой есть всего один идентификатор: foo.

**Зона 2** заключает в себе область видимости foo, которая включает в себя три идентификатора: a, bar и b.

**Зона 3** заключает в себе область видимости bar и она включает в себя всего один идентификатор: c.

Зоны областей видимости определяются тем, где написаны блоки области видимости, которые последовательно вложены друг в друга. В следующей главе, мы обсудим различные единицы области видимости, а сейчас давайте просто допустим, что каждая функция создает новую зону области видимости.

Зона для bar полностью содержится в зоне для foo, потому что (и только поэтому) это то место, которое мы выбрали для создания функции bar.

Заметьте, что эти вложенные зоны вложены однозначно и четко. Мы не говорим сейчас о круговых диаграммах Эйлера—Венна, где зоны могут пересекать границы. Иными словами, ни одна зона для функции не может одновременно существовать (частично) внутри двух других зон внешних областей видимости, как и ни одна функция не может частично быть внутри каждой из двух родительских функциях.

### Поиски

Структура и относительное положение этих зон областей видимости полностью объясняет *Движку* все места, в которые ему нужно заглянуть, чтобы найти идентификатор.

В вышеприведенном коде, *Движок* выполняет оператор console.log(..) и идет искать три переменных a, b, и c, на которые есть ссылки. Сначала он начинает с самой внутренней зоны области видимости, области видимости функции bar(..). Он не найдет там a, поэтому пойдет на уровень выше, наружу к следующей ближайшей зоне области видимости, области видимости foo(..). Там он наконец найдет a, и поэтому использует эту a. То же самое и для b. А вот c он найдет внутри bar(..).

Если бы c была и внутри bar(..), и внутри foo(..), то оператор console.log(..) нашел и использовал ту, что в bar(..), никогда не трогая такую же из foo(..).

**Поиск в области видимости прекращается как только он находит первое совпадение**. Одно и то же имя идентификатора может быть указано в нескольких слоях вложенных областей видимости, что называется "затенение (shadowing)" (внутренний идентификатор "затеняет (shadows)" внешний). Независимо от затенения, поиск в области видимости всегда начинается с самой внутренней области видимости, исполняющейся в данный момент и работает таким путем по направлению наружу/вверх пока не найдется первое совпадение и тогда останавливается.

**Примечание:** Глобальные переменные также автоматически являются свойствами глобального объекта (window в браузерах и т.п.), поэтому *можно* ссылаться на глобальную переменную не прямо по ее лексическому имени, а вместо этого косвенно использовать ссылку на свойство глобального объекта.

window.a

Эта техника дает доступ к глобальной переменной, которая в противном случае была бы недоступна из-за затенения. Однако, неглобальные затененные переменные не могут быть доступны.

Не важно *откуда* вызывается функция или даже *как* она вызывается, ее лексическая область видимости определена **только** тем, где функция была объявлена.

Процесс поиска в лексической области видимости применяется *только* к идентификаторам первого класса, таким как a, b и c. Если у вас есть ссылка на foo.bar.baz в строке кода, поиск в лексической области будет применен чтобы найти идентификатор foo, но как только он находит эту переменную, ему на смену приходят правила доступа к свойствам объекта, чтобы разрешить имена свойств bar и baz, соответственно.

**Обманываем лексическую область видимости**

Если лексическая область видимости определяется только тем, где объявлена функция, что целиком во власти момента написания кода, какой может быть возможный путь "изменить" (т.е. обмануть) лексическую область во время выполнения?

В JavaScript ест два таких механизма. К ним обоим одинаково неодобрительно относятся в широком сообществе как к плохим практикам использования кода. Но типичные аргументы против них обычно не видят самого главного: **обман лексической области видимости ведет к более худшей производительности.**

Перед тем как я объясню проблему с производительностью, всё же, давайте взглянем на то, как работают эти два механизма.

### eval

Функция eval(..) в JavaScript берет строку как аргумент и интерпретирует содержимое строки как если бы это был код, написанный в этой точке программы. Другими словами, вы можете программно генерировать код внутри вашего собственного кода и запускать сгенерированный код как если бы он был там в время написания кода.

При вычислении eval(..) в таком свете, должно быть ясно как eval(..) позволяет модифицировать окружение лексической области видимости обманывая и притворяясь, что этот код был тут всё время.

На последующих строках кода после того, как выполнена eval(..), *Движок* не "узнает" или не "позаботится" о том, что предыдущий код, о котором идет речь, был динамически интерпретирован и таким образом изменил окружение лексической области видимости. *Движок* просто выполнит свои поиски по лексической области видимости как он обычно это делает.

Представьте следующий код:

function foo(str, a) {

eval( str ); // обман!

console.log( a, b );

}

var b = 2;

foo( "var b = 3;", 1 ); // 1, 3

Строка "var b = 3;" интерпретируется в точке вызова eval(..), как будто этот код был тут всегда. Поскольку этот код объявляет новую переменную b, он изменяет существующую лексическую область foo(..). Фактически, как было указано выше, этот код на самом деле создает переменную b внутри foo(..), которая затеняет b, которая была объявлена во внешней (глобальной) области видимости.

Когда происходит вызов console.log(..), он находит и a, и b в области видимости foo(..), но никогда не найдет внешнюю b. По этой причине, мы напечатаем "1, 3" вместо "1, 2" как это было бы в обычном случае.

**Примечание:** В этом примере для простоты строка "кода", которую мы передали, была фиксированным литералом. Но она легко может быть создана программно соединением символов вместе на основе логики вашей программы. eval(..) обычно используется для динамически созданного кода, поскольку динамическое вычисление по существу статического кода из строкового литерала не дает никакого реального преимущества перед простым написанием этого кода напрямую.

По умолчанию, если строка кода, которую выполняет eval(..), содержит одно или более объявлений (переменных или функций), тогда это действие меняет существующую лексическую область видимости, в которой располагается eval(..). Технически, eval(..) может быть вызвана "неявно", путем различных трюков (вне нашего обсуждения здесь), которые приводят к тому, что она вместо этого запускается в контексте глобальной области видимости, таким образом меняя ее. Но в любом случае, eval(..) может в время исполнения менять лексическую область видимости, определенную на момент написания кода.

**Примечание:** Когда eval(..) используется в программе, работающей в строгом режиме, она работает со своей собственной лексической областью видимости, что означает, что объявления, сделанные внутри eval(), не поменяют окружающую область видимости.

function foo(str) {

"use strict";

eval( str );

console.log( a ); // ReferenceError: a is not defined

}

foo( "var a = 2" );

Есть другие средства в JavaScript, которые почти равносильны по эффекту вызовам eval(..). setTimeout(..) и setInterval(..), которые *могут* принимать строку как свой первый аргумент, содержимое которой вычисляется как код динамически сгенерированной функции. Это старая, унаследованная функциональность и давным-давно устаревшая. Не делайте так!

Конструктор функции new Function(..) аналогично принимает строку кода в своем **последнем** аргументе, чтобы превратить ее в динамически сгенерированную функцию (первые аргументы, если указаны, являются именованными параметрами для новой функции). Такой синтаксис конструктора функции немного безопаснее, чем eval(..), но его также следует избегать в вашем коде.

Варианты использования динамически сгенерированного кода внутри вашей программы крайне редки, поскольку снижение производительности почти никогда не стоит такой возможности.

### with

Еще одна возможность в JavaScript, к которой неодобрительно относятся (и которая сейчас устарела!), которая обманывает лексическую область видимости, это ключевое слово with. Есть много подходящих путей, чтобы объяснить что такое with, но я выберу объяснение с точки зрения того, как оно взаимодействует и влияет на лексическую область видимости.

with обычно описывают как сокращение для выполнения множественных ссылок на свойства объекта *без* повторения каждый раз ссылки на сам объект.

Например:

var obj = {

a: 1,

b: 2,

c: 3

};

// более "скучно" повторять "obj"

obj.a = 2;

obj.b = 3;

obj.c = 4;

// "легкое" сокращение

with (obj) {

a = 3;

b = 4;

c = 5;

}

Однако, здесь происходит нечто большее, чем просто удобное сокращение для доступа к свойствам объекта. Представьте:

function foo(obj) {

with (obj) {

a = 2;

}

}

var o1 = {

a: 3

};

var o2 = {

b: 3

};

foo( o1 );

console.log( o1.a ); // 2

foo( o2 );

console.log( o2.a ); // undefined

console.log( a ); // 2 — Упс, утекшая глобальная переменная!

В этом примере кода, создаются два объекта o1 и o2. У одного есть свойство a, а у другого — нет. Функция foo(..)берет ссылку на объект obj как аргумент, а затем вызывает with (obj) { .. } с этой ссылкой. Внутри блока with мы делаем, как нам представляется, обычную лексическую ссылку на a, на самом деле LHS-ссылку (см. главу 1), чтобы присвоить ей значение 2.

Когда мы передаем o1, присвоение a = 2 находит свойство o1.a и присваивает ему значение 2, что нашло свое отражение в последующем операторе console.log(o1.a). Однако, когда мы передаем o2, поскольку у него нет свойства a, это свойство не создается, а o2.a остается undefined.

Но затем мы замечаем своеобразный побочный эффект, факт того, что присвоение a = 2 создало глобальную переменную a. Как такое может быть?

Оператор with берет объект, у которого есть ноль или более свойств, и **трактует этот объект как если бы *он* являлся целиком отдельной лексической областью видимости**, и таким образом свойства объекта воспринимаются как лексически определенные идентификаторы в этой "области видимости".

**Примечание:** Даже если блок with трактует объект как лексическую область видимости, обычное объявление var внутри этого блока with не будет входить в область этого блока with, а вместо этого будет в области видимости функции, содержащей этот блок.

В том время как функция eval(..) может менять существующую лексическую область видимости, если она принимает строку кода с одним или более объявлениями в ней, то оператор with на самом деле создает **полностью новую лексическую область действия** на ровном месте из объекта, который вы ему передаете.

Таким образом мы поняли, что "область видимости", объявленная оператором with когда мы передали o1, была o1 и что у этой "области видимости" есть "идентификатор", который соответствует свойству o1.a. Но когда мы использовали o2 как "область видимости", у нее не было такого "идентификатора" a и поэтому сработали обычные правила поиска LHS-идентификатора (см. главу 1).

Ни "область видимости" o2, ни область видимости foo(..), ни даже глобальная область видимости не нашли у себя идентификатор a, поэтому когда выполняется a = 2, это приводит к созданию автоматической глобальной переменной (поскольку мы в нестрогом режиме).

Это незнакомая и отчасти ошеломляющая мысль увидеть как with превращает, во время выполнения, объект и его свойства в "область видимости" *с* "идентификаторами". Но это — самое понятное объяснение, которое я могу дать результатам, которые мы видели.

**Примечание:** В дополнение к тому, что является плохой идеей их использовать, как eval(..), так и withподвергаются воздействию (ограничиваются) строгим режимом. with полностью запрещено, в то время как различные формы скрытых или небезопасных eval(..) запрещены при сохранении базовой функциональности.

### Быстродействие

Оба eval(..) и with обманывают в той или иной форме лексическую область видимости, определенную на этапе кодирования, изменением или созданием новой лексической области видимости во время выполнения.

Итак, что тут такого, спросите вы? Если они предлагают улучшенную функциональность и гибкость кодирования, разве это не *хорошие* возможности? **Нет.**

В *Движке* JavaScript есть много оптимизаций быстродействия, которые он выполняет во время фазы компиляции. Некоторые из этих оптимизаций сводятся к возможности по сути статически анализировать код, как только он разбирается на лексемы, и заранее определять где находятся все переменные и функции, для того чтобы понадобилось меньше усилий для разрешения имен идентификаторов во время выполнения.

Но если *Движок* находит в коде eval(..) или with, он фактически должен *предположить*, что все его знание о местонахождении идентификаторов может быть неправильным, потому что он не знает точно во время написания кода какой код вы можете передать в eval(..), чтобы поменять лексическую область видимости или какое содержимое объекта вы можете передать в with, чтобы создать новую лексическую область видимости, чтобы быть в курсе.

Иными словами, с точки зрения пессимистического здравого смысла, большинство таких оптимизаций, которые он *мог бы* сделать, бессмысленны, если есть eval(..) или with, поэтому он просто не выполняет *никаких* оптимизаций.

Ваш код почти определенно будет стремиться работать медленнее просто из-за того факта, что вы включили eval(..)или with где-либо в вашем коде. Не важно насколько умным может быть *Движок* пытаясь ограничить побочные эффекты этих пессимистических предположений, но **нельзя обойти тот факт, что без оптимизаций код работает медленнее.**

**Обзор**

Лексическая область видимости означает, что область видимости определена решениями о том, где объявляются функции на стадии написания кода. Фаза разбиения на лексемы при компиляции фактически способна узнать где и как объявлены все идентификаторы, и таким образом предсказать как их будут искать во время выполнения.

Два механизма в JavaScript могут "обмануть" лексическую область видимости: eval(..) и with. Первый может менять существующую лексическую область видимости (во время выполнения) исполняя строку "кода", в которой есть одно или несколько объявлений. Второй по сути создает целую новую лексическую область видимости (снова во время выполнения) интерпретируя ссылку на объект *как* "область видимости", а свойства этого объекта как идентификаторы этой области.

Недостаток этих механизмов в том, что они лишают смысла возможность  *Движка* выполнять оптимизации во время компиляции, принимающие во внимание поиск в области видимости, так как *Движок* должен пессимистически предположить, что такие оптимизации будут неправильными. Код *будет* выполняться медленнее в результате использования любой из этих возможностей. **Не используйте их!**

## Глава 3: Область видимости: функции против блоков

Как мы уже исследовали в главе 2, область видимости состоит из серии "зон", каждая из которых действует как контейнер или корзина, в которой объявляются идентификаторы (переменные, функции). Эти зоны четко вкладываются друг в друга и это вложение определяется во время написания кода.

Но что именно конкретно создает новую зону? Только функция? Могут ли другие структуры в JavaScript создавать зоны областей видимости?

### Область видимости из функций

Самый общий ответ на эти вопросы такой — в JavaScript есть области видимости в функциях. То есть, каждая функция, которую вы объявляете, создает зону для себя, но больше ни одна структура не создает свою собственную зону области видимости. Как мы скоро увидим, это не совсем правда.

Но сначала, давайте исследуем область видимости функции и ее применения.

Представьте такой код:

function foo(a) {

var b = 2;

// некоторый код

function bar() {

// ...

}

// еще код

var c = 3;

}

В этом коде зона области видимости для foo(..) включает в себя идентификаторы a, b, c и bar. **Не важно** *где* в области видимости появится объявление, переменная или функция принадлежит к содержащей их зоне области видимости, вне зависимости от места объявления. Мы исследуем в следующей главе как *это* в точности работает.

У bar(..) есть своя собственная зона области видимости. Также и у глобальной области видимости, у которой есть всего один идентификатор: foo.

Так как a, b, c и bar все принадлежат к зоне области видимости foo(..), они недоступны вне foo(..). То есть, нижеприведенный код весь целиком приведет к ошибкам ReferenceError, так как идентификаторы недоступны в глобальной области видимости:

bar(); // ошибка

console.log( a, b, c ); // все 3 вызовут ошибку

Однако, все эти идентификаторы (a, b, c, foo и bar) доступны *внутри* foo(..) и конечно также доступны внутри bar(..) (предполагаем, что нет ни одного объявления затеняющих идентификаторов внутри bar(..)).

Область видимости функции поощряет идею, что все переменные принадлежат функции и могут быть использованы и переиспользованы на всем протяжении функции (и более того, доступны даже во вложенных областях видимости). Этот подход к дизайну может быть довольно полезен и несомненно может использовать в полном объеме "динамическую" природу переменных JavaScript, чтобы иметь дело со значениями разных типов при необходимости.

С другой стороны, если вы не принимаете меры предосторожности, переменные, существующие на всей протяженности области видимости, могут привести к неожиданным ошибкам.

**Прячемся на виду у всей области видимости**

Традиционный путь рассуждений о функциях таков: вы определяете функцию, а затем добавляете код внутрь нее. Но обратное рассуждение столь же обоснованно и удобно: взять любую произвольную часть кода, которую вы написали, и обернуть ее в объявление функции, что в сущности "прячет" код.

Практическим результатом является создание зоны области видимости вокруг кода, о котором идет речь, а это значит, что любые объявления (переменной или функции) в этом коде будут ограничены областью видимости новой оборачивающей код функции, вместо ранее охватывающей области видимости. Иными словами, вы можете "спрятать" переменные и функции заключив их в область видимости функции.

