

Часть 1

Язык JavaScript

JS

Илья Кантор

2015

Язык JavaScript

Сборка от 17 марта 2017 г.

Последняя версия учебника находится на сайте learn.javascript.ru.

Мы постоянно работаем над улучшением учебника. При обнаружении ошибок пишите о них на [нашем баг-трекере](#) ☞.

- Введение
 - Введение в JavaScript
 - Справочники и спецификации
 - Редакторы для кода
 - Консоль разработчика
- Основы JavaScript
 - Привет, мир!
 - Внешние скрипты, порядок исполнения
 - Структура кода
 - Современный стандарт, "use strict"
 - Переменные
 - Правильный выбор имени переменной
 - Шесть типов данных, typeof
 - Основные операторы
 - Операторы сравнения и логические значения
 - Побитовые операторы
 - Взаимодействие с пользователем: alert, prompt, confirm
 - Условные операторы: if, '?'
 - Логические операторы
 - Преобразование типов для примитивов
 - Циклы while, for
 - Конструкция switch
 - Функции
 - Функциональные выражения
 - Рекурсия, стек
 - Именованные функциональные выражения
 - Всё вместе: особенности JavaScript
- Качество кода
 - Отладка в браузере Chrome
 - Советы по стилю кода
 - Как писать неподдерживаемый код?
 - Автоматические тесты при помощи chai и mocha
- Структуры данных
 - Введение в методы и свойства
 - Числа
 - Строки
 - Объекты как ассоциативные массивы
 - Объекты: перебор свойств
 - Объекты: передача по ссылке
 - Массивы с числовыми индексами
 - Массивы: методы
 - Массив: перебирающие методы
 - Псевдомассив аргументов "arguments"
 - Дата и Время
- Замыкания, область видимости
 - Глобальный объект
 - Замыкания, функции изнутри
 - [[Scope]] для new Function
 - Локальные переменные для объекта
 - Модули через замыкания
 - Управление памятью в JavaScript
 - Устаревшая конструкция "with"
- Методы объектов и контекст вызова
 - Методы объектов, this
 - Преобразование объектов: toString и valueOf

- Создание объектов через "new"
- Дескрипторы, геттеры и сеттеры свойств
- Статические и фабричные методы
- Явное указание this: "call", "apply"
- Привязка контекста и карринг: "bind"
- Функции-обёртки, декораторы
- Некоторые другие возможности
 - Типы данных: [[Class]], instanceof и утки
 - Формат JSON, метод toJSON
 - setTimeout и setInterval
 - Запуск кода из строки: eval
 - Перехват ошибок, "try..catch"
- ООП в функциональном стиле
 - Введение
 - Внутренний и внешний интерфейс
 - Геттеры и сеттеры
 - Функциональное наследование
- ООП в прототипном стиле
 - Прототип объекта
 - Свойство F.prototype и создание объектов через new
 - Встроенные "классы" в JavaScript
 - Свои классы на прототипах
 - Наследование классов в JavaScript
 - Проверка класса: "instanceof"
 - Свои ошибки, наследование от Error
 - Примеси
- Современные возможности ES-2015
 - ES-2015 сейчас
 - Переменные: let и const
 - Деструктуризация
 - Функции
 - Строки
 - Объекты и прототипы
 - Классы
 - Тип данных Symbol
 - Итераторы
 - Set, Map, WeakSet и WeakMap
 - Promise
 - Генераторы
 - Модули
 - Проху

Эта часть позволит вам изучить JavaScript с нуля или упорядочить и дополнить существующие знания.

Мы будем использовать браузер в качестве окружения, но основное внимание будет уделяться именно самому языку JavaScript.

Введение

Про язык JavaScript и окружение для разработки на нём.

Введение в JavaScript

Давайте посмотрим, что такого особенного в JavaScript, почему именно он, и какие еще технологии существуют, кроме JavaScript.

Что такое JavaScript?

JavaScript изначально создавался для того, чтобы сделать web-странички «живыми». Программы на этом языке называются *скриптами*. В браузере они подключаются напрямую к HTML и, как только загружается страничка – тут же выполняются.