Почему же "прятать" переменные и функции — полезная техника?

Есть множество причин, мотивирующих на это сокрытие, основанное на области видимости. Они имеют тенденцию проистекать из принципа дизайна ПО "Принцип наименьших привилегий", также иногда называемый "Наименьшие полномочия" или "Наименьшая открытость". Этот принцип заявляет, что в дизайне ПО, таком как API для модуля/объекта, вам следует выставлять наружу только то, что минимально необходимо и "прятать" всё остальное.

Этот принцип распространяется и на выбор того, какая область видимости будет содержать переменные и функции. Если все переменные и функции были бы в глобальной области видимости, они были бы, конечно, доступны любой вложенной области видимости. Но это нарушает принцип "Наименьшей..." в том, что вы (видимо) открываете многие переменные или функции, которые вам следовало в противном случае держать приватными, коль скоро правильное использование кода будет препятствовать доступу к этим переменным/функциям.

Например:

function doSomething(a) {

b = a + doSomethingElse( a \* 2 );

console.log( b \* 3 );

}

function doSomethingElse(a) {

return a - 1;

}

var b;

doSomething( 2 ); // 15

В этом коде, переменная b и функция doSomethingElse(..) — похоже "частные" детали того, как doSomething(..)выполняет свою работу. Предоставление окружающей области видимости "доступа" к b и doSomethingElse(..) не только не необходимо, но также возможно и "опасно" тем, что их могут нечаянно использовать, намеренно или нет, и это может нарушить предварительные соглашения в doSomething(..).

Более "правильный" дизайн скрыл бы эти частные детали внутри области видимости doSomething(..), например как тут:

function doSomething(a) {

function doSomethingElse(a) {

return a - 1;

}

var b;

b = a + doSomethingElse( a \* 2 );

console.log( (b \* 3) );

}

doSomething( 2 ); // 15

Теперь, b и doSomethingElse(..) недоступны для любого внешнего воздействия, а взамен управляются только в doSomething(..). Функциональность и конечный результат оказались не затронуты, а дизайн хранит частные детали частными, что обычно считается лучшим ПО.

### Предотвращение коллизий

Еще одно преимущество "скрытия" переменных и функций внутри области действия — чтобы избежать неумышленных коллизий между двумя идентификаторами с одним и тем же именем, но с разным целевым использованием. Коллизии часто приводят к неумышленной перезаписи значений.

Например:

function foo() {

function bar(a) {

i = 3; // меняем `i` в окружающей области видимости цикла for-loop

console.log( a + i );

}

for (var i=0; i<10; i++) {

bar( i \* 2 ); // упс, впереди бесконечный цикл!

}

}

foo();

Присваивание i = 3 внутри bar(..) неожиданно перезаписывает i, которая была объявлена в foo(..) внутри цикла for-loop. В этом случае, это приведет к бесконечному циклу, так как i установлена в фиксированное значение 3 и она всегда будет оставаться < 10.

Присваиванию внутри bar(..) нужно объявить локальную переменную для своих нужд, независимо от того, какое имя идентификатора выбрано. var i = 3; исправило бы эту проблему (и создало бы ранее упоминавшееся объявление "затененной переменной" i). *Дополнительная*, не альтернативная возможность — выбрать совсем другое имя идентификатора, такое как var j = 3;. Но дизайн вашего ПО естественно может подразумевать использование одного и того же имени идентификатора, поэтому использование области видимости, чтобы "скрыть" ваше внутреннее объявление — это ваша лучшая/единственная возможность в этом случае.

### Глобальные "Пространства имен"

Особенно яркий пример (наверное) коллизии переменных возникает в глобальной области видимости. Многие библиотеки, загружаемые в вашу программу, могут без всякого труда вступить в коллизию друг с другом, если они не спрячут должным образом свои внутренние/приватные функции и переменные.

Такие библиотеки, как правило, создают единственное объявление переменной, часто объекта, с достаточно уникальным именем в глобальной области видимости. Этот объект затем используется как "пространство имен" для этой библиотеки, где вся открываемая характерная функциональность делается как свойства этого объекта (пространства имен), вместо того, чтобы выставлять свои идентификаторы в области видимости на самом верхнем уровне.

Например:

var MyReallyCoolLibrary = {

awesome: "stuff",

doSomething: function() {

// ...

},

doAnotherThing: function() {

// ...

}

};

### Функции как области видимости

Мы уже видели, что можно взять любой кусок кода и обернуть его в функцию, и это эффективно "скроет" любые вложенные определения переменных или функций от внешней области видимости во внутренней области видимости этой функции.

Например:

var a = 2;

function foo() { // <-- вставляем это

var a = 3;

console.log( a ); // 3

} // <-- и это

foo(); // <-- и это

console.log( a ); // 2

Несмотря на то, что эта техника "работает", она не обязательно очень идеальна. Она привносит некоторые проблемы. Первая, это то, что мы должны объявить именованную функцию foo(), которая означает, что само имя идентификатора foo "засоряет" окружающую область видимости (в данном случае глобальную). Мы также должны явно вызвать функцию по имени (foo()) для того ,чтобы обернутый код выполнился на самом деле.

Было бы идеально, если бы функции не было нужно имя (или точнее, чтобы это имя не загрязняло окружающую область видимости) и если бы функция могла бы автоматически выполняться.

К счастью, JavaScript предлагает решение обеих проблем.

var a = 2;

(function foo(){ // <-- вставляем это

var a = 3;

console.log( a ); // 3

})(); // <-- и это

console.log( a ); // 2

Давайте разберем в деталях что же тут происходит.

Сперва, заметьте, что оператор окружающей функции начинается с (function... в противоположность простому function.... Хоть это и может показаться несущественной деталью, это на самом деле огромное изменение. Вместо того, чтобы трактовать функцию как стандартное объявление, функция трактуется как функциональное выражение.

**Примечание:** Самый легкий путь отличить объявление от выражения — позиция слова "function" в операторе (не только строка, но и отдельный оператор). Если "function" — самое первое, что стоит в операторе, то это объявление функции. Иначе, это функциональное выражение.

Ключевое отличие между объявлением функции и функциональным выражением, которое мы можем заметить тут, связано с тем, где его имя связывается с идентификатором.

Сравните два предыдущих кода. В первом, имя foo связано с окружающей областью видимости и мы вызываем его напрямую через foo(). Во втором коде, имя foo не связано с окружающей областью видимости, а взамен связано только со своей собственной функцией.

Иными словами, (function foo(){ .. }) как выражение означает, что идентификатор foo может быть найден *только*в области видимости, которая обозначена .., но не во внешней области видимости. Скрывание имени foo внутри себя означает, что оно не будет неоправданно загрязнять окружающую область видимости.

### Анонимный против названного

Возможно, вы лучше всех знакомы с функциональными выражениями как параметрами функций обратного вызова, например:

setTimeout( function(){

console.log("I waited 1 second!");

}, 1000 );

Это называется "анонимное функциональное выражение", так как у function()... нет именованного идентификатора. Функциональные выражения могут быть анонимными, но объявления функций не могут опускать имя — это было бы невалидным синтаксисом JS.

Анонимные функциональные выражения быстро и легко вводить и многие библиотеки и утилиты проявляют тенденцию к поощрению этого идиоматического стиля кода. Однако, у них есть несколько недостатков о которых нужно упомянуть:

1. У анонимных функций нет удобного имени для отображения в стектрейсах (stacktrace), что может затруднить отладку.
2. Если функции без имени будет нужно сослаться на себя же, для рекурсии или чего-то подобного, к сожалению требуется **устаревшая** ссылка arguments.callee. Еще один пример необходимости в ссылке на себя, когда функция обработчика события хочет отписать себя от события после выполнения.
3. Анонимные функции опускают имя, что часто удобно для обеспечения большей читаемости/понятности кода. Наглядное же имя помогает самодокументировать рассматриваемый код.

**Встраиваемые функциональные выражения** — мощные и полезные инструменты, выбор между анонимными и именованными не умаляет их достоинств. Именование вашего функционального выражения достаточно эффективно решает все перечисленные недостатки, и при этом не имеет ощутимых минусов. Лучшая практика — это всегда именовать ваши функциональные выражения:

setTimeout( function timeoutHandler(){ // <-- Смотри, у меня есть имя!

console.log( "I waited 1 second!" );

}, 1000 );

**Вызов функциональных выражений по месту**

var a = 2;

(function foo(){

var a = 3;

console.log( a ); // 3

})();

console.log( a ); // 2

Теперь, когда у нас есть функция как выражение благодаря обертыванию ее в пару ( ), мы можем вызвать эту функцию добавив еще одни () в конце, как тут (function foo(){ .. })(). Первая окружающая пара ( ) делает функцию выражением, а вторая () выполняет функцию.

Этот шаблон настолько в ходу, что несколько лет назад сообщество условилось о термине для него: IIFE, что означает **I**mmediately (немедленно) **I**nvoked (вызываемое) **F**unction (функциональное) **E**xpression (выражение).

Конечно, IIFE не обязательно нужны имена, самая распространенная форма IIFE — использование анонимного функционального выражения. Несмотря на то, что определенно менее используемое, именование IIFE имеет все вышеперечисленные преимущества над анонимными функциональными выражениями, поэтому взять его на вооружение — хорошая практика.

var a = 2;

(function IIFE(){

var a = 3;

console.log( a ); // 3

})();

console.log( a ); // 2

Есть легкая вариация формы традиционной IIFE, которую предпочитают некоторые: (function(){ .. }()). Посмотрим ближе, чтобы увидеть разницу. В первой форме функциональное выражение обернуто в ( ), а затем пара вызывающих ее () снаружи прямо за ними. Во второй форме, вызывающая пара () переместилась внутрь внешней окружающей пары ( ).

Эти две формы идентичны по функциональности. **Какую из них предпочесть — всего лишь ваш стилистический выбор**.

Еще одна вариация IIFE, которая широко распространена, это использование того факта, что они, на самом деле, просто вызовы функций, и передача им аргумента(ов).

Например:

var a = 2;

(function IIFE( global ){

var a = 3;

console.log( a ); // 3

console.log( global.a ); // 2

})( window );

console.log( a ); // 2

Мы передаем ссылку на объект window, но параметр мы назвали global, так что у нас есть ясное стилистическое разграничение для глобальных и неглобальных ссылок. Конечно, вы можете передать внутрь что-то из окружающей области видимости, если хотите, и можете назвать параметр(ы) так, как удобно вам. Это в основном всего лишь стилистический выбор.

Еще одно применение этого шаблона решает (узкоспециализированную) проблему, что значение идентификатора по умолчанию undefined может быть некорректно перезаписано, приведя к неожиданным результатам. Давая параметру имя undefined, но не передавая никакого значения для этого аргумента, мы можем гарантировать, что идентификатор undefined фактически имеет незаданное значение в этом блоке кода:

undefined = true; // устанавливаем мину для другого кода! остерегайтесь!

(function IIFE( undefined ){

var a;

if (a === undefined) {

console.log( "Undefined is safe here!" );

}

})();

Еще одна вариация IIFE меняет порядок вещей на обратный, где вызываемая функция идет второй, *после* вызова и параметров, которые в нее передаются. Этот шаблон используется в проекте UMD (Universal Module Definition). Некоторые люди находят его более ясным для понимания, хотя он немного более многословный.

var a = 2;

(function IIFE( def ){

def( window );

})(function def( global ){

var a = 3;

console.log( a ); // 3

console.log( global.a ); // 2

});

Функциональное выражение def определяется во второй половине кода, а затем передается как параметр (также названный def) в функцию IIFE, определенную в первой половине кода. Наконец, параметр def (функция) вызывается, передавая window в нее как параметр global.

### Блоки как области видимости

В то время как функции являются самыми распространенными единицами области видимости и определенно самые распространенные подходы к разработке в большей части JS, есть и другие единицы области видимости, и использование этих единиц области видимости может привести к даже более лучшему, более чистому управлению кодом.

Многие языки, отличные от JavaScript, поддерживают блочную область видимости и поэтому разработчики из этих языков привыкли к такому мышлению, в свою очередь те, кто изначально работали в JavaScript могут найти эту концепцию немного чуждой.

Но даже если вы никогда не писали ни строчки кода в стиле блочной области видимости, вы возможно все-таки знакомы с этой чрезвычайно общей идиомой в JavaScript:

for (var i=0; i<10; i++) {

console.log( i );

}

Мы объявляем переменную i прямо внутри заголовка цикла for-loop, скорее всего потому, что наше *намерение* — использовать i только в контексте этого цикла for-loop и в основном игнорировать факт того, что переменная на самом деле заключает себя в окружающую область видимости (функции или глобальную).

Вот это и есть самое важное в блочной области видимости. Объявление переменных как можно ближе и как можно локальней к тому месту, где они будут использоваться. Еще один пример:

var foo = true;

if (foo) {

var bar = foo \* 2;

bar = something( bar );

console.log( bar );

}

Мы используем переменную bar только в контексте оператора if, поэтому вполне разумно, что мы объявили ее внутри блока if. Однако, место, где мы объявляем переменные не имеет значения при использовании var, так как они всегда принадлежат окружающей области видимости. Этот код по существу "поддельная" блочная область видимости, только для целей стилистики, и полагается на самообязательство не использовать случайно bar в другом месте этой области видимости.

Блочная область видимости — это инструмент для расширения ранее упоминаемого "Принципа наименьшей  открытости" с сокрытия информации в функциях на сокрытие информации в блоках вашего кода.

Давайте еще раз посмотрим пример с for-loop:

for (var i=0; i<10; i++) {

console.log( i );

}

Зачем загрязнять всю область видимости функции переменной i, которая будет использоваться (или только *следует*, по меньшей мере) в цикле for-loop?

Но более важно, что разработчики могут предпочесть *проверить* себя на предмет случайного (пере)использования переменных снаружи от их предполагаемой области действия, как например получить ошибку о неизвестной переменной, если вы пытаетесь использовать ее не в том месте. Блочная область видимости (если бы она была возможна) для переменной i сделала бы i доступной только для цикла for-loop, приводя к ошибке, если к i осуществляется доступ в другом месте функции. Это помогает убедиться, что переменные не будут переиспользованы в нечетких или труднообслуживаемых сценариях.

Но, печальная реальность состоит в том, что, на первый взгляд, в JavaScript нет возможности блочной области видимости.

Или, если точнее, пока вы не копнете немного глубже.

### with

Мы изучали with в главе 2. Несмотря на то, что к нему относятся с неодобрением с момента его появления, оно *является* примером (формой) блочной области видимости, поскольку область видимости, которая создается из объекта, существует только в течение жизненного цикла этого оператора with, а не окружающей области видимости.

### try/catch

*Очень* малоизвестным фактом является то, что JavaScript в ES3 специфицирует объявление переменной в блоке catch оператора try/catch как принадлежащее блочной области видимости блока catch.

Например:

try {

undefined(); // нелегальная операция, чтобы вызвать исключение!

}

catch (err) {

console.log( err ); // работает!

}

console.log( err ); // ReferenceError: `err` not found

Как видите, err существует только в блоке catch и выбрасывает ошибку когда вы пытаетесь сослаться на нее где-либо в другом месте.

**Примечание:** Несмотря на то, что такое поведение было определено и истинно практически во всех стандартных средах JS (за исключением разве что старого IE), многие линтеры (средства контроля качества кода) похоже до сих пор жалуются, если у вас есть два или более блоков catch в одной и той же области видимости, каждый из которых объявляет свою переменную ошибки с одинаковым именем идентификатора. В действительности это не переопределение, поскольку переменные безопасно обернуты в блочные области видимости, но линтеры похоже до сих пор раздражающе жалуются на этот факт.

Чтобы избежать таких ненужных предупреждений, некоторые разработчики именуют свои переменные в catch как err1, err2 и т.д.. Другие разработчики просто выключают проверку линтера на повторяющиеся имена переменных.

Природа блочной области видимости catch может казаться бесполезным академическим фактом, но лучше загляните в приложение B, чтобы получить детальную информацию о том насколько полезна она может быть.

### let

До сих пор мы видели, что в JavaScript есть только несколько странных нишевых возможностей, которые предлагают функциональность блочной области видимости. Если бы это было всё, что у нас есть, а так и *было* на протяжении многих, многих лет, то блочная область видимости не была бы столь полезна для разработчика на JavaScript.

К счастью, ES6 поменял ситуацию и представляет новое ключевое слово let, которое соседствует с var как еще один путь объявления переменных.

Ключевое слово let присоединяет объявление переменной к области видимости того блока (обычно пара { .. }), в котором оно содержится. Иными словами, let неявно похищает у любой блочной области видимости ее объявления переменных.

var foo = true;

if (foo) {

let bar = foo \* 2;

bar = something( bar );

console.log( bar );

}

console.log( bar ); // ReferenceError

Использование let для присоединения переменной к существующему блоку в некоторой степени неявно. Оно может ввести в заблуждение, если вы не оказываете пристальное внимание на то, в каких блоках есть переменные, принадлежащие к их области видимости, и на привычку перемещать блоки, оборачивать их в другие блоки, и т.п., по мере того, как вы разрабатываете и развиваете код.

Создание явных блоков для блочной области видимости может решить некоторые из этих вопросов, делая более очевидным то, куда присоединены переменные, а куда — нет. Обычно, явный код предпочтительней неявного или едва заметного. Такой явный стиль блочной области видимости легко получить и он более естественно походит на то, как в других языках работает блочная область видимости:

var foo = true;

if (foo) {

{ // <-- явный блок

let bar = foo \* 2;

bar = something( bar );

console.log( bar );

}

}

console.log( bar ); // ReferenceError

Можно создать обычный блок для использования let просто включая пару { .. } в любом месте, где этот оператор является валидным синтаксисом. В этом случае, мы сделали явный блок *внутри* оператора if, который потом будет легче перемещать как целый блок при рефакторинге, без изменения позиции и семантики окружающего оператора if.

**Примечание:** Еще один путь объявить явные блочные области видимости есть в приложении B.

В главе 4 мы рассмотрим поднятие переменных (hoisting), которая расскажет об объявлениях, которые воспринимаются как существующие для всей области видимости, в которой они появляются.

Однако, объявления, сделанные с помощью let, *не* поднимаются во всей области видимости блока, в котором они появляются. Такие объявления очевидно не будут "существовать" в блоке до оператора объявления.

{

console.log( bar ); // ReferenceError!

let bar = 2;

}

### Сборка мусора

Еще одна причина полезности блочной области видимости связана с замыканиями и сборкой мусора, чтобы освободить память. Мы кратко проиллюстрируем это здесь, но детально механизм замыканий будет рассматриваться в главе 5.

Пример:

function process(data) {

// делаем что-то интересное

}

var someReallyBigData = { .. };

process( someReallyBigData );

var btn = document.getElementById( "my\_button" );

btn.addEventListener( "click", function click(evt){

console.log("button clicked");

}, /\*capturingPhase=\*/false );

Обратный вызов обработчика щелчка click совсем не *требует* переменную someReallyBigData. Это значит, теоретически, что после выполнения process(..), большая памятезатратная структура данных может быть собрана сборщиком мусора. Однако, весьма вероятно (хотя зависит от реализации), что движок JS все еще должен будет оставить структуру в памяти, поскольку у функции click есть замыкание, действующее во всей области видимости.

Блочная область видимости может устранить этот недостаток, делая более явным для движка то, что ему не нужна someReallyBigData:

function process(data) {

// делаем что-то интересное

}

// всё, что объявлено внутри этого блока, может исчезнуть после него!

{

let someReallyBigData = { .. };

process( someReallyBigData );

}

var btn = document.getElementById( "my\_button" );

btn.addEventListener( "click", function click(evt){

console.log("button clicked");

}, /\*capturingPhase=\*/false );

Объявление явных блоков для переменных, чтобы локально привязать их к блокам — мощный инструмент, который вы можете добавить в свой арсенал.

### let в циклах

Особый случай, в котором let показывает себя с лучшей стороны — в случае с циклом for, как мы уже обсуждали ранее.

for (let i=0; i<10; i++) {

console.log( i );

}

console.log( i ); // ReferenceError

let в заголовке цикла for не только привязывает i к телу цикла, но фактически, он **перепривязывает ее** в каждой *итерации* цикла, обязательно переприсваивая ей значение с окончания предыдущей итерации.

Вот еще один способ показать режим пред-итеративной привязки:

{

let j;

for (j=0; j<10; j++) {

let i = j; // перепривязка в каждой итерации!

console.log( i );

}

}

Причина, по которой эта пред-итеративная привязка интересна, станет ясна в главе 5, когда мы обсудим замыкания.

Так как объявления let присоединяются к обычным блокам вместо присоединения к окружающей области видимости функции (или глобальной), могут быть ошибки в программе, когда у существующего кода есть скрытая надежда на объявления var, действующие в рамках области видимости функции, и замена var на let может потребовать дополнительного внимания при рефакторинге кода.

Пример:

var foo = true, baz = 10;

if (foo) {

var bar = 3;

if (baz > bar) {

console.log( baz );

}

// ...

}

Этот код довольно легко отрефакторить в такой:

var foo = true, baz = 10;

if (foo) {

var bar = 3;

// ...

}

if (baz > bar) {

console.log( baz );

}

Но, остерегайтесь таких изменений, когда используете переменные блочной области видимости:

var foo = true, baz = 10;

if (foo) {

let bar = 3;

if (baz > bar) { // <-- не забудьте про `bar`, если будете перемещать этот блок!

console.log( baz );

}

}

См. приложение B, чтобы узнать об альтернативном (более явном) стиле организации блочной области видимости, который может дать более простое обслуживание/рефакторинг кода, что более разумно для этих сценариев.

### const

В дополнение к let, ES6 представляет ключевое слово const, которое также создает переменную блочной области видимости, но чье значение фиксированно (константа). Любая попытка изменить это значение позже приведет к ошибке.

var foo = true;

if (foo) {

var a = 2;

const b = 3; // в блочной области видимости содержащего ее `if`

a = 3; // просто отлично!

b = 4; // ошибка!

}

console.log( a ); // 3

console.log( b ); // ReferenceError!

**Обзор**

Функции — самые распространенные единицы области видимости в JavaScript. Переменные и функции, которые объявляются внутри другой функции, по существу "скрыты" от любой из окружающих "областей видимости", что является намеренным принципом разработки хорошего ПО.

Но функции — отнюдь не только единицы области видимости. Блочная область видимости ссылается на идею, что переменные и функции могут принадлежать произвольному блоку (обычно, любой паре { .. }) кода, нежели только окружающей функции.

Начиная с ES3, в структуре try/catch есть блочная область видимости в выражении catch.

В ES6 представлено ключевое слово let (родственница ключевого слова var), чтобы позволить объявления переменных в любом произвольном блоке кода. if (..) { let a = 2; } объявит переменную a, которая фактически похитит область видимости блока { .. } в if и присоединит себя к ней.

Хоть некоторые похоже и верят в это, но блочную область видимости не следует использовать как полную замену функциональной области видимости var. Обе функциональности сосуществуют вместе, а разработчики могут и должны использовать обе техники области видимости: функциональную и блочную, в соответствующих местах, чтобы создавать лучший, более читаемый/обслуживаемый код.

## Глава 4: Поднятие переменных (Hoisting)

Теперь вы должно быть вполне уверенно разбираетесь в идее области видимости, и в том, как переменные присоединяются к различным уровням области видимости в зависимости от того, где и как они объявлены. Обе области видимости, как функции, так и блока работают по одинаковым правилам в таком виде: любая переменная, объявленная в области видимости, присоединяется к этой области видимости.

Но есть маленький нюанс в том, как работает присоединение к области видимости с объявлениями, которые появляются в различных местах области видимости, и это именно тот нюанс, который мы тут и исследуем.

**Курица или яйцо?**

Есть искушение подумать, что весь код, который вы видите в программе на JavaScript, интерпретируется строка за строкой. Несмотря на то, что по сути это правда, есть одна часть этого предположения, которая может привести к некорректному пониманию вашей программы.

Представьте такой код:

a = 2;

var a;

console.log( a ); //2

Что вы ожидаете увидеть в выводе оператора console.log(..)?

Многие разработчики ожидают увидеть undefined, поскольку оператор var a идет после a = 2, и было бы естественным предположить, что переменная переопределена и потому ей присвоено значение по умолчанию undefined. Однако, результат будет 2.

Представьте еще один код:

console.log( a );

var a = 2; // undefined

Вы можете склониться к предположению, что поскольку в предыдущем показанном коде есть некоторое поведение в стиле "немного с ног на голову", то возможно в этом коде также будет выведено 2. Другие могут подумать, что поскольку переменная a используется раньше, чем объявлена, то это приведет к выбросу ReferenceError.

К сожалению, оба предположения неверны. Будет выведено undefined.

**Так что же здесь происходит?** Похоже тут вопрос сродни "что раньше: курица или яйцо?". Что идет первым: объявление ("яйцо") или присваивание ("курица")?

**Компилятор снова наносит удар**

Чтобы ответить на этот вопрос, нам нужно вернуться в главу 1 и нашу дискуссию о компиляторах. Вспомните, что *Движок* на самом деле скомпилирует ваш код JavaScript до того, как начнет интерпретировать его. Частью фазы компиляции является нахождение и ассоциация всех объявлений с их соответствующими областями видимости. Глава 2 показала нам, что это и есть сердце лексической области видимости.

Поэтому, лучший путь думать об этих вещах — что все объявления как переменных, так и функций, обрабатываются в первую очередь, до того как будет выполнена любая часть вашего кода.

Когда видите var a = 2;, вы наверное думаете о нем как об одном операторе. Но JavaScript на самом деле думает о нем как о двух операторах: var a; и a = 2;. Первый оператор, объявление, обрабатывается во время фазы компиляции. Второй оператор, присваивание, остается **на своем месте** в фазе исполнения.

Следовательно о нашем первом коде следует думать как об обрабатываемом следующим образом:

var a;

a = 2;

console.log( a );

...где первая часть — компиляция, а вторая — выполнение.

Аналогично, наш второй код в действительности будет обработан так:

var a;

console.log( a );

a = 2;

Получается, один из путей представить это, в какой-то степени образно, что эти объявления переменной и функции "переехали" с того места, где они появились в коде в начало кода. Это дало начало названию "Поднятие (Hoisting)".

Другими словами, **яйцо (объявление) появилось до курицы (присваивания)**.

**Примечание:** Поднимаются только сами объявления, тогда как любые присваивания или другая логика выполнения остается *на месте*. Если поднятие намеренно используется, чтобы перестроить логику выполнения вашего кода, то это может вызвать хаос.

foo();

function foo() {

console.log( a ); // undefined

var a = 2;

}

Объявление функции foo (которое в этом случае *включает в себя* соответствующее значение как актуальную функцию) поднимается, благодаря чему ее вызов в первой строке может быть выполнен.

Также важно отметить, что каждое поднятие соотносится **с областью видимости**. Поэтому несмотря на то, что наши предыдущие примеры кода были упрощенными в том, что были включены только в глобальную область видимости, функция foo(..), которую мы сейчас изучаем, показывает, что var a поднимается наверх foo(..) (а не, очевидно, наверх всей программы). Так что программу можно было бы прочесть более точно как:

function foo() {

var a;

console.log( a ); // undefined

a = 2;

}

foo();

Объявления функций поднимаются, как мы только что видели. А функциональные выражения — нет.

foo(); // не ReferenceError, но TypeError!

var foo = function bar() {

// ...

};

Идентификатор переменной foo поднимается и присоединяется к включающей его области видимости (глобальной) этой программы, поэтому foo() не вызовет ошибки ReferenceError. Но в foo пока еще нет значения (как если бы это было объявление обычной функции вместо выражения). Поэтому, foo() пытается вызвать значение undefined, которое является неправильной операцией с вызовом ошибки TypeError.

Также помните, что даже если это именованное функциональное выражение, идентификатор имени недоступен в окружающей области видимости:

foo(); // TypeError

bar(); // ReferenceError

var foo = function bar() {

// ...

};

Этот код более точно интерпретируется (с учетом поднятия) как:

var foo;

foo(); // TypeError

bar(); // ReferenceError

foo = function() {

var bar = ...self...

// ...

}

**Сначала функции**

Как объявления функций, так и переменных поднимаются. Но тонкость (которая *поможет* объяснить множественные объявления "дубликатов" в коде) в том, что сперва поднимаются функции, а затем уже переменные.

Представим:

foo(); // 1

var foo;

function foo() {

console.log( 1 );

}

foo = function() {

console.log( 2 );

};

1 выводится вместо 2! Этот код интерпретируется *Движком* так:

function foo() {

console.log( 1 );

}

foo(); // 1

foo = function() {

console.log( 2 );

};

Обратите внимание, что var foo является дублем объявления (и поэтому игнорируется), даже несмотря на то, что идет до объявления function foo()..., потому что объявления функций поднимаются до обычных переменных.

В то время как множественные/дублирующие объявления var фактически игнорируются, последовательные объявления функции *перекрывают* предыдущие.

foo(); // 3

function foo() {

console.log( 1 );

}

var foo = function() {

console.log( 2 );

};

function foo() {

console.log( 3 );

}

Несмотря на то, что всё это может прозвучать как не более чем любопытный академический факт, это привлекает тем, что дубли определений в одной и той же области видимости — в самом деле плохая идея и часто приводит к странным результатам.

Объявления функций, которые появляются внутри обычных блоков, обычно поднимаются в окружающей области видимости, вместо того чтобы быть условными как показывает код ниже:

foo(); // "b"

var a = true;

if (a) {

function foo() { console.log( "a" ); }

}

else {

function foo() { console.log( "b" ); }

}

Однако, важно отметить, что такое поведение небезопасно и может измениться в будущих версиях JavaScript, поэтому лучше избегать объявления функций в блоках.

**Обзор**

У нас есть соблазн смотреть на var a = 2; как на один оператор, но *Движок* JavaScript видит это по-другому. Он видит var a и a = 2 как два отдельных оператора, первый — как задачу фазы компиляции, а второй — как задачу фазы выполнения.

Это приводит к тому, что все объявления в области видимости, независимо от того где они появляются, обрабатываются *первыми*, до того, как сам код будет выполнен. Можно мысленно представить себе это как объявления (переменных и функций), "переезжающие" в начало их соответствующих областей видимости, что мы называем "поднятие (hoisting)".

Сами объявления поднимаются, а присваивания, даже присваивания функциональных выражений, *не* поднимаются.

Остерегайтесь дублей объявлений, особенно смешанных обычных объявлений var и объявлений функций — вас будут поджидать неприятности!

## Глава 5: Замыкание области видимости

Мы добрались до этого места, как я надеюсь, с очень здравым, цельным пониманием того, как работает область видимости.

Мы сосредоточимся на крайне важной, но постоянно ускользающей, *почти мифологичной* части языка: **замыканиях**. Если вы следили за нашей дискуссией о лексической области видимости до этого момента, ее развязка заключается в том, что замыкание будет чрезвычайно, разочаровывающе, практически самоочевидным. *За ширмой фокусника есть человек и мы вот-вот его увидим*. Нет, его зовут не Крокфорд (Crockford) (прим. переводчика: постоянный участник развития языка JavaScript, создатель текстового формата обмена данными JSON)!

Если у вас всё еще есть изводящие вас вопросы о лексической области видимости, то тогда самое время вернуться и просмотреть еще раз главу 2 перед тем как продолжить читать далее.

**Прозрение**

Для тех, кто уже мало-мальски получил опыт в JavaScript, но кто, пожалуй, никогда не ухватывал суть замыканий, *понимание замыкания* может выглядеть как особая нирвана и чтобы достичь ее он должен бороться и идти на жертвы.

Я помню, что несколько лет назад у меня были навыки владения JavaScript, но не было представления о том, что такое замыкания. Подсказка о том, что была *та самая другая сторона* языка, та, которая обещала даже больше возможностей чем те, которыми я обладал, дразнила и манила меня. Я помню как читал исходный код ранних фреймворков пытаясь понять как именно они работают. Я помню тот первый раз, когда что-то подобное "шаблону модуля" начало возникать в моих мыслях. Я очень живо помню эти "*ага!*".

Чего я не знал в те времена и что потребовало от меня годы, чтобы понять, и что я надеюсь передать вам теперь — это вот этот секрет: **замыкание повсюду окружает вас в JavaScript, вы всего лишь должны распознать его и воспользоваться им.** Замыкания — это не особый инструмент, для которого вы должны изучить новый синтаксис и шаблоны. Нет, замыкания — это даже не оружие, которые вы должны изучить, чтобы уметь обращаться и подчинять его себе как Люк обучался Силе.

Замыкания получаются в результате написания кода, который исходит из лексической области видимости. Они просто возникают. Вам и в самом деле не нужно намеренно создавать замыкания, чтобы получить их преимущества. Замыкания создаются и используются вами на протяжении всего вашего кода. Что вы *упускаете*, так это то, что должен быть подходящий мысленный контекст, чтобы распознавать, пользоваться и получать выигрыш от замыканий по вашему собственному желанию.