Программы на JavaScript – обычный текст. Они не требуют какой-то специальной подготовки.

В этом плане JavaScript сильно отличается от другого языка, который называется [Java](#).

Почему JavaScript?

Когда создавался язык JavaScript, у него изначально было другое название: «LiveScript». Но тогда был очень популярен язык Java, и маркетологи решили, что схожее название сделает новый язык более популярным.

Планировалось, что JavaScript будет эдаким «младшим братом» Java. Однако, история распорядилась по-своему, JavaScript сильно вырос, и сейчас это совершенно независимый язык, со своей спецификацией, которая называется [ECMAScript](#), и к Java не имеет никакого отношения.

У него много особенностей, которые усложняют освоение, но по ходу учебника мы с ними разберёмся.

JavaScript может выполняться не только в браузере, а где угодно, нужна лишь специальная программа – [интерпретатор](#). Процесс выполнения скрипта называют «интерпретацией».

Компиляция и интерпретация, для программистов

Для выполнения программ, не важно на каком языке, существуют два способа: «компиляция» и «интерпретация».

- *Компиляция* – это когда исходный код программы, при помощи специального инструмента, другой программы, которая называется «компилятор», преобразуется в другой язык, как правило – в машинный код. Этот машинный код затем распространяется и запускается. При этом исходный код программы остаётся у разработчика.
- *Интерпретация* – это когда исходный код программы получает другой инструмент, который называют «интерпретатор», и выполняет его «как есть». При этом распространяется именно сам исходный код (скрипт). Этот подход применяется в браузерах для JavaScript.

Современные интерпретаторы перед выполнением преобразуют JavaScript в машинный код или близко к нему, оптимизируют, а уже затем выполняют. И даже во время выполнения стараются оптимизировать. Поэтому JavaScript работает очень быстро.

Во все основные браузеры встроен интерпретатор JavaScript, именно поэтому они могут выполнять скрипты на странице. Но, разумеется, JavaScript можно использовать не только в браузере. Это полноценный язык, программы на котором можно запускать и на сервере, и даже в стиральной машинке, если в ней установлен соответствующий интерпретатор.

Поговорим о браузерах

Далее в этой главе мы говорим о возможностях и ограничениях JavaScript именно в контексте браузера.

Что умеет JavaScript?

Современный JavaScript – это «безопасный» язык программирования общего назначения. Он не предоставляет низкоуровневых средств работы с памятью, процессором, так как изначально был ориентирован на браузеры, в которых это не требуется.

Что же касается остальных возможностей – они зависят от окружения, в котором запущен JavaScript. В браузере JavaScript умеет делать всё, что относится к манипуляции со страницей, взаимодействию с посетителем и, в какой-то мере, с сервером:

- Создавать новые HTML-теги, удалять существующие, менять стили элементов, прятать, показывать элементы и т.п.
- Реагировать на действия посетителя, обрабатывать клики мыши, перемещения курсора, нажатия на клавиатуру и т.п.
- Посылать запросы на сервер и загружать данные без перезагрузки страницы (эта технология называется "AJAX").
- Получать и устанавливать cookie, запрашивать данные, выводить сообщения...
- ...и многое, многое другое!

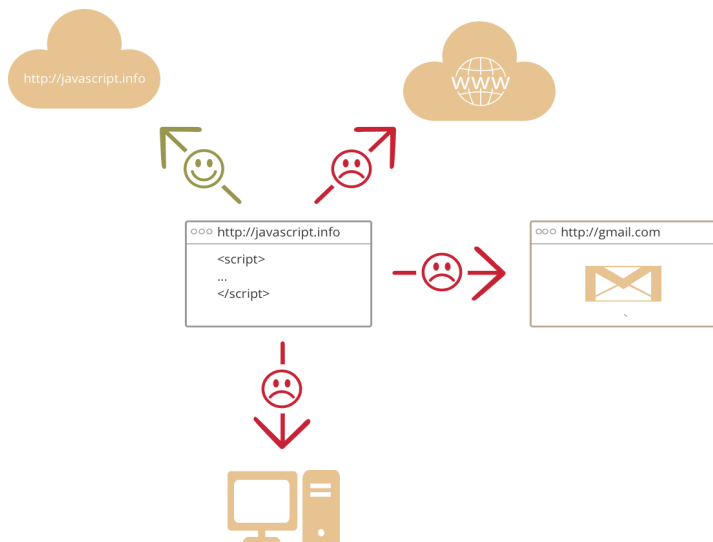
Что НЕ умеет JavaScript?