Момент просветления будет выглядеть примерно так: **О, замыкания уже встречаются везде в моем коде, теперь я их наконец-то *вижу*.** Понимание замыкания похоже на то, когда Нео впервые увидел Матрицу.

**Суть дела**

Хорошо, достаточно гипербол и бесстыдных ссылок на кино.

Вот откровенное определение того, что вам нужно знать, чтобы понимать и распознавать замыкания:

Замыкание — это когда функция умеет запоминать и имеет доступ к лексической области видимости даже тогда, когда эта функция выполняется вне своей лексической области видимости.

Давайте рассмотрим небольшой код, чтобы проиллюстрировать это определение.

function foo() {

var a = 2;

function bar() {

console.log( a ); // 2

}

bar();

}

foo();

Этот код выглядит знакомым, напоминая о наших обсуждениях вложенных областей видимости. У функции bar() есть *доступ* к переменной a во внешней окружающей области видимости на основании правил поиска лексической области видимости (в этом случае это поиск ссылок RHS).

Это "замыкание"?

Ну, технически... *возможно*. Но согласно нашему же определению "что вам нужно знать о замыканиях" выше... *не совсем так*. Думаю, что самым точным образом объяснить ссылку в bar() на a будет с помощью правил поиска в лексической области видимости и эти правила — *единственная* (важная!) **часть** того, что такое замыкание.

С чисто академической точки зрения, про вышеуказанный код можно сказать, что у функции bar() есть *замыкание* на область foo() (и разумеется, даже на остальные области видимости, к которым у нее есть доступ, такие как глобальная область видимости в нашем случае). Немного другими словами, говорят, что bar() перекрывает область видимости foo(). Почему? Потому что bar() вложен внутри foo(). Ясно и просто.

Но, замыкание, определенное подобным путем, не является ни совсем уж *характерным*, ни выглядящим *проверяемым*в этом коде. Мы ясно видим лексическую область видимости, но замыкание остается какой-то мистической двигающейся тенью за кодом.

Тогда давайте посмотрим на код, который представит замыкание во всей красе:

function foo() {

var a = 2;

function bar() {

console.log( a );

}

return bar;

}

var baz = foo();

baz(); // 2 -- Ого, замыкание только что было раскрыто, мужики!

У функции bar() есть доступ в лексической области видимости к внутренней области видимости foo(). Но в то же время, мы берем bar(), саму функцию, и передаем ее *как* значение. В этом случае мы возвращаем сам объект функции, на который ссылается bar.

После выполнения foo() мы присвоим значение, которая она возвращает (нашу внутренную функцию bar()) в переменную baz, а затем мы фактически вызовем baz(), который конечно вызовет нашу внутренную функцию bar(), просто с помощью другой ссылки в идентификаторе.

Естественно bar() выполнится. Но в этом случае, она выполняется *снаружи* относительно своей объявленной лексической области видимости.

После выполнения foo(), обычно мы ожидаем, что вся внутренная область видимости foo() целиком будет удалена, поскольку мы знаем что *Движок* использует *сборщик мусора*, который исследует и освобождает память, которая больше не используется. Поскольку видно, что содержимое foo() больше не используется, кажется естественным, что оно должно быть *удалено* сборщиком.

Но "магия" замыканий не даст этому произойти. Эта внутренняя область видимости на самом деле *все еще*"используется", а потому не будет удалена. Кто ее использует? **Сама функция bar()**.

Благодаря тому, где она была объявлена, у bar() есть замыкание лексической области видимости на внутренную область видимости foo(), которая удерживает область видимости для bar(), чтобы ссылаться на нее позднее.

**bar() все еще содержит ссылку на эту область видимости и эта ссылка называется замыканием.**

Таким образом, несколькими микросекундами позже, когда переменная baz вызывается (вызывая внутреннюю функцию, которую мы изначально пометили как bar), у нее как и полагается есть *доступ* к лексической области видимости, определенной на этапе разработки, поэтому у нее есть доступ к переменной a как мы и ожидали.

Функция вызывается корректно вне ее лексической области видимости, определенной на этапе разработки. **Замыкание** дает функции возможность продлить доступ к лексической области видимости, которая была определена на этапе разработки.

Конечно, любой из различных путей, которыми можно эти функции *передавать* как значения, а следовательно вызывать в других местах, является примером контролирующего замыкания.

function foo() {

var a = 2;

function baz() {

console.log( a ); // 2

}

bar( baz );

}

function bar(fn) {

fn(); // смотри мам, я видел замыкание!

}

Мы передаем внутреннюю функцию baz в bar и вызываем ее (теперь как fn), и когда мы это делаем, ее замыкание на внутренню область видимости foo() соблюдается посредством доступа к a.

Такие передачи функций могут быть также и непрямыми.

var fn;

function foo() {

var a = 2;

function baz() {

console.log( a );

}

fn = baz; // присваиваем `baz` глобальной переменной

}

function bar() {

fn(); // смотри мам, я видел замыкание!

}

foo();

bar(); // 2

Какую бы возможность мы ни использовали, чтобы *передать* внутреннюю функцию вне ее лексической области видимости, эта возможность будет помогать удерживать ссылку на область видимости там, где она была объявлена изначально и когда бы мы ни выполняли ее это замыкание будет соблюдаться.

**Теперь я вижу**

Ранее приводимые части кода были немного академичны и искусственно сконструированы, чтобы проиллюстрировать *использование замыкания*. Но я обещал вам что-то большее, нежели просто новую крутую игрушку. Я обещал, что замыкание будет чем-то, что окружает вас повсюду в вашем коде. Давайте теперь *взглянем* на эту правду.

function wait(message) {

setTimeout( function timer(){

console.log( message );

}, 1000 );

}

wait( "Привет, замыкание!" );

Мы берем внутреннюю функцию (с именем timer) и передаем ее в setTimeout(..). Но у timer есть замыкание области видимости на область wait(..), разумеется с хранением и использованием ссылки на переменную message.

Через тысячу миллисекунд после того, как мы выполнили wait(..) и ее внутренняя область видимости, которой в ином случае следовало исчезнуть, у внутренней функции timer все еще есть замыкание на ту область видимости.

Глубоко во внутренностях *Движка*, встроенная функция setTimeout(..) держит ссылку на некоторый параметр, возможно названный fn или func, или как-то похоже. *Движок* выполняет эту функцию, которая вызывает нашу внутреннюю функцию timer, а ссылка на лексическую область видимости все еще остается целой.

### Замыкание

Или если вы из лагеря jQuery (или любого другого похожего JS фреймворка):

function setupBot(name,selector) {

$( selector ).click( function activator(){

console.log( "Активирую: " + name );

} );

}

setupBot( "Closure Bot 1", "#bot\_1" );

setupBot( "Closure Bot 2", "#bot\_2" );

Я не знаю какой код пишете вы, но я регулярно пишу код, который ответственен за управление всей глобальной армией ботов замыканий, так что это абсолютно правдоподобно!

Если серьезно, фактически *когда бы* и *где бы* вы ни обращались с функциями (у который есть доступ к их собственным лексическим областям видимости) как со значениями первого класса и ни передавали их повсюду, вы скорее всего увидите, что эти функции образуют замыкание. Будь это таймеры, обработчики событий, Ajax-запросы, кросс-оконные сообщения, вэб-воркеры или любые другие асинхронные (или синхронные!) задачи, когда вы передаете *колбэк-функцию*, приготовьтесь разбрасывать вокруг замыкания!

**Примечание:** Глава 3 рассказывала о шаблоне IIFE. Несмотря на то, что часто говорят, что IIFE (отдельно) — пример явного замыкания, я немного не соглашусь, исходя из нашего определения, данного выше.

var a = 2;

(function IIFE(){

console.log( a );

})();

Этот код "работает", но не выглядит как точная демонстрация замыкания. Почему? Потому что функция (которую мы назвали тут "IIFE") — не выполняется вне своей лексической области видимости. Она все еще вызывается прямо в той же области видимости, в которой была объявлена (окружающая/глобальная область видимости, которая хранит a). a обнаруживается через обычный поиск в лесической области видимости, а не через замыкание.

Несмотря на то, что замыкание может технически получаться во время объявления переменных, оно *не* обязательно наблюдаемо явно, а потому, как говорят, *это как опадающее дерево в лесу, которое никто не видит.*

Хотя IIFE *сама по себе* не является примером замыкания, она несомненно создает область видимости, и это один из самых распространенных инструментов, который мы используем для создания области видимости, которая может быть замкнута. Таким образом, IIFE, разумеется, тесно связана с замыканием, даже несмотря на то, что не производит самих замыканий.

Отложи сейчас эту книгу, дорогой читатель. У меня есть для тебя задача. Открой какой-нибудь свой недавний JavaScript-код. Поищи функции как значения и определи где ты уже используешь замыкания, где ты возможно их раньше не замечал.

Я подожду.

Теперь... ты видишь!

### Циклы + замыкание

Самый распространенный канонический пример, используемый для иллюстрации замыкания, приводят с использованием скромного цикла for.

for (var i=1; i<=5; i++) {

setTimeout( function timer(){

console.log( i );

}, i\*1000 );

}

**Примечание:** Линтеры частенько ругаются на то, что вы помещаете функции внутрь циклов, так как ошибки непонимания замыкания — **настолько типичны среди разработчиков**. Мы объясним, как в данном случае сделать правильно, раскрыв всю мощь замыкания. Но эта тонкость часто ускользает от линтеров и они ругаются в любом случае, предполагая что вы *на самом деле* не знаете, что делаете.

Сущность этого кусочка кода в том, что мы обычно *ожидаем* такого поведения, что в итоге будут распечатаны числа "1", "2", .. "5", по одному за раз, по одному в секунду соответственно.

На самом деле, если вы запустите этот код, вы получите вывод числа "6" 5 раз подряд, через односекундные интервалы.

**Да?**

Во-первых, позвольте объяснить откуда взялась цифра 6. Условие окончания цикла — когда i *не* <=5. Первый раз это происходит, когда i равно 6. Таким образом, вывод результата отражает конечное значение i после завершения цикла.

На самом деле это кажется очевидным при повторном взгляде. Все функции обратного вызова функции timeout запускаются прямо после завершения цикла. Фактически, пока тикают таймеры, даже если это был вызов setTimeout(.., 0) в каждой итерации цикла, все эти функции обратного вызова все равно будут запущены строго после выполнения цикла, и как следствие напечатают 6 каждый раз.

Но здесь есть более глубокий вопрос на повестке дня. Чего *не хватает* в нашем коде чтобы в самом деле вести себя так как мы себе это семантически представляли?

А не хватает того, что мы пытаемся *предполагать*, что каждая итерация цикла "захватывает" свою собственную копию i во время выполнения итерации. Но, тем путем, которым работает область видимости, все 5 этих функций, хотя они и определяются отдельно в каждой итерации цикла, все они **замыкаются на одну и ту же глобальную разделяемую область видимости**, в которой, фактически, только одна переменная i.

В таком свете, *конечно*, все функции разделяют ссылку на одну и ту же i. Кое-что в структуре циклов как правило смущает нас мыслями о том, что есть еще что-то сложное в их работе. Это не так. Нет никакой разницы по сравнению с тем, как если бы мы объявили каждый вызов timeout из 5 друг за другом вообще без всякого цикла.

Хорошо, итак вернемся обратно к нашему животрепещущему вопросу. Чего же не хватает? Нам нужна  более изолированная область видимости. Точнее говоря, нам нужна новая изолированная область видимости для каждой итерации цикла.

Мы изучили в главе 3, что IIFE создает область видимости объявляя и сразу выполняя функцию.

Давайте попробуем:

for (var i=1; i<=5; i++) {

(function(){

setTimeout( function timer(){

console.log( i );

}, i\*1000 );

})();

}

Это сработает? Попробуйте. Я снова жду.

Я не буду и дальше интриговать вас. **А вот и нет!** Но почему? Теперь у нас очевидно есть еще одна лексическая область видимости. Каждая функция обратного вызова для timeout, разумеется, замкнута на свою собственную область видимости на итерацию, создаваемую явно каждым вызовом IIFE.

Недостаточно иметь изолированную область видимости **если эта область пуста**. Взгляните поближе. Наша IIFE — всего лишь пустая ничего не делающая область видимости. Нужно *что-то* внутри нее, чтобы она стала полезной для нас.

Ей нужна ее собственная переменная, с копией значения i в каждой итерации.

for (var i=1; i<=5; i++) {

(function(){

var j = i;

setTimeout( function timer(){

console.log( j );

}, j\*1000 );

})();

}

**Эврика! Работает!**

Небольшая вариация на тему, которую некоторые предпочтут вышеуказанной:

for (var i=1; i<=5; i++) {

(function(j){

setTimeout( function timer(){

console.log( j );

}, j\*1000 );

})( i );

}

Конечно, поскольку эти IIFE — всего лишь функции, мы можем передать в них i и мы можем назвать передаваемый параметр j, если хотим, или даже можем снова назвать его i. В любом случае код теперь работает.

Использование IIFE внутри каждой итерации создало новую область видимости для каждой итерации, что дало обратным вызовам функции при вызове timeout возможность захватывать новую область видимости в каждой итерации, в каждой из которых есть переменная с правильным итерационным значением в ней.

Проблема решена!

**И вновь рассмотрим блочную область видимости**

Внимательно изучите наш анализ предыдущего решения. Мы использовали IIFE, чтобы создать новую область видимости на каждую итерацию. Иными словами, нам фактически *необходима* **блочная область видимости** для каждой итерации. Глава 3 открыла для нас объявление через let, которая "похищает" блок и объявляет переменную прямо в нем.

**Фактически она превратила блок в область видимости, которую мы можем охватить.** Таким образом, следующий прекрасный код "просто работает":

for (var i=1; i<=5; i++) {

let j = i; // да-да, блочная область видимости для замыкания!

setTimeout( function timer(){

console.log( j );

}, j\*1000 );

}

*Но и это еще не всё!* (голосом Якубовича). Есть особое поведение для объявлений let в заголовке цикла for. Это поведение определяет, что переменная будет объявлена не один раз для всего цикла, **а для каждой итерации**. И, она, конечно же, будет инициализирована в каждой последующей итерации значением с окончания предыдущей итерации.

for (let i=1; i<=5; i++) {

setTimeout( function timer(){

console.log( i );

}, i\*1000 );

}

Насколько это круто? Блочная область видимости и замыкание работают рука об руку, решая все мировые проблемы. Не знаю насчет вас, но меня это делает счастливым JavaScript-ером.

### Модули

Есть еще паттерны программирования, которые эффективно используют мощь замыканий, но которые внешне не особенно похожи на функции обратного вызова. Давайте рассмотрим самый мощный из них: *модуль*.

function foo() {

var something = "cool";

var another = [1, 2, 3];

function doSomething() {

console.log( something ); // "cool"

}

function doAnother() {

console.log( another.join( " ! " ) ); // "1 ! 2 ! 3"

}

}

Как видно из этого кода, здесь нет никакого явного замыкания. У нас просто есть некоторые приватные переменные something и another, а также парочка внутренних функций doSomething() и doAnother(), у обеих из которых есть лексическая область видимости (а потому и замыкание!) над внутренней областью foo().

А теперь смотрите:

function CoolModule() {

var something = "cool";

var another = [1, 2, 3];

function doSomething() {

console.log( something );

}

function doAnother() {

console.log( another.join( " ! " ) );

}

return {

doSomething: doSomething,

doAnother: doAnother

};

}

var foo = CoolModule();

foo.doSomething(); // cool

foo.doAnother(); // 1 ! 2 ! 3

Этот шаблон в JavaScript мы называем модуль. Самый распространенный путь реализации шаблона модуля часто называют "Действенный модуль" и это тот вариант, который мы тут и представили.

Давайте изучим некоторые факты об этом коде.

Во-первых, CoolModule() — просто функция, но ее *надлежит вызвать* для создания объекта модуля. Без выполнения внешней функци не случится ни создание внутренней области видимости, ни создание замыканий.

Во-вторых, функция CoolModule() возвращает объект, выполненный в синтаксисе объектного литерала { key: value, ... }. У объекта, который мы возвращаем, есть ссылки на наши внутренние функции, но *не* на наши внутренние переменные. Мы храним их скрытыми и приватными. Правильнее всего будет думать о возвращаемом значении в виде объекта, как по существу о **публичном API для нашего модуля**.

Это объектное возвращаемое значение в итоге присваивается внешей переменной foo, а затем мы можем получить доступ к этим методам в API, например к foo.doSomething().

**Примечание:** Не обязательно возвращать настоящий объект из нашего модуля. Мы могли бы просто вернуть напрямую внутреннюю функцию. jQuery как раз является хорошим тому примером. Идентификаторы jQuery и $ — это публичное API для "модуля" jQuery, но они сами по сути просто функции (которые сами по себе могут иметь свойства, поскольку все функции являются объектами).

У функций doSomething() и doAnother() есть замыкание на внутреннюю область видимости "экземпляра" модуля (полученного на самом деле вызовом CoolModule()). Когда мы передаем эти функции вне лексической области видимости путем ссылок на свойства объекта, который мы возвращаем, то мы в этот момент фактически получаем условия, при которых замыкание может появиться и выполниться.

Излагая проще, есть два "требования", которые должны выполняться для шаблона модуля:

1. Должна быть внешняя окружающая функция и она должны быть вызвана хотя бы раз (каждый раз создается новый экземпляр модуля).
2. Окружающая функция должна возвращать хотя бы одну внутреннюю функцию, для того, чтобы у этой внутренней функции было замыкание на приватную область видимости и был доступ и/или возможность изменения ее внутреннего состояния.

Объект со свойством-функцией сам по себе *на самом деле* не модуль. Объект, возвращаемый вызовом функции, у которого есть только свойства-данные и ни одной замыкающей функции *фактически* не является модулем, с точки зрения здравого смысла.

Вышеприведенный код показывает автономный конструктор модуля с названием CoolModule(), который можно вызвать любое количество раз, каждый раз создавая новый экземпляр модуля. Небольшая вариация этого шаблона — это когда вы заботитесь о том, чтобы был только один экземпляр, что-то вроде "синглтона":

var foo = (function CoolModule() {

var something = "cool";

var another = [1, 2, 3];

function doSomething() {

console.log( something );

}

function doAnother() {

console.log( another.join( " ! " ) );

}

return {

doSomething: doSomething, // объект { doSomething: doSomething, doAnother: doAnother }

doAnother: doAnother

};

})();

foo.doSomething(); // cool

foo.doAnother(); // 1 ! 2 ! 3

Тут мы превратили нашу модульную функцию в IIFE (см. главу 3), *сразу же* вызвали ее и присвоили возвращаемое ею значение прямо нашему единственному идентификатору foo.

Модули — это всего лишь функции, поэтому они могут принимать параметры:

function CoolModule(id) {

function identify() {

console.log( id );

}

return {

identify: identify

};

}

var foo1 = CoolModule( "foo 1" );

var foo2 = CoolModule( "foo 2" );

foo1.identify(); // "foo 1"

foo2.identify(); // "foo 2"

Еще одна небольшая, но полнофункциональная вариация модульного шаблона — дать имя объекту, который вы возвращаете как публичное API:

var foo = (function CoolModule(id) {

function change() {

// modifying the public API

publicAPI.identify = identify2;

}

function identify1() {

console.log( id );

}

function identify2() {

console.log( id.toUpperCase() );

}

var publicAPI = {

change: change,

identify: identify1

};

return publicAPI;

})( "foo module" );

foo.identify(); // foo module

foo.change();

foo.identify(); // FOO MODULE

Сохраняя внутреннюю ссылку на объект публичного API внутри экземпляра модуля вы можете менять эту ссылку на модуль **изнутри**, включая добавление и удаление методов, свойств *и* изменение их значений.

### Современные модули

Различные загрузчики/менеджеры модульных зависимостей фактически упаковывают это определение модульного шаблона в дружественное API. Прежде чем начать изучать любую отдельную библиотеку, позвольте мне показать *очень простой* экспериментальный вариант **только в иллюстративных целях**:

var MyModules = (function Manager() {

var modules = {};

function define(name, deps, impl) {

for (var i=0; i<deps.length; i++) {

deps[i] = modules[deps[i]];

}

modules[name] = impl.apply( impl, deps );

}

function get(name) {

return modules[name];

}

return {

define: define,

get: get

};

})();

Ключевая часть этого кода — modules[name] = impl.apply(impl, deps). Это вызов функции-обертки определения для модуля (с передачей внутрь любых зависимостей) и сохранение возвращаемого значения, API модуля, во внутренний список модулей со ссылкой по имени.

И вот как именно я мог бы его использовать для определения нескольких модулей:

MyModules.define( "bar", [], function(){

function hello(who) {

return "Let me introduce: " + who;

}

return {

hello: hello

};

} );

MyModules.define( "foo", ["bar"], function(bar){

var hungry = "hippo";

function awesome() {

console.log( bar.hello( hungry ).toUpperCase() );

}

return {

awesome: awesome

};

} );

var bar = MyModules.get( "bar" );

var foo = MyModules.get( "foo" );

console.log(

bar.hello( "hippo" )

); // Let me introduce: hippo

foo.awesome(); // LET ME INTRODUCE: HIPPO

Оба модуля "foo" и "bar" объявлены с функцией, которая возвращает публичное API. "foo" даже получает экземпляр "bar" как параметр-зависимость и может его использовать соответствующим образом.

Потратьте немного времени на изучение этих примеров кода, чтобы полностью осознать всю мощь замыканий и потом воспользоваться ими в собственных целях. Ключевой вывод тут в том, что нет никакого особого "волшебства" в модульных менеджерах. Они соответствуют обеим характеристикам шаблона модулей, которые я перечислил выше: вызов обертки определения функции и хранение возвращаемого значения в качестве API для этого модуля.

Иными словами, модули — это просто модули, даже если вы делаете ставку на хорошо знакомый вам инструмент для этого.

### Будущие модули

ES6 добавляет синтаксическую поддержку первого класса для концепции модулей. При загрузке через модульную систему, ES6 обрабатывает файл как отдельный модуль. Каждый модуль может как импортировать другие модули или отдельные члены из API, так и экспортировать свои собственные публичные члены API.

**Примечание:** Модули, основанные на функциях — не статически распознаваемый шаблон (о котором знает компилятор), поэтому их семантика API не учитывается до момента времени выполнения. То есть, вы фактически можете менять API модуля в процессе выполнения кода (см. ранее обсуждение публичного API).

В противоположность этому, модульное API ES6 — статическое (API не меняется во время выполнения). Поскольку компилятор *это* знает, он может (и делает!) проверить во время (загрузки файла и) компиляции, что ссылка на член API импортируемого модуля *действительно существует*. Если ссылка на API не существует, компилятор выдаст "заблаговременную" ошибку на этапе компиляции, вместо того, чтобы ждать традиционного динамического разрешения ссылок во время выполнения (и ошибок, если есть).

У модулей ES6 **нет** "встраиваемого" формата, они должны определяться в отдельных файлах (по одному на модуль). У браузеров/движков есть "загрузчик модулей" по умолчанию (который может быть переопределен, но это уже далеко за пределами нашего здесь обсуждения), который синхронно загружает файл модуля, когда он импортируется.

Рассмотрим код:

**bar.js**

function hello(who) {

return "Let me introduce: " + who;

}

export hello;

**foo.js**

// импортирует только `hello()` из модуля "bar"

import hello from "bar";

var hungry = "hippo";

function awesome() {

console.log(

hello( hungry ).toUpperCase()

);

}

export awesome;

// импортирует модули "foo" и "bar" целиком

module foo from "foo";

module bar from "bar";

console.log(

bar.hello( "rhino" )

); // Let me introduce: rhino

foo.awesome(); // LET ME INTRODUCE: HIPPO

**Примечание:** Необходимо создать отдельные файлы **"foo.js"** и **"bar.js"**, с указанным выше содержимым. Затем ваша программа загрузит/импортирует эти модули, чтобы использовать их как показано в третьем фрагменте кода.

import импортирует один или более членов API модуля в текущую область видимости, каждый в свою переменную (hello в нашем случае). module импортирует API модуля целиком в указанную переменную (foo, bar в нашем случае). export экспортирует идентификатор (переменную, функцию) в публичное API текущего модуля. Эти операторы можно использовать в определении модуля столько раз, сколько потребуется.

Содержимое внутри *файла модуля* обрабатывается как если бы оно было заключено в замыкание области видимости, также как и у модулей с функцией-замыканием, рассмотренных ранее.

### Итог

Похоже, что знания о замыкании полны предрассудков и суеверий как загадочный мир, стоящий особняком внутри JavaScript, который могут познать только самые храбрые души. Но на самом деле — это всего лишь стандартный и почти очевидный факт о том, как писать код в среде лексической области видимости, где функции являются значениями и могут свободно передаваться куда угодно.

**Замыкание — это когда функция может запомнить и иметь доступ к своей лексической области видимости даже тогда, когда она вызывается вне своей лексической области видимости.**

Замыкания могут сбить нас с толку, например в циклах, если мы не озаботимся тем, чтобы распознавать их и то как они работают. Но они еще и являются весьма мощным инструментом, дающим доступ к шаблонам, таким как *модули* в их различных формах.

Модули требуют две ключевых характеристики: 1) внешнюю функцию-обертку, которую будут вызывать, чтобы создать закрытую область видимости 2) возвращаемое значение функции-обертки должно включать в себя ссылку на не менее чем одну внутреннюю функцию, у которой потом будет замыкание на внутреннюю область видимости обертки.

Теперь вы сможете заметить замыкания повсюду в своем существующем коде и у нас теперь есть возможность обнаруживать и использовать все их преимущества!

## Приложение A: Динамическая область видимости

В главе 2 мы говорили о "динамической области видимости" как о противопоставлении модели "лексической области видимости", которая и есть то, как работает область видимости в JavaScript (а по факту и во многих других языках).

Мы кратко рассмотрим динамическую область видимости, чтобы крепко усвоить разницу. Но, важнее то, что динамическая область видимости — ближайшая родственница другого механизма (this) в JavaScript, который мы рассмотрели в книге "*this и прототипы объектов*".

Как мы уже отметили в главе 2, лексическая область видимости — это набор правил о том, как именно *Движок* может искать переменную и как он ее может найти. Ключевая характеристика лексической области видимости — она определяется на этапе написания кода (предполагая, что вы не жульничаете с помощью eval() или with).

Динамическая область видимости похоже означает, и не зря, что есть модель, при помощи которой область видимости можно определить динамически во время выполнения, вместо статического определения при написании кода. То, что нам нужно. Давайте отразим это в коде:

function foo() {

console.log( a ); // 2

}

function bar() {

var a = 3;

foo();

}

var a = 2;

bar();

Лексическая область видимости хранит информацию о том, что RHS-ссылка на a в foo() будет разрешена в глобальную переменную a, что приведет к тому, что будет выведено значение 2.

Динамическая область видимости, напротив, не интересуется тем как и где были объявлены функции и области видимости, а скорее интересуется тем **откуда они буду вызываться**. Иными словами, здесь цепочка областей видимости основана на стеке вызовов, а не на вложенности областей видимости в коде.

Итак, если бы в JavaScript была динамическая область видимости, то когда выполнилась бы foo(), **теоретически** нижеприведенный код привел бы к выводу 3.

function foo() {

console.log( a ); // 3 (not 2!)

}

function bar() {

var a = 3;

foo();

}

var a = 2;

bar();

Как такое может быть? А всё потому, что когда foo() не может разрешить ссылку на переменную a, вместо поднятия по цепочке вложенных лексических областей видимости, она взбирается вверх по стеку вызовов, чтобы найти откуда foo() была *вызвана*. Поскольку foo() вызывалась из bar(), она проверяет переменные в области видимости bar()и находит там a со значением 3.

Странно? Возможно вы так и подумали, на какой-то момент.

Но это всего лишь потому, что вы возможно работали только с (или по меньшей мере глубоко изучили) кодом, который работает в лексической областью видимости. Поэтому то динамическая область видимости и кажется чужой. Если бы вы писали код на языке с динамической областью видимости, для вас всё это показалось бы естественным, а лексическая область видимости показалось бы чудаковатой.

Чтобы внести ясность, в JavaScript **нет, на самом деле, динамической области видимости**. В нем есть лексическая область видимости. Проще некуда. Но механизм работы this немного похож на динамическую область видимости.

Ключевое сравнение: **лексическая область видимости определяется временем написания кода, тогда как динамическая область видимости (и this!) определяется во время выполнения**. Лексическую область видимости интересует *где функция была объявлена*, а динамическую — *откуда была вызвана* функция.

И наконец: this интересует *как была вызвана функция*, что показывает как близко связаны механизм this с идеей динамической области видимости. Чтобы изучить во всех подробностях this, прочтите книгу "*this и прототипы объектов*".

## Приложение C: Лексический this

Хотя эта книга и не рассматривает механизм this во всех деталях, есть одна тема в ES6, которая связывает this с лексической областью видимости существенным образом, по которому мы быстро пробежимся.

ES6 добавляет особую синтаксическую форму объявления функции, названную "стрелочная функция". Она выглядит примерно так:

var foo = a => {

console.log( a );

};

foo( 2 ); // 2

Так называемая "жирная стрелка" часто упоминается как сокращение для *утомительно длинного* (сарказм) ключевого слова function.

Но есть кое-что более важное в стрелочных функциях, что не имеет ничего общего с экономией символов в вашем коде.

В двух словах, этот код страдает от одной проблемы:

var obj = {

id: "awesome",

cool: function coolFn() {

console.log( this.id );

}

};

var id = "not awesome";

obj.cool(); // awesome

setTimeout( obj.cool, 100 ); // not awesome

Проблема заключается в потере привязки this в функции cool(). Есть разные пути решения этой проблемы, но наиболее частое решение — var self = this;.

Выглядит это примерно так:

var obj = {

count: 0,

cool: function coolFn() {

var self = this;

if (self.count < 1) {

setTimeout( function timer(){

self.count++;

console.log( "красиво?" );

}, 100 );

}

}

};

obj.cool(); // красиво?

Чтобы не слишком углубляться в подробности, "решение "var self = this всего лишь избавляется целиком от всей проблемы понимания и правильного использования привязки this, а взамен возвращается к чему-то более удобному для нас: лексической области видимости. self становится всего лишь идентификатором, который может быть определен с помощью лексической области видимости и замыкания, и не заботится о том, что случится с привязкой this по пути.

Люди не любят писать подробно особенно то, что делают снова и снова. Таким образом, мотивацией ES6 является помощь в облегчении таких сценариев и разумеется в *устранении* общих проблем идиом, таких как эта.

Решение в ES6, стрелочная функция, вводит поведение, называемое "лексический this".

var obj = {

count: 0,

cool: function coolFn() {

if (this.count < 1) {

setTimeout( () => { // стрелочная функция, выигрышный вариант?

this.count++;

console.log( "красиво?" );

}, 100 );

}

}

};

obj.cool(); // красиво?

Вкратце, стрелочные функции ведут себя совсем не как обычные функции в том, что касается их привязки к this. Они отбрасывают все обычные правила для привязки this, а взамен берут значение this из их непосредственной окружаюшей области видимости, неважно из какой.

Так что в этом примере кода стрелочная функция не получает свой this непривязанным каким-то непредсказуемым путем, она всего лишь "наследует" привязку this функции cool() (что правильно, если мы вызываем ее так, как показано выше!).

Несмотря на то, что это делает код короче, моя точка зрения в том, что стрелочные функции — всего лишь на самом деле закодированная в синтаксис языка распространенная *ошибка* разработчиков, которая приводит к тому, чтобы запутать и соединить правила "привязки this" с правилами "лексической области видимости".

Другими словами: зачем искать неприятности и ударяться в словоблудие используя парадигму кодирования стиля this, всего лишь чтобы подрезать ему крылья смешивая его с лексическими ссылками. Кажется естественным принять тот или иной подход для любой конкретной части кода, а не смешивать их в одном и том же месте.

**Примечание:** еще один недостаток стрелочных функций в том, что они анонимны, не именованы. Загляните в главу 3, чтобы ознакомиться с причинами почему анонимные функции менее предпочтительны, чем именованные.

Более подходящий подход, с моей точки зрения, к этой "проблеме", использовать и рассматривать механизм thisправильно.

var obj = {

count: 0,

cool: function coolFn() {

if (this.count < 1) {

setTimeout( function timer(){

this.count++; // `this` безопасен из-за `bind(..)`

console.log( "еще красивее" );

bind( this ), 100 ); // смотри, `bind()`!

}

}

};

obj.cool(); // еще красивее

Чтобы вы ни предпочли: новое поведение лексического this стрелочных функций или испытанный и верный bind(), важно отметить, что стрелочные функции — **не только** сокращение написания "function".