JavaScript – быстрый и мощный язык, но браузер накладывает на его исполнение некоторые ограничения...

Это сделано для безопасности пользователей, чтобы злоумышленник не мог с помощью JavaScript получить личные данные или как-то навредить компьютеру пользователя.

Этих ограничений нет там, где JavaScript используется вне браузера, например на сервере. Кроме того, современные браузеры предоставляют свои механизмы по установке плагинов и расширений, которые обладают расширенными возможностями, но требуют специальных действий по установке от пользователя

Большинство возможностей JavaScript в браузере ограничено текущим окном и страницей.



- JavaScript не может читать/записывать произвольные файлы на жесткий диск, копировать их или вызывать программы. Он не имеет прямого доступа к операционной системе.

Современные браузеры могут работать с файлами, но эта возможность ограничена специально выделенной директорией – «песочницей». Возможности по доступу к устройствам также прорабатываются в современных стандартах и частично доступны в некоторых браузерах.

- JavaScript, работающий в одной вкладке, не может общаться с другими вкладками и окнами, за исключением случая, когда он сам открыл это окно или несколько вкладок из одного источника (одинаковый домен, порт, протокол).

Есть способы это обойти, и они раскрыты в учебнике, но они требуют специального кода на оба документа, которые находятся в разных вкладках или окнах. Без него, из соображений безопасности, залезть из одной вкладки в другую при помощи JavaScript нельзя.

- Из JavaScript можно легко посылать запросы на сервер, с которого пришла страница. Запрос на другой домен тоже возможен, но менее удобен, т. к. и здесь есть ограничения безопасности.

В чём уникальность JavaScript?

Есть как минимум *три* замечательных особенности JavaScript:

- Полная интеграция с HTML/CSS.
- Простые вещи делаются просто.
- Поддерживается всеми распространёнными браузерами и включён по умолчанию.

Этих трёх вещей одновременно нет больше ни в одной браузерной технологии.

Поэтому JavaScript и является самым распространённым средством создания браузерных интерфейсов.

Тенденции развития

Перед тем, как вы планируете изучить новую технологию, полезно ознакомиться с её развитием и перспективами. Здесь в JavaScript всё более чем хорошо.

HTML 5

HTML 5 – эволюция стандарта HTML, добавляющая новые теги и, что более важно, ряд новых возможностей браузерам.

Вот несколько примеров:

- Чтение/запись файлов на диск (в специальной «песочнице», то есть не любые).
- Встроенная в браузер база данных, которая позволяет хранить данные на компьютере пользователя.
- Многозадачность с одновременным использованием нескольких ядер процессора.
- Проигрывание видео/аудио, без Flash.
- 2D и 3D-рисование с аппаратной поддержкой, как в современных играх.

Многие возможности HTML5 всё ещё в разработке, но браузеры постепенно начинают их поддерживать.

Тенденция: JavaScript становится всё более и более мощным и возможности браузера растут в сторону десктопных приложений.

ECMAScript 6

Сам язык JavaScript улучшается. Современный стандарт ECMAScript 5 включает в себя новые возможности для разработки, ECMAScript 6 будет шагом вперёд в улучшении синтаксиса языка.

Современные браузеры улучшают свои движки, чтобы увеличить скорость исполнения JavaScript, исправляют баги и стараются следовать стандартам.

Тенденция: JavaScript становится всё быстрее и стабильнее, в язык добавляются новые возможности.

Очень важно то, что новые стандарты HTML5 и ECMAScript сохраняют максимальную совместимость с предыдущими версиями. Это позволяет избежать неприятностей с уже существующими приложениями.

Впрочем, небольшая проблема с «супер-современными штучками» всё же есть. Иногда браузеры стараются включить новые возможности, которые ещё не полностью описаны в стандарте, но настолько интересны, что разработчики просто не могут ждать.