У них есть *намеренная разница в поведении*, которую необходимо изучить и понимать, и если мы их выбираем, то использовать по максимуму их возможности.

Теперь, когда мы полностью понимаем образование лексической области видимости (и замыкания!), понять лексический this будет проще простого!

# This и Прототипы Объектов (Don`t know JS)

## Глава 1: this (тут) или That (там)?

Одним из наиболее запутанных механизмов в Javascript является ключевое слово this. Это специальное ключевое слово идентификатор, которое автоматически определяется внутри области видимости каждой функции, но к чему именно оно относится сбивает с толку даже опытных Javascript-разработчиков.

Любая достаточно продвинутая технология неотличима от магии. -- Артур Си. Клэрк

Механизм this Javascript на самом деле не такой уж и продвинутый, но разработчики часто перефразируют эту цитату вставив "сложный" или "сбивающий с толку", и совершенно понятно, что без четкого понимания это может казаться совершенно магическим в вашем понимании.

**Примечание:** Слово "this" — это достаточно распространенное местоимение в общих беседах. Поэтому, может быть очень сложно, особенно на словах, определить используем мы "this" как местоимение или же используем его, чтобы ссылаться на данное ключевое слово. Для ясности, я всегда буду использовать this для ссылки на специальное ключевое слово, а "this" или *this* или this в остальных случаях.

### Зачем нужен this?

Раз механизм this такой запутанный даже для опытных JavaScript-разработчиков, можно задаться вопросом, а точно ли он полезный? Может у него больше недостатков, чем достоинств? Перед тем, как перейти к тому *как он работает*, мы должны проанализировать *зачем он нужен*.

Давайте попытаемся проиллюстрировать мотивацию и полезность механизма this:

function identify() {

return this.name.toUpperCase();

}

function speak() {

var greeting = "Hello, I'm " + identify.call( this );

console.log( greeting );

}

var me = {

name: "Kyle"

};

var you = {

name: "Reader"

};

identify.call( me ); // KYLE

identify.call( you ); // READER

speak.call( me ); // Hello, I'm KYLE

speak.call( you ); // Hello, I'm READER

Если то, *как работает* этот фрагмент кода путает вас, не волнуйтесь! Мы скоро вернемся к этому. Просто отложите ваши вопросы в сторону, чтобы мы могли более четко взглянуть на то, *зачем* это нужно.

Этот фрагмент кода позволяет функциям identify() и speak() быть переиспользованными с разными объектами *контекста* (me и you), а не требовать новой версии функции для каждого объекта.

Вместо того, чтобы полагаться на this, вы могли бы явно передать *объект контекста* функциям identify() и speak().

function identify(context) {

return context.name.toUpperCase();

}

function {

var greeting = "Hello, I'm " + identify( context );

console.log( greeting );

}

identify( you ); // READER

speak( me ); // Hello, I'm KYLE

Однако, механизм this предоставляет более элегантный путь, неявно "передавая" ссылку на объект, что приводит к чистому дизайну API и облегчению повторного переиспользования.

Чем сложнее будет используемый вами паттерн, тем более ясно вы увидите, что указание контекста явным параметром часто запутаннее, чем неявное указание контекста this. Когда мы изучим объекты и прототипы, вы увидите полезность коллекции функций, которые способны автоматически ссылаться на правильный объект контекста.

**Заблуждения**

Мы скоро объясним как this *на самом деле* работает, но сначала мы должны рассеять несколько заблуждений о том, как он *на самом деле не* работает.

Имя "this" создает заблуждение, когда разработчики пытаются думать о нем слишком буквально. Есть два часто предполагаемых значения, но оба являются неверными.

**Сама функция**

Первый общий соблазн это предполагать, что this ссылается на саму функцию. Это, как минимум, резонное грамматическое заключение.

Но зачем вы бы хотели ссылаться на функцию из неё же? Наиболее распространенной причиной может быть такая вещь как рекурсия(вызов функции внутри себя) или чтобы назначить обработчик события, который сработает при вызове функции.

Разработчики, незнакомые с механизмами JavaScript, часто думают, что ссылка на функцию как на объект(все функции в JavaScript являются объектами!) позволяет хранить состояния(значения в свойствах) между вызовами функций. Хотя это, конечно, возможно, но это имеет некоторые ограничения в использовании, остаток книги будет повествовать о многих других шаблонах для *лучшего* хранения состояния, чем объект функции.

Но для начала мы используем этот шаблон, чтобы проиллюстрировать как this не дает функции получить ссылку на саму себя, как мы могли бы предположить.

Рассмотрим следующий код, где мы попытаемся отследить сколько раз функция (foo) была вызвана:

function foo(num) {

console.log( "foo: " + num );

// Отслеживаем сколько раз `foo` была вызвана

this.count++;

}

foo.count = 0;

var i;

for (i=0; i<10; i++) {

if (i > 5) {

foo( i );

}

}

// foo: 6

// foo: 7

// foo: 8

// foo: 9

// Сколько раз была вызвана `foo`?

console.log( foo.count ); // 0 -- WTF?

foo.count *до сих пор* равен 0, даже не смотря на то, что 4 инструкции console.log очевидно показывают, что foo(..) на самом деле была вызвана 4 раза. Разочарование происходит от *слишком буквального* толкования того, что означает this (в this.count++).

Когда код выполняет команду foo.count = 0, он на самом деле добавляет свойство count в объект функции foo. Но для ссылки this.count внутри функции this фактически не указывает на тот же объект функции, и несмотря на то, что имена свойств одинаковые, это разные объекты, вот тут то и начинается неразбериха.

**Примечание:** ответственный разработчик в этом месте должен спросить: "Если я увеличил свойство count, но оно не то, которое я ожидал, то какое count было мной увеличено?". На самом деле, если он копнет глубже, он обнаружит что случайно создал глобальную переменную count(смотрите в главе 2 *как* это произошло!), а её текущим значением является NaN. Конечно, после того, как он определит это, у него появится совсем другой ряд вопросов: "почему она стала глобальной и почему она имеет значение NaN, вместо правильного значения счетчика?". (см. главу 2).

Вместо того, чтобы остановиться на этом месте и копнуть глубже, чтобы узнать почему ссылка this не ведет себя как *ожидалось*, большинство разработчиков просто откладывают проблему целиком и ищут другие решения, например, создают другой объект для хранения свойства count:

function foo(num) {

console.log( "foo: " + num );

// отслеживаем сколько раз вызывалась `foo`

data.count++;

}

var data = {

count: 0

};

var i;

for (i=0; i<10; i++) {

if (i > 5) {

foo( i );

}

}

// foo: 6

// foo: 7

// foo: 8

// foo: 9

// сколько раз вызывалась `foo`?

console.log( data.count ); // 4

Хоть это и верно, что этот подход "решает" проблему, к сожалению, это просто игнорирование реальной проблемы — недостатка понимания того, что значит this и как он работает и вместо этого возвращение в зону комфорта более простого механизма: области видимости.

**Примечание:** Области видимости - замечательный и полезный механизм. Я не против использования их любым способом(см. книгу *"Области видимости и замыкания"* из этой серии книг). Но постоянно гадать, как использовать this, и, как правило, ошибаться — не лучшая причина возвращаться к областям видимости и никогда не узнать *почему* this ускользает от вас.

Для ссылки на объект функции изнутри этой функции, this самого по себе обычно бывает недостаточно. Вам обычно нужна ссылка на объект функции через лексический идентификатор (переменную), который указывает на него.

Рассмотрим эти 2 функции:

function foo() {

foo.count = 4; // `foo` ссылается на саму себя

}

setTimeout( function(){

// анонимная функция (без имени), не может

// ссылаться на себя

}, 10 );

В первой функции вызывалась "именованная функция", foo — это ссылка, которая может быть использована для ссылки на функцию из самой себя.

Но во втором примере функция обратного вызова, передаваемая в setTimeout(..), не имела имени идентификатора (так называемая "анонимная функция"), так что у неё нет правильного пути чтобы обратиться к её объекту.

**Примечание:** Старомодная, но ныне устаревшая и неиспользуемая ссылка arguments.callee внутри функции *также*указывает на объект функции, которая в данный момент выполняется. Эта ссылка обычно используется как возможность получить объект анонимной функции изнутри этой функции. Лучший подход, однако, состоит в том, чтобы избежать использования анонимных функций, по крайней мере тех, которые требуют обращения к себе изнутри, и вместо них использовать именованные функции. arguments.callee устарела и не должна использоваться.

Таким образом, другое решение нашего примера — это использовать идентификатор foo как ссылку на объект функции в каждом месте и вообще не использовать this, и это *работает*:

function foo(num) {

console.log( "foo: " + num );

// следим, сколько раз вызывается функция

foo.count++;

}

foo.count = 0;

var i;

for (i=0; i<10; i++) {

if (i > 5) {

foo( i );

}

}

// foo: 6

// foo: 7

// foo: 8

// foo: 9

// сколько раз `foo` была вызвана?

console.log( foo.count ); // 4

Однако, этот подход также является уклонением от фактического понимания this, и полностью зависит от области видимости переменной foo.

Еще один путь решения проблемы - это заставить this действительно указывать на объект функции foo:

function foo(num) {

console.log( "foo: " + num );

// следим, сколько раз вызывается функция

// Заметьте: `this` теперь действительно ссылается на `foo`, это основано на том,

// как `foo` вызывается (см. ниже)

this.count++;

}

foo.count = 0;

var i;

for (i=0; i<10; i++) {

if (i > 5) {

// используя `call(..)` мы гарантируем что `this`

// ссылается на объект функции (`foo`) изнутри

foo.call( foo, i );

}

}

// foo: 6

// foo: 7

// foo: 8

// foo: 9

// сколько раз `foo` была вызвана?

console.log( foo.count ); // 4

**Вместо избегания this, мы воспользовались им.** Мы отведем немного времени на то, чтобы объяснить более детально как такие методы работают, так что не волнуйтесь если вы до сих пор недоумеваете как это работает!

**Это область видимости функции**

Следующее большое общее заблуждение касательно того, на что указывает this - это то, что он каким-то образом ссылается на область видимости функции. Это очень сложный вопрос, потому что с одной стороны так и есть, но с другой это совершенно не так.

Для ясности, this, в любом случае, не ссылается на область видимости функции. Это правда, что внутри область видимости имеет вид объекта со свойствами для каждого определенного значения. Но "объект" области видимости не доступен в JavaScript коде. Это внутренняя часть механизма реализации языка (интерпретатора).

Рассмотрим код, который пытается (и безуспешно!) перейти границу и использовать this неявно ссылаясь на область видимости функции:

function foo() {

var a = 2;

this.bar();

}

function bar() {

console.log( this.a );

}

foo(); //undefined

В этом коде содержится более одной ошибки. Хотя он может казаться надуманным, код который вы видите — это фрагмент из реального практического кода, которым обменивались в публичных форумах сообщества. Это замечательная (если не печальная) иллюстрация того, насколько ошибочным может быть предположение о this.

Во-первых, попытка ссылаться на функцию bar() как this.bar(). Это почти наверняка *случайность*, что это работает, но мы коротко объясним *как* это работает позже. Наиболее естественным путем вызвать bar() было бы опустить предшествующий this. и просто сделать ссылку на идентификатор.

Однако, разработчик, который писал этот код, пытался использовать this, чтобы создать мост между областями видимости foo() и bar() так, чтобы bar() получила доступ к переменной a внутри области видимости foo(). **Не всякий мост возможен.** Вы не можете использовать ссылку this, чтобы найти что-нибудь в области видимости. Это невозможно.

Каждый раз, когда вы чувствуете, что вы смешиваете поиски в области видимости с this, напоминайте себе: *это не мост*.

### Что же такое this?

Оставив ошибочные предположения, давайте обратим наше внимание на то, как механизм this действительно работает.

Мы ранее сказали, что this привязывается не во время написания функции, а во время её вызова. Это вытекает из контекста, который основывается на обстоятельствах вызова функции. Привязка this не имеет ничего общего с определением функции, но зависит от того при каких условиях функция была вызвана.

Когда функция вызывается, создается запись активации, также известная как контекст вызова. Эта запись содержит информацию о том, откуда функция была вызвана (стэк вызова), *как* функция была вызвана, какие параметры были в неё переданы и т.д. Одним из свойств этой записи является ссылка this, которая будет использоваться на протяжении выполнения этой функции.

В следующей главе мы научимся находить **место вызова** функции, чтобы определить как оно связано с определением this

### Обзор (TL;DR)

Определение this - постоянный источник заблуждений для JavaScript разработчиков, которые не уделяют времени на изучение того, как этот механизм в действительности работает. Гадать, методом проб и ошибок, и слепо копировать код из StackOverflow - неэффективный и неправильный путь использовать этот важный механизм this .

Чтобы понять что такое this, вам сначала нужно понять чем this не является, несмотря на любые предположения или заблуждения, которые могут тянуть вас вниз. this — это не ссылка функции на саму себя и это не ссылка на область видимости функции.

В действительности this — это привязка, которая создается во время вызова функции, и на *что* она ссылается определяется тем, где и при каких условиях функция была вызвана.

## Глава 2: Весь this теперь приобретает смысл!

В главе 1 мы отбросили различные ложные представления о this и взамен изучили, что привязка this происходит при каждом вызове функции, целиком на основании ее **места вызова** (как была вызвана функция).

### Точка вызова

Чтобы понять привязку this, мы должны понять что такое точка вызова: это место в коде, где была вызвана функция (**не там, где она объявлена**). Мы должны исследовать точку вызова, чтобы ответить на вопрос: на что же *этот* this указывает?

В общем поиск точки вызова выглядит так: "найти откуда вызывается функция", но это не всегда так уж легко, поскольку определенные шаблоны кодирования могут ввести в заблуждение относительно *истинной* точки вызова.

Важно поразмышлять над **стеком вызовов** (стеком функций, которые были вызваны, чтобы привести нас к текущей точке исполнения кода). Точка вызова, которая нас интересует, находится *в* вызове *перед* текущей выполняемой функцией.

Продемонстрируем стек вызовов и точку вызова:

function baz() {

// стек вызовов: `baz`

// поэтому наша точка вызова — глобальная область видимости

console.log( "baz" );

bar(); // <-- точка вызова для `bar`

}

function bar() {

// стек вызовов: `baz` -> `bar`

// поэтому наша точка вызова в `baz`

console.log( "bar" );

foo(); // <-- точка вызова для `foo`

}

function foo() {

// стек вызовов: `baz` -> `bar` -> `foo`

// поэтому наша точка вызова в `bar`

console.log( "foo" );

}

baz(); // <-- точка вызова для `baz`

Позаботьтесь при анализе кода о том, чтобы найти настоящую точку вызова (из стека вызовов), поскольку это единственная вещь, которая имеет значение для привязки this.

**Примечание:** Вы можете мысленно визуализировать стек вызовов посмотрев цепочку вызовов функций в том порядке, в котором мы это делали в комментариях в коде выше. Но это утомительно и чревато ошибками. Другой путь посмотреть стек вызовов — это использование инструмента отладки в вашем браузере. Во многих современных настольных браузерах есть встроенные инструменты разработчика, включающие JS-отладчик. В вышеприведенном коде вы могли бы поставить точку остановки в такой утилите на первой строке функции foo() или просто вставить оператор debugger; в первую строку. Как только вы запустите страницу, отладчик остановится в этом месте и покажет вам список функций, которые были вызваны, чтобы добраться до этой строки, каковые и будут являться необходимым стеком вызовов. Таким образом, если вы пытаетесь выяснить привязку this, используйте инструменты разработчика для получения стека вызовов, затем найдите второй элемент стека от его вершины и это и будет реальная точка вызова.

**Ничего кроме правил**

Теперь обратим наш взор на то, *как* точка вызова определяет на что будет указывать this во время выполнения функции.

Вам нужно изучить точку вызова и определить какое из 4 правил применяется. Сначала разъясним каждое из 4 правил по отдельности, а затем проиллюстрируем их порядок приоритета, для случаев когда к точке вызова *могут*применяться несколько правил сразу.

### Привязка по умолчанию

Первое правило, которое мы изучим, исходит из самого распространенного случая вызовов функции: отдельный вызов функции. Представьте себе *это* правило this как правило, действующее по умолчанию когда остальные правила не применяются.

Рассмотрим такой код:

function foo() {

console.log( this.a );

}

var a = 2;

foo(); // 2

Первая вещь, которую можно отметить, если вы еще не сделали этого, то, что переменные, объявленные в глобальной области видимости, как например var a = 2, являются синонимами глобальных свойств-объектов с таким же именем. Они не являются копиями друг друга, они и *есть* одно и то же. Представляйте их как две стороны одной монеты.

Во-вторых, видно, что когда вызывается foo() this.a указывает на нашу глобальную переменную a. Почему? Потому что в этом случае, для this применяется *привязка по умолчанию* при вызове функции и поэтому this указывает на глобальный объект.

Откуда мы знаем, что здесь применяется *привязка по умолчанию*? Мы исследуем точку вызова, чтобы выяснить как вызывается foo(). В нашем примере кода foo() вызывается по прямой, необернутой ссылке на функцию. Ни одного из демонстрируемых далее правил тут не будет применено, поэтому вместо них применяется *привязка по умолчанию*.

Когда включен strict mode, объект 'global' не подпадает под действие *привязки по умолчанию*, поэтому в противоположность обычному режиму this устанавливается в undefined.

function foo() {

"use strict";

console.log( this.a );

}

var a = 2;

foo(); // TypeError: `this` is `undefined`

Едва уловимая, но важная деталь: даже если все правила привязки this целиком основываются на точке вызова, глобальный объект подпадает под *привязку по умолчанию* **только** если **содержимое** foo() **не** выполняется в режиме strict mode; Состояние strict mode в точке вызова foo() не имеет значения.

function foo() {

console.log( this.a );

}

var a = 2;

(function(){

"use strict";

foo(); // 2

})();

**Примечание:** К намеренному смешиванию включения и выключения strict mode в коде обычно относятся неодобрительно. Вся программа пожалуй должна быть либо **строгой**, либо **нестрогой**. Однако, иногда вы подключаете сторонние библиотеки, в которых этот режим **строгости** отличается от вашего, поэтому нужно отнестись с вниманием к таким едва уловимым деталям совместимости.

**Неявная привязка**

Рассмотрим еще одно правило: есть ли у точки вызова объект контекста, также называемый как владеющий или содержащий объект, хотя *эти* альтернативные термины могут немного вводить в заблуждение.

Рассмотрим:

function foo() {

console.log( this.a );

}

var obj = {

a: 2,

foo: foo

};

obj.foo(); // 2

Во-первых, отметим способ, которым была объявлена foo(), а затем позже добавлена как ссылочное свойство в obj. Независимо от того была ли foo() изначально объявлена *в* obj или добавлена позднее как ссылка (как в вышеприведенном коде), ни в том, ни в другом случае **функция** на самом деле не "принадлежит" или "содержится" в объекте obj.

Однако, точка вызова *использует* контекст obj, чтобы **ссылаться** на функцию, поэтому *можно* сказать, что объект obj "владеет" или "содержит" **ссылку на функцию** в момент вызова функции.

Какое название вы бы ни выбрали для этого шаблона, в момент когда вызывается foo(), ей предшествует объектная ссылка на obj. Когда есть объект контекста для ссылки на функцию, правило *неявной привязки* говорит о том, что именно *этот* объект и следует использовать для привязки this к вызову функции.

Поскольку obj является this для вызова foo(), this.a — синоним obj.a.

Только верхний/последний уровень ссылки на свойство объекта в цепочке имеет значение для точки вызова. Например:

function foo() {

console.log( this.a );

}

var obj2 = {

a: 42,

foo: foo

};

var obj1 = {

a: 2,

obj2: obj2

};

obj1.obj2.foo(); // 42

**Неявно потерянный**

Одним из самых распространенных недовольств, которые вызывает привязка this — когда *неявно привязанная* функция теряет эту привязку, что обычно означает что она вернется к *привязке по умолчанию*, либо объекта global, либо undefined, в зависимости от режима strict mode.

Представим такой код:

function foo() {

console.log( this.a );

}

var obj = {

a: 2,

foo: foo

};

var bar = obj.foo; // ссылка/алиас на функцию!

var a = "ой, глобальная"; // `a` также и свойство глобального объекта

bar(); // "ой, глобальная"

Несмотря на то, что bar по всей видимости ссылка на obj.foo, фактически, это на самом деле другая ссылка на саму foo. Более того, именно точка вызова тут имеет значение, а точкой вызова является bar(), который является прямым непривязанным вызовом, а следовательно применяется *привязка по умолчанию*.

Более неочевидный, более распространенный и более неожиданный путь получить такую ситуацию когда мы предполагаем передать функцию обратного вызова:

function foo() {

console.log( this.a );

}

function doFoo(fn) {

// `fn` — просто еще одна ссылка на `foo`

fn(); // <-- точка вызова!

}

var obj = {

a: 2,

foo: foo

};

var a = "ой, глобальная"; // `a` еще и переменная в глобальном объекте

doFoo( obj.foo ); // "ой, глобальная"

Передаваемый параметр — всего лишь неявное присваивание, а поскольку мы передаем функцию, это неявное присваивание ссылки, поэтому окончательный результат будет таким же как в предыдущем случае.

Что если функция, которую вы передаете в качестве функции обратного вызова, не ваша собственная, а встроенная в язык? Никакой разницы, такой же результат.

function foo() {

console.log( this.a );

}

var obj = {

a: 2,

foo: foo

};

var a = "ой, глобальная"; // `a` еще и переменная в глобальном объекте

setTimeout( obj.foo, 100 ); // "ой, глобальная"

Поразмышляйте над этой грубой теоретической псевдо-реализацией setTimeout(), которая есть в качестве встроенной в JavaScript-среде:

function setTimeout(fn,delay) {

// подождать (так или иначе) `delay` миллисекунд

fn(); // <-- точка вызова!

}

Достаточно распространенная ситуация, когда функции обратного вызова *теряют* свою привязку this, как мы только что видели. Но еще один способ, которым this может удивить нас, когда функция, которой мы передаем нашу функцию обратного вызова, намеренно меняет this для этого вызова. Обработчики событий в популярных JavaScript-библиотеках часто любят, чтобы в вашей функции обратного вызова this принудительно указывал, например, на DOM-элемент, который вызвал это событие. Несмотря на то, что иногда это бывает полезно, в другое время это может прямо таки выводить из себя. К сожалению, эти инструменты редко дают возможность выбирать.

Каким бы путем ни менялся неожиданно this, у вас в действительности нет контроля над тем как будет вызвана ваша функция обратного вызова, таким образом у вас нет возможности контролировать точку вызова, чтобы получить заданную привязку. Мы кратко рассмотрим способ "починки" этой проблемы *починив* this.

**Явная привязка**

В случае *неявной привязки*, как мы только что видели, нам требуется менять объект, о котором идет речь, чтобы включить в него функцию и использовать эту ссылку на свойство-функцию, чтобы опосредованно (неявно) привязать this к этому объекту.

Но, что если вам надо явно использовать при вызове функции указанный объект для привязки this, без помещения ссылки на свойство-функцию в объект?

У "всех" функций в языке есть несколько инструментов, доступных для них (через их [[Прототип]], о котором подробности будут позже), которые могут оказаться полезными в решении этой задачи. Говоря конкретнее, у функций есть методы call(..) и apply(..) . Технически, управляющие среды JavaScript иногда обеспечивают функции, которые настолько специфичны, что у них нет такой функциональности. Но таких мало. Абсолютное большинство предоставляемых функций и конечно все функции, которые создете вы сами, безусловно имеют доступ к call(..) и apply(..).

Как работают эти инструменты? Они оба принимают в качестве первого параметра объект, который будет использоваться в качестве this, а затем вызывают функцию с указанным this. Поскольку вы явно указываете какой this вы хотите использовать, мы называем такой способ *явной привязкой*.

Представим такой код:

function foo() {

console.log( this.a );

}

var obj = {

a: 2

};

foo.call( obj ); // 2

Вызов foo с *явной привязкой* посредством foo.call(..) позволяет нам указать, что this будет obj.

Если в качестве привязки this вы передадите примитивное значение (типа string, boolean или number), то это примитивное значение будет обернуто в свою объектную форму (new String(..), new Boolean(..) или new Number(..) соответственно). Часто это называют "упаковка".

\**Примечание:* \* В отношении привязки this call(..) и apply(..) идентичны. Они *по-разному* ведут себя с дополнительными параметрами, но мы не будем сейчас на этом останавливаться.

К сожалению, *явная привязка* сама по себе все-таки не предлагает никакого решения для указанной ранее проблемы "потери" функцией ее привязки this, либо оставляет это на усмотрение фреймворка.

**Жесткая привязка**

Но поиграв с вариациями на тему *явной привязки* на самом деле можно получить желаемое. Пример:

function foo() {

console.log( this.a );

}

var obj = {

a: 2

};

var bar = function() {

foo.call( obj );

};

bar(); // 2

setTimeout( bar, 100 ); // 2

// `bar` жестко привязывает `this` в `foo` к `obj`

// поэтому его нельзя перекрыть

bar.call( window ); // 2

Давайте изучим как работает этот вариант. Мы создаем функцию bar(), которая внутри вручную вызывает foo.call(obj), таким образом принудительно вызывая foo с привязкой obj для this. Неважно как вы потом вызовете функцию bar, она всегда будет вручную вызывать foo с obj. Такая привязка одновременно явная и сильная, поэтому мы называем ее *жесткой привязкой*.

Самый типичный способ обернуть функцию с *жесткой привязкой* — создать сквозную обертку, передающую все параметры и возвращающую полученное значение:

function foo(something) {

console.log( this.a, something );

return this.a + something;

}

var obj = {

a: 2

};

var bar = function() {

return foo.apply( obj, arguments );

};

var b = bar( 3 ); // 2 3

console.log( b ); // 5

Еще один способ выразить этот шаблон — создать переиспользуемую вспомогательную функцию:

function foo(something) {

console.log( this.a, something );

return this.a + something;

}

// простая вспомогательная функция `bind`

function bind(fn, obj) {

return function() {

return fn.apply( obj, arguments );

};

}

var obj = {

a: 2

};

var bar = bind( foo, obj );

var b = bar( 3 ); // 2 3

console.log( b ); // 5

Поскольку *жесткая привязка* — очень распространеный шаблон, он есть как встроенный инструмент в ES5: Function.prototype.bind, а используется вот так:

function foo(something) {

console.log( this.a, something );

return this.a + something;

}

var obj = {

a: 2

};

var bar = foo.bind( obj );

var b = bar( 3 ); // 2 3

console.log( b ); // 5

bind(..) возвращает новую функцию, в которой жестко задан вызов оригинальной функции с именно тем контекстом this, который вы указываете.

**Примечание:** Начиная с ES6, в функции жесткой привязки, выдаваемой bind(..), есть свойство .name, наследуемое от исходной *функции*. Например: у bar = foo.bind(..) должно быть в bar.name значение "bound foo", которое является названием вызова функции, которое должно отражаться в стеке вызовов.

**"Контексты" в вызовах API**

Функции многих библиотек, и разумеется многие встроенные в язык JavaScript и во внешнее окружение функции, предоставляют необязательный параметр, обычно называемый "контекст", который спроектирован как обходной вариант для вас, чтобы не пользоваться bind(..), чтобы гарантировать, что ваша функция обратного вызова использует данный this.

Например:

function foo(el) {

console.log( el, this.id );

}

var obj = {

id: "awesome"

};

// используем `obj` как `this` для вызовов `foo(..)`

[1, 2, 3].forEach( foo, obj ); // 1 awesome 2 awesome 3 awesome

Внутренне эти различные функции почти наверняка используют *явную привязку* через call(..) или apply(..), избавляя вас от хлопот.

**Привязка new**

Четвертое и последнее правило привяки this потребует от нас переосмысления самого распространенного заблуждения о функциях и объектах в JavaScript.

В традиционных классо-ориентированных языках, "конструкторы" — это особые методы, связанные с классами, таким образом, что когда создается экземпляр класса с помощью операции new, вызывается конструктор этого класса. Обычно это выглядит как-то так:

something = new MyClass(..);

В JavaScript есть операция new и шаблон кода, который используется для этого, выглядит в основном идентично такой же операции в класс-ориентированных языках; многие разработчики полагают, что механизм JavaScript выполняет что-то похожее. Однако, на самом деле *нет никакой связи* с классо-ориентированной функциональностью у той, что предполагает использование new в JS.

Во-первых, давайте еще раз посмотрим что такое "конструктор" в JavaScript. В JS конструкторы — это **всего лишь функции**, которые, так уж получилось, были вызваны с операцией new перед ними. Они ни связаны с классами, ни создают экземпляров классов. Они — даже не особые типы функций. Они — всего лишь обычные функции, которые, по своей сути, "украдены" операцией new при их вызове.

Например, функция Number(..) действует как конструктор, цитируя спецификацию ES5.1:

15.7.2 Конструктор Number

Когда Number вызывается как часть выражения new, оно является конструктором: оно инициализирует только что созданный объект.

Так что, практически любая старенькая функция, включая встроенные объектные функции, такие как Number(..) (см. главу 3), могут вызываться с new перед ними и это превратит такой вызов функции в *вызов конструктора*. Это важное, но едва уловимое различие: нет такой вещи как "функции-конструкторы", а скорее есть вызовы, конструирующие *из* функций.

Когда функция вызывается с указанием перед ней new, также известный как вызов конструктора, автоматически выполняются следующие вещи:

1. Создается новенький объект (т.е. конструируется) прямо из воздуха
2. *Только что сконструированный объект связывается с [[Прототипом]]*
3. Только что сконструированный объект устанавливается как привязка this для этого вызова функции
4. За исключением тех случаев, когда функция возвращает свой собственный альтернативный **объект**, вызов функции с new *автоматически* вернет только что сконструированный объект.

Пункты 1, 3 и 4 применимы к нашему текущему обсуждению. Сейчас мы пропустим пункт 2 и вернемся к нему в главе 5.

Взглянем на такой код:

function foo(a) {

this.a = a;

}

var bar = new foo( 2 );

console.log( bar.a ); // 2

Вызывая foo(..) с new впереди нее, мы конструируем новый объект и устанавливаем этот новый объект как thisдля вызова foo(..). **Таким образом new — единственный путь, которым this при вызове функции может быть привязан.** Мы называем это *привязкой new*.

**Всё по порядку**

Итак, теперь мы раскрыли 4 правила привязки this в вызовах функций. *Всё*, что вам нужно сделать — это найти точку вызова и иссследовать ее, чтобы понять какое правило применяется. Но что если к точке вызова можно применить несколько соответствующих правил? Должен быть порядок очередности применения этих правил, а потому далее мы покажем в каком порядке применяются эти правила.

Думаю, совершенно ясно, что *привязка по умолчанию* имеет самый низкий приоритет из четырех. Поэтому мы отложим ее в сторону.

Что должно идти раньше: *неявная привязка* или *явная привязка*? Давайте проверим:

function foo() {

console.log( this.a );

}

var obj1 = {

a: 2,

foo: foo

};

var obj2 = {

a: 3,

foo: foo

};

obj1.foo(); // 2

obj2.foo(); // 3

obj1.foo.call( obj2 ); // 3

obj2.foo.call( obj1 ); // 2

Итак, *явная привязка* имеет приоритет над *неявной привязкой*, что означает, что вы должны спросить себя применима ли **сначала** *явная привязка* до проверки на *неявную привязку*.

Теперь, нам нужно всего лишь указать куда подходит по приоритету *привязка new*.

function foo(something) {

this.a = something;

}

var obj1 = {

foo: foo

};

var obj2 = {};

obj1.foo( 2 );

console.log( obj1.a ); // 2

obj1.foo.call( obj2, 3 );

console.log( obj2.a ); // 3

var bar = new obj1.foo( 4 );

console.log( obj1.a ); // 2

console.log( bar.a ); // 4

Хорошо, *привязка new* более приоритетна, чем *неявная привязка*. Но как вы думаете: *привязка new* более или менее приоритетна, чем *явная привязка*?

**Примечание:** new и call/apply не могут использоваться вместе, поэтому new foo.call(obj1) не корректно, чтобы сравнить напрямую *привязку new* с *явной привязкой*. Но мы все-таки можем использовать *жесткую привязку*, чтобы проверить приоритет этих двух правил.

До того, как мы начнем исследовать всё это на примере кода, постарайтесь вспомнить как физически работает *жесткая привязка*, которая есть в Function.prototype.bind(..), которая создает новую функцию-обертку, и в ней жестко задано игнорировать ее собственную привязку this (какой бы она ни была) и использовать указанную вручную нами.

По этой причине, кажется очевидным предполагать, что *жесткая привязка* (которая является формой *явной привязки*) более приоритетна, чем *привязка new*, а потому и не может быть перекрыта действием new.