...Однако, со временем стандарт меняется и браузерам приходится подстраиваться к нему, что может привести к ошибкам в уже написанном, основанном на старой реализации, JavaScript-коде. Поэтому следует дважды подумать перед тем, как применять на практике такие «супер-новые» решения.

При этом все браузеры сходятся к стандарту, и различий между ними уже гораздо меньше, чем всего лишь несколько лет назад.

Тенденция: всё идет к полной совместимости со стандартом.

Альтернативные браузерные технологии

Вместе с JavaScript на страницах используются и другие технологии. Связка с ними может помочь достигнуть более интересных результатов в тех местах, где браузерный JavaScript пока не столь хорош, как хотелось бы.

Java

Java – язык общего назначения, на нём можно писать самые разные программы. Для интернет-страниц есть особая возможность – написание *апплетов*.

Апплет – это программа на языке Java, которую можно подключить к HTML при помощи тега `applet`, выглядит это примерно так:

```
<applet code="BTApplet.class" codebase="/files/tutorial/intro/alt/">
  <param name="nodes" value="50,30,70,20,40,60,80,35,65,75,85,90">
  <param name="root" value="50">
</applet>
```

Такой тег загружает Java-программу из файла `BTApplet.class` и выполняет её с параметрами `param`. Апплет выполняется в отдельной части страницы, в прямоугольном «контейнере». Все действия пользователя внутри него обрабатывает апплет. Контейнер, впрочем, может быть и скрыт, если апплету нечего показывать.

Конечно, для этого на компьютере должна быть установлена и включена среда выполнения Java, включая браузерный плагин. Кроме того, апплет должен быть подписан сертификатом издателя (в примере выше апплет без подписи), иначе Java заблокирует его.

Чем нам, JavaScript-разработчикам, может быть интересен Java?

В первую очередь тем, что подписанный Java-апплет может всё то же, что и обычная программа, установленная на компьютере посетителя. Конечно, для этого понадобится согласие пользователя при открытии такого апплета.

Достоинства

- Java может делать всё от имени посетителя, совсем как установленная программа. Потенциально опасные действия требуют подписанного апплета и согласия пользователя.

Недостатки

- Java требует больше времени для загрузки.
- Среда выполнения Java, включая браузерный плагин, должна быть установлена на компьютере посетителя и включена.
- Java-апплет не интегрирован с HTML-страницей, а выполняется отдельно. Но он может вызывать функции JavaScript.

Плагины и расширения для браузера

Все современные браузеры предоставляют возможность написать плагины. Для этого можно использовать как JavaScript (Chrome, Opera, Firefox), так и язык C (ActiveX для Internet Explorer).

Эти плагины могут как отображать содержимое специального формата (плагин для проигрывания музыки, для показа PDF), так и взаимодействовать со страницей.

Как и в ситуации с Java-апплетом, у них широкие возможности, но посетитель поставит их в том случае, если вам доверяет.

Adobe Flash

Adobe Flash – кросс-браузерная платформа для мультимедиа-приложений, анимаций, аудио и видео.

Flash-полики – это скомпилированная программа, написанная на языке ActionScript. Её можно подключить к HTML-странице и запустить в прямоугольном контейнере.

В первую очередь Flash полезен тем, что позволяет кросс-браузерно работать с микрофоном, камерой, с буфером обмена, а также поддерживает продвинутые возможности по работе с сетевыми соединениями.

Достоинства

- Сокеты, UDP для P2P и другие продвинутые возможности по работе с сетевыми соединениями
- Поддержка мультимедиа: изображения, аудио, видео. Работа с веб-камерой и микрофоном.

Недостатки

- Flash должен быть установлен и включён. А на некоторых устройствах он вообще не поддерживается.
- Flash не интегрирован с HTML-страницей, а выполняется отдельно.
- Существуют ограничения безопасности, однако они немного другие, чем в JavaScript.

Из Flash можно вызывать JavaScript и наоборот, поэтому обычно сайты используют JavaScript, а там, где он не справляется – можно подумать о Flash.

Языки поверх JavaScript

Синтаксис JavaScript устраивает не всех: одним он кажется слишком свободным, другим – наоборот, слишком ограниченным, третьи хотят добавить в язык дополнительные возможности, которых нет в стандарте...

Это нормально, ведь требования и проекты у всех разные.

В последние годы появилось много языков, которые добавляют различные возможности «поверх» JavaScript, а для запуска в браузере – при помощи специальных инструментов «трансляторов» превращаются в обычный JavaScript-код.

Это преобразование происходит автоматически и совершенно прозрачно, при этом неудобств в разработке и отладке практически нет.

При этом разные языки выглядят по-разному и добавляют совершенно разные вещи:

- Язык [CoffeeScript](#) – это «синтаксический сахар» поверх JavaScript. Он сосредоточен на большей ясности и краткости кода. Как правило, его особенно любят программисты на Ruby.
- Язык [TypeScript](#) сосредоточен на добавлении строгой типизации данных. Он предназначен для упрощения разработки и поддержки больших систем. Его разрабатывает Microsoft.
- Язык [Dart](#) интересен тем, что он не только транслируется в JavaScript, как и другие языки, но и имеет свою независимую среду выполнения, которая даёт ему ряд возможностей и доступна для встраивания в приложения (вне браузера). Он разрабатывается компанией Google.

ES6 и ES7 прямо сейчас

Существуют также трансляторы, которые берут код, использующий возможности будущих стандартов JavaScript, и преобразуют его в более старый вариант, который понимают все браузеры.

Например, [babeljs](#).

Благодаря этому, мы можем использовать многие возможности будущего уже сегодня.

Итого

Язык JavaScript уникален благодаря своей полной интеграции с HTML/CSS. Он работает почти у всех посетителей.

...Но хороший JavaScript-программист не должен забывать и о других технологиях.

Ведь наша цель – создание хороших приложений, и здесь Flash, Java и браузерные расширения имеют свои уникальные возможности, которые можно использовать вместе с JavaScript.

Что же касается CoffeeScript, TypeScript и других языков, построенных над JavaScript – они могут быть очень полезны. Рекомендуется посмотреть их, хотя бы в общих чертах, но, конечно, уже после освоения самого JavaScript.

Справочники и спецификации

В этом разделе мы познакомимся со справочниками и спецификациями.

Если вы только начинаете изучение, то вряд ли они будут нужны прямо сейчас. Тем не менее, эта глава находится в начале, так как предсказать точный момент, когда вы захотите заглянуть в справочник – невозможно, но точно известно, что этот момент настанет.

Поэтому рекомендуется кратко взглянуть на них и взять на заметку, чтобы при необходимости вернуться к ним в будущем.

Справочники, и как в них искать

Самая полная и подробная информация по JavaScript и браузерам есть в справочниках.

Её объём таков, что перевести всё с английского невозможно. Даже сделать «единый полный справочник» не получается, так как изменений много и они происходят постоянно.

Тем не менее, жить вполне можно если знать, куда смотреть.

Есть три основных справочника по JavaScript на английском языке:

1. [Mozilla Developer Network](#) – содержит информацию, верную для основных браузеров. Также там присутствуют расширения только для Firefox (они помечены).
Когда мне нужно быстро найти «стандартную» информацию по RegExp – ввожу в Google «RegExp MDN», и ключевое слово «MDN» (Mozilla Developer Network) приводит к информации из этого справочника.
2. [MSDN](#) – справочник от Microsoft. Там много информации, в том числе и по JavaScript (они называют его «JScript»). Если нужно что-то, специфичное для IE – лучше лезть сразу туда.
Например, для информации об особенностях RegExp в IE – полезное сочетание: «RegExp msdn». Иногда к поисковой фразе лучше добавить термин «JScript»: «RegExp msdn jscript».
3. [Safari Developer Library](#) – менее известен и используется реже, но в нём тоже можно найти ценную информацию.

Есть ещё справочники, не от разработчиков браузеров, но тоже хорошие:

1. <http://help.dottoro.com> – содержит подробную информацию по HTML/CSS/JavaScript.
2. <http://javascript.ru/manual> – справочник по JavaScript на русском языке, он содержит основную информацию по языку, без функций для работы с документом. К нему можно обращаться и по адресу, если знаете, что искать. Например, так: <http://javascript.ru/RegExp>.
3. <http://www.quirksmode.org> – информация о браузерных несовместимостях. Этот ресурс сам по себе довольно старый и, в первую очередь, полезен для поддержки устаревших браузеров. Для поиска можно пользоваться комбинацией «quirksmode onkeypress» в Google.
4. <http://caniuse.com> – ресурс о поддержке браузерами новейших возможностей HTML/CSS/JavaScript. Например, для поддержки функций криптографии: <http://caniuse.com/#feat=cryptography>.
5. <https://kangax.github.io/compat-table> – таблица с обзором поддержки спецификации ECMAScript различными платформами.

Спецификации

Спецификация – это самый главный, определяющий документ, в котором написано, как себя ведёт JavaScript, браузер, CSS и т.п.

Если что-то непонятно, и справочник не даёт ответ, то спецификация, как правило, раскрывает тему гораздо глубже и позволяет расставить точки над i.

Спецификация ECMAScript

Спецификация (формальное описание синтаксиса, базовых объектов и алгоритмов) языка Javascript называется [ECMAScript](#).

Её перевод есть на сайте в разделе [стандарт языка](#).

Почему не просто "JavaScript" ?

Вы можете спросить: «Почему спецификация для JavaScript не называется просто «JavaScript», зачем существует какое-то отдельное название?»

Всё потому, что JavaScript™ – зарегистрированная торговая марка, принадлежащая корпорации Oracle.

Название «ECMAScript» было выбрано, чтобы сохранить спецификацию независимой от владельцев торговой марки.

Спецификация может рассказать многое о том, как работает язык, и она является самым фундаментальным, доверенным источником информации.

Спецификации HTML/DOM/CSS

JavaScript – язык общего назначения, поэтому в спецификации ECMAScript нет ни слова о браузерах.

Главная организация, которая занимается HTML, CSS, XML и множеством других стандартов – [Консорциум Всемирной паутины](#) (World Wide Consortium, сокращённо W3C).

Информацию о них можно найти на сайте w3.org. К сожалению, найти в этой куче то, что нужно, может быть нелегко, особенно когда неизвестно в каком именно стандарте искать. Самый лучший способ – попросить Google с указанием сайта.

Например, для поиска `document.cookie` набрать `document.cookie site:w3.org`.

Последние версии стандартов расположены на домене dev.w3.org.

Кроме того, в том, что касается HTML5 и DOM/CSS, W3C активно использует наработки другой организации – [WhatWG](#). Поэтому самые актуальные версии спецификаций по этим темам обычно находятся на <https://whatwg.org/specs/>.

Иногда бывает так, что информация на сайте <http://dev.w3.org> отличается от <http://whatwg.org>. В этом случае, как правило, следует руководствоваться <http://whatwg.org>.

Итого

Итак, посмотрим какие у нас есть источники информации.

Справочники:

- [Mozilla Developer Network](#) – информация для Firefox и большинства браузеров. Google-комбо: "RegExp MDN", ключевое слово «MDN».
- [MSDN](#) – информация по IE. Google-комбо: "RegExp msdn". Иногда лучше добавить термин «JScript»: "RegExp msdn jscript".
- [Safari Developer Library](#) – информация по Safari.
- <http://help.dottoro.com> – подробная информация по HTML/CSS/JavaScript с учётом браузерной совместимости. Google-комбо: "RegExp dottoro".
- <http://javascript.ru/manual> – справочник по JavaScript на русском языке. К нему можно обращаться и по адресу, если знаете, что искать. Например, так: <http://javascript.ru/RegExp>. Google-комбо: "RegExp site:javascript.ru".

Спецификации содержат важнейшую информацию о том, как оно «должно работать»:

- JavaScript, современный стандарт [ES5 \(англ\)](#) [↗](#), и предыдущий [ES3 \(рус\)](#).
- HTML/DOM/CSS – на сайте <http://w3.org> [↗](#). Google-комбо: "document.cookie site:w3.org".
- ...А самые последние версии стандартов – на <http://dev.w3.org> [↗](#) и на <http://whatwg.org/specs/> [↗](#).

То, как оно на самом деле работает и несовместимости:

- <http://quirksmode.org/> [↗](#). Google-комбо: "innerHeight quirksmode".

Поддержка современных и новейших возможностей браузерами:

- <http://caniuse.com> [↗](#). Google-комбо: "caniuse geolocation".

Редакторы для кода

Для разработки обязательно нужен хороший редактор.

Выбранный вами редактор должен иметь в своем арсенале:

1. Подсветку синтаксиса.
2. Автодополнение.
3. «Фолдинг» (от англ. folding) – возможность скрыть-раскрыть блок кода.

IDE

Термин IDE (Integrated Development Environment) – «интегрированная среда разработки», означает редактор, который расширен большим количеством «наворотов», умеет работать со вспомогательными системами, такими как багтрекер, контроль версий, и много чего ещё.

Как правило, IDE загружает весь проект целиком, поэтому может предоставлять автодополнение по функциям всего проекта, удобную навигацию по его файлам и т.п.

Если вы ещё не задумывались над выбором IDE, посмотрите к следующим вариантам.

- Продукты IntelliJ: [WebStorm](#) [↗](#), а также в зависимости от дополнительного языка программирования [PHPStorm \(PHP\)](#) [↗](#), [IDEA \(Java\)](#) [↗](#), [RubyMine \(Ruby\)](#) [↗](#) и другие.
- Visual Studio, в сочетании с разработкой под .NET (Win)
- Продукты на основе Eclipse, в частности [Aptana](#) [↗](#) и [Zend Studio](#)
- [Komodo IDE](#) [↗](#) и его облегчённая версия [Komodo Edit](#) [↗](#).
- [Netbeans](#) [↗](#)

Почти все они, за исключением Visual Studio, кросс-платформенные.

Сортировка в этом списке ничего не означает. Выбор осуществляется по вкусу и по другим технологиям, которые нужно использовать вместе с JavaScript.

Большинство IDE – платные, с возможностью скачать и бесплатно использовать некоторое время. Но их стоимость, по сравнению с зарплатой веб-разработчика, невелика, поэтому ориентироваться можно на удобство.

Лёгкие редакторы

Лёгкие редакторы – не такие мощные, как IDE, но они быстрые и простые, мгновенно стартуют.

Основная сфера применения лёгкого редактора – мгновенно открыть нужный файл, чтобы что-то в нём поправить.

На практике «лёгкие» редакторы могут обладать большим количеством плагинов, так что граница между IDE и «лёгким» редактором размыта, спорить что именно редактор, а что IDE – не имеет смысла.

Достоинны внимания:

- [Sublime Text](#) [↗](#) (кросс-платформенный, shareware).
- [Atom](#) [↗](#) (кросс-платформенный, free).
- [SciTe](#) [↗](#) простой, легкий и очень быстрый (Windows, бесплатный).
- [Notepad++](#) [↗](#) (Windows, бесплатный).
- Vim, Emacs. Если умеете их готовить.

Мои редакторы

Лично мои любимые редакторы:

- Как IDE – редакторы от JetBrains: для чистого JavaScript [WebStorm](#) [↗](#), если ещё какой-то язык, то в зависимости от языка: [PHPStorm \(PHP\)](#) [↗](#), [IDEA \(Java\)](#) [↗](#), [RubyMine \(Ruby\)](#) [↗](#). У них есть и другие редакторы под разные языки, но я ими не пользовался.
- Как быстрый редактор – [Sublime Text](#) [↗](#).
- Иногда Visual Studio, если разработка идёт под платформу .NET (Win).

Если не знаете, что выбрать – можно посмотреть на них ;)

Не будем ссориться

В списках выше перечислены редакторы, которые использую я или мои знакомые – хорошие разработчики. Конечно, существуют и другие отличные редакторы, если вам что-то нравится – пользуйтесь.

Выбор редактора, как и любого инструмента, во многом индивидуален и зависит от ваших проектов, привычек, личных предпочтений.

Консоль разработчика

При разработке скриптов всегда возможны ошибки... Впрочем, что