Давайте проверим:

function foo(something) {

this.a = something;

}

var obj1 = {};

var bar = foo.bind( obj1 );

bar( 2 );

console.log( obj1.a ); // 2

var baz = new bar( 3 );

console.log( obj1.a ); // 2

console.log( baz.a ); // 3

Ого! bar жестко связан с obj1, но new bar(3) **не** меняет obj1.a на значение 3 что было бы ожидаемо нами. Вместо этого *жестко связанный* (с obj1) вызов bar(..) ***может*** быть перекрыт с new. Поскольку был применен new, обратно мы получили новый созданный объект, который мы назвали baz, и в результате видно, что в baz.a значение 3.

Это должно быть удивительно с учетом ранее рассмотренной "фальшивой" вспомогательной функции привязки:

function bind(fn, obj) {

return function() {

fn.apply( obj, arguments );

};

}

Если вы порассуждаете о том, как работает код этой вспомогательной функции, в нем нет способа для перекрытия жесткой привязки к obj операцией new как мы только что выяснили.

Но встроенная Function.prototype.bind(..) из ES5 — более сложная, даже очень на самом деле. Вот (немного отформатированный) полифиллинг кода, предоставленный со страницы MDN для функции bind(..):

if (!Function.prototype.bind) {

Function.prototype.bind = function(oThis) {

if (typeof this !== "function") {

// наиболее подходящая вещь в ECMAScript 5

// внутренняя функция IsCallable

throw new TypeError( "Function.prototype.bind - what " +

"is trying to be bound is not callable"

);

}

var aArgs = Array.prototype.slice.call( arguments, 1 ),

fToBind = this,

fNOP = function(){},

fBound = function(){

return fToBind.apply(

(

this instanceof fNOP &&

oThis ? this : oThis

),

aArgs.concat( Array.prototype.slice.call( arguments ) )

);

}

;

fNOP.prototype = this.prototype;

fBound.prototype = new fNOP();

return fBound;

};

}

**Примечание:** Полифиллинг bind(..), показанный выше, отличается от встроенной bind(..) в ES5, учитывающей функции жесткой привязки, которые используются с new (см. ниже почему это может быть полезно). Поскольку полифиллинг не может создавать функцию без .prototype так, как это делает встроенная утилита, есть едва уловимый окольный путь, чтобы приблизиться к такому же поведению. Двигайтесь осторожно, если планируете использовать new вместе с функцией жесткой привязки и полагаетесь на этот полифиллинг.

Часть, которая позволяет перекрыть new:

this instanceof fNOP &&

oThis ? this : oThis

// ... and:

fNOP.prototype = this.prototype;

fBound.prototype = new fNOP();

Мы не будем на самом деле углубляться в объяснения того, как работает эта хитрость (это сложно и выходит за рамки нашего обсуждения), но по сути утилита определяет была ли вызвана или нет функция жесткой привязки с new (в результате получая новый сконструированный объект в качестве ее this), и если так, то она использует *этот*свежесозданный this вместо ранее указанной *жесткой привязки* для this.

Почему перекрытие операцией new *жесткой привязки* может быть полезным?

Основная причина такого поведения — чтобы создать функцию (которую можно использовать вместе с new для конструирования объектов), которая фактически игнорирует *жесткую привязку* this, но которая инициализирует некоторые или все аргументы функции. Одной из возможностей bind(..) является умение сделать аргументы, переданные после после первого аргумента, привязки this, стандартными аргументами по умолчанию для предшествующей функции (технически называемое "частичным применением", которое является подмножеством "карринга").

Пример:

function foo(p1,p2) {

this.val = p1 + p2;

}

// используем здесь `null`, т.к. нам нет дела до

// жесткой привязки `this` в этом сценарии, и она

// будет переопределена вызовом с операцией `new` в любом случае!

var bar = foo.bind( null, "p1" );

var baz = new bar( "p2" );

baz.val; // p1p2

**Определяем this**

Теперь можно кратко сформулировать правила для определения this по точке вызова функции, в порядке их приоритета. Зададим вопросы в том же порядке и остановимся как только будет применено первое же правило.

1. Функция вызвана с new (**привязка new**)? Раз так, то this — новый сконструированный объект.

var bar = new foo()

1. Функция вызвана с call или apply (**явная привязка**), даже скрыто внутри *жесткой привязки* в bind? Раз так, this — явно указанный объект.

var bar = foo.call( obj2 )

1. Функция вызвана с контекстом (**неявная привязка**), иначе называемым как владеющий или содержащий объект? Раз так, this является *тем самым* объектом контекста.

var bar = obj1.foo()

1. В противном случае, будет this по умолчанию (**привязка по умолчанию**). В режиме strict mode, это будет undefined, иначе будет объект global.

var bar = foo()

Вот и всё. Вот *всё, что нужно*, чтобы понимать правила привязки this для обычных вызовов функций. Ну... почти.

**Исключения привязок**

Как обычно, из "правил" есть несколько *исключений*.

Поведение привязки this в некоторых сценариях может быт неожиданным, там где вы подразумеваете одну привязку, а получаете в итоге поведение привязки по правилу *привязки по умолчанию* (см. ранее).

**Проигнорированный this**

Если вы передаете null или undefined в качестве параметра привязки this в call, apply или bind, то эти значения фактически игнорируются, а взамен к вызову применяется правило *привязки по умолчанию*.

function foo() {

console.log( this.a );

}

var a = 2;

foo.call( null ); // 2

Зачем вам бы понадобилось намеренно передавать что-то подобное null в качестве привязки this?

Довольно распространено использовать apply(..) для распаковки массива значений в качестве параметров вызова функции. Аналогично и bind(..) может каррировать параметры (предварительно заданные значения), что может быть очень полезно.

function foo(a,b) {

console.log( "a:" + a + ", b:" + b );

}

// распакуем массив как параметры

foo.apply( null, [2, 3] ); // a:2, b:3

// каррируем с помощью `bind(..)`

var bar = foo.bind( null, 2 );

bar( 3 ); // a:2, b:3

Обе этих инструмента требуют указания привязки this в качестве первого параметра. Если рассматриваемым функциям не важен this, то вам нужно -значение-заменитель, и null — это похоже разумный выбор, как мы видели выше.

**Примечание:** В этой книге мы не уделим этому внимания, но в ES6 есть операция расширения ..., которая дает возможности синтаксически "развернуть" массив как параметры без необходимости использования apply(..), например как в foo(...[1,2]), что равносильно foo(1,2) — синтаксически избегая привязки this, раз она не нужна. К сожалению, в ES6 нет синтаксической замены каррингу, поэтому параметр this вызова bind(..) все еще требует внимания.

Однако, есть некоторая скрытая "опасность" в том, чтобы всегда использовать null, когда вам не нужна привязка this. Если вы когда-нибудь воспользуетесь этим при вызове функции (например, функции сторонней библиотеки, которой вы не управляете) и эта функция *все-таки* воспользуется ссылкой на this, сработает правило *привязки по умолчанию*, что повлечет за собой ненамеренно ссылку (или еще хуже, мутацию!) на объект global (window в браузере).

Очевидно, что такая ловушка может привести к ряду *очень трудно* диагностируемых/отслеживаемых ошибок.

**Более безопасный this**

Пожалуй в некоторой степени "более безопасная" практика — передавать особым образом настроенный объект для this, который гарантирует отсутствие побочных эффектов в вашей программе. Заимствуя терминологию из сетевых (и военных) технологий, мы можем создать объект "DMZ" (демилитаризованной зоны (de-militarized zone)) — не более чем полностью пустой, неделегированный (см. главы 5 и 6) объект.

Если всегда передавать DMZ-объект для привязок this, которые не требуются, то мы можем быть уверены в том, что любое скрытое/неожидаемое использование this будет ограничено пустым объектом, который защитит объект global нашей программы от побочных эффектов.

Поскольку этот объект совершенно пустой, лично я люблю давать его переменной имя ø (математический символ пустого множества в нижнем регистре). На многих клавиатурах (как например US-раскладка на Mac), этот символ легко можно ввести с помощью ⌥+o (option+o). В некоторых системах есть возможность назначать горячие клавиши на определенные символы. Если вам не нравится символ ø или на вашей клавиатуре сложно набрать такой символ, вы конечно же можете назвать переменную как вам угодно.

Как бы вы ни назвали ее, самый простой путь получить **абсолютно пустой** объект — это Object.create(null) (см. главу 5). Object.create(null) — похож на { }, но без передачи Object.prototype, поэтому он "более пустой", чем просто { }.

function foo(a,b) {

console.log( "a:" + a + ", b:" + b );

}

// наш пустой DMZ-объект

var ø = Object.create( null );

// распаковываем массив как параметры

foo.apply( ø, [2, 3] ); // a:2, b:3

// каррируем с помощью `bind(..)`

var bar = foo.bind( ø, 2 );

bar( 3 ); // a:2, b:3

Не только функционально "безопаснее", но еще и стилистически выгоднее использовать ø, что семантически отражает желаение "Я хочу, чтобы this был пустым" немного точнее, чем null. Но опять таки, называйте свой DMZ-объект как хотите.

**Косвенность**

Еще одной вещью, которую нужно опасаться, является создание (намеренно или нет) "косвенных ссылок" на функции, и в этих случаях, когда такая ссылка на функцию вызывается, то также применяется правило *привязки по умолчанию*.

Самый распространенный путь появления *косвенных ссылок* — при присваивании:

function foo() {

console.log( this.a );

}

var a = 2;

var o = { a: 3, foo: foo };

var p = { a: 4 };

o.foo(); // 3

(p.foo = o.foo)(); // 2

*Результатом* выражения присваивания p.foo = o.foo будет всего лишь ссылка на внутренний объект функции. В силу этого, настоящая точка вызова - это просто foo(), а не p.foo() или o.foo() как вы могли бы предположить. Согласно вышеприведенным правилам будет применено правило *привязки по умолчанию*.

Напоминание: независимо от того как вы добрались до вызова функции используя правило *привязки по умолчанию*, статус **содержимого** вызванной функции в режиме strict mode, использующего ссылку на this, а не точка вызова функции, определяет значение *привязки по умолчанию*: либо объект global если не в strict mode или undefined в strict mode.

**Смягчение привязки**

Ранее мы отметили, что *жесткая привязка* была одной из стратегий для предотвращения случайного действия правила *привязки по умолчанию* при вызове функции, заставив ее привязаться к указанному this (до тех пор, пока вы не используете new, чтобы переопределить это поведение!). Проблема в том, что *жесткая приязка* значительно уменьшает гибкость функции, не давая указывать this вручную, чтобы перекрыть *неявную привязку* или даже последующие попытки *явной привязки*.

Было бы неплохо, если бы был путь указать другое умолчание для *привязки по умолчанию* (не global или undefined), но при этом оставив возможность для функции вручную привязать this через технику *неявной* или *явной* привязки.

Можно собрать инструмент так называемой *мягкой привязки*, который эмулирует желаемое поведение.

if (!Function.prototype.softBind) {

Function.prototype.softBind = function(obj) {

var fn = this,

curried = [].slice.call( arguments, 1 ),

bound = function bound() {

return fn.apply(

(!this ||

(typeof window !== "undefined" &&

this === window) ||

(typeof global !== "undefined" &&

this === global)

) ? obj : this,

curried.concat.apply( curried, arguments )

);

};

bound.prototype = Object.create( fn.prototype );

return bound;

};

}

Инструмент softBind(..), представленный здесь, работает подобно встроенному в ES5 инструменту bind(..), за исключением нашего поведения *мягкой привязки*. Он делает обертку указанной функции с логикой, которая проверяет this в момент вызова и если это global или undefined, использует указанное заданее альтернативное *умолчание* (obj). В противном случае this остается как есть. Также этот инструмент дает возможность опционального карринга (см. ранее обсуждениеbind(..)).

Продемонстрируем его в действии:

function foo() {

console.log("name: " + this.name);

}

var obj = { name: "obj" },

obj2 = { name: "obj2" },

obj3 = { name: "obj3" };

var fooOBJ = foo.softBind( obj );

fooOBJ(); // name: obj

obj2.foo = foo.softBind(obj);

obj2.foo(); // name: obj2 <---- смотрите!!!

fooOBJ.call( obj3 ); // name: obj3 <---- смотрите!

setTimeout( obj2.foo, 10 ); // name: obj <---- возврат к мягкой привяке

Для мягкопривязанной версии функции foo() можно вручную привязать this к obj2 или obj3 как показано выше, но он возвращается к obj в случае применения *привязки по умолчанию*.

**Лексический this**

В обычных функциях строго соблюдаются 4 правила, которые мы только что рассмотрели. Но в ES6 представлен особый вид функции, которая не использует эти правила: стрелочная функция.

Стрелочные функции обозначаются не ключевым словом function, а операцией =>, так называемой "жирной стрелкой". Вместо использования четырех стандартных this-правил, стрелочные функции заимствуют привязку thisиз окружающей (функции или глобальной) области видимости.

Проиллюстрируем лексическую область видимости стрелочной функции:

function foo() {

// возвращаем стрелочную функцию

return (a) => {

// Здесь `this` лексически заимствован из `foo()`

console.log( this.a );

};

}

var obj1 = {

a: 2

};

var obj2 = {

a: 3

};

var bar = foo.call( obj1 );

bar.call( obj2 ); // 2, а не 3!

Стрелочная функция, созданная в foo(), лексически захватывает любой this в foo() во время ее вызова. Поскольку в foo() this был привязан к obj1, bar (ссылка на возвращаемую стрелочную функцию) также будет с привязкой this к obj1. Лексическая привязка стрелочной функции не может быть перекрыта (даже с помощью new!).

Самый распространенный вариант использования стрелочной функции — обычно при использовании функций обратного вызова, таких как обработчики событий или таймеры:

function foo() {

setTimeout(() => {

// Здесь `this` лексически заимствован из `foo()`

console.log( this.a );

},100);

}

var obj = {

a: 2

};

foo.call( obj ); // 2

Несмотря на то, что стрелочные функции предоставляют альтернативу применению bind(..) к функции, чтобы гарантировать определенный this, что может выглядеть весьма привлекательно, важно отметить, что они фактически запрещают традиционный механизм this в пользу более понятной лексической области видимости. До ES6, у нас уже был довольно распространенный шаблон для выполнения такой задачи, который по сути почти неотличим от сущности стрелочных функций ES6:

function foo() {

var self = this; // лексический захват `this`

setTimeout( function(){

console.log( self.a );

}, 100 );

}

var obj = {

a: 2

};

foo.call( obj ); // 2

В том время как self = this и стрелочные функции обе кажутся хорошим "решением" при нежелании использовать bind(..), они фактически убегают от this вместо того, чтобы понять и научиться использовать его.

Если вы застали себя пишущим код в стиле this, но большую часть или всё время вы сводите на нет механизм this с помощью трюков лексической конструкции self = this или стрелочной функции, возможно вам следует сделать что-то одно из этого:

1. Использовать только лексическую область видимости и забыть о фальшивости кода в стиле this.
2. Полностью научиться использовать механизмы this-стиля, включая применение bind(..), где необходимо, и попытаться избегать трюков "лексического this" с помощью self = this и стрелочной функции.

Программа может эффективно использовать оба стиля кодирования (лексический и this), но внутри одной и той же функции и, разумеется, при одних и тех же видах поисков переменных, смешивание двух этих механизмов обычно приводит к менее обслуживаемому коду, и возможно будет слишком перегруженным, чтобы выглядеть умным.

**Обзор**

Определение привязки this для вызова функции требует поиска непосредственной точки вызова этой функции. Как уже выяснилось, к точке вызова могут быть применены четыре правила, в *именно таком* порядке приоритета:

1. Вызвана с new? Используем только что созданный объект.
2. Вызвана с помощью call или apply (или bind)? Используем указанный объект.
3. Вызвана с объектом контекста, владеющего вызовом функции? Используем этот объект контекста.
4. По умолчанию: undefined в режиме strict mode, в противном случае объект global.

Остерегайтесь случайного/неумышленного вызова с применением правила *привязки по умолчанию*. В случаях, когда вам нужно "безопасно" игнорировать привязку this, "DMZ"-объект, подобный ø = Object.create(null), — хорошая замена, защищающая объект global от непредусмотренных побочных эффектов.

Вместо четырех стандартных правил привязки стрелочные функции ES6 используют лексическую область видимости для привязки this, что означает, что они заимствуют привязку this (какой бы она ни была) от вызова своей окружающей функции. Они по существу являются синтаксической заменой self = this в до-ES6 коде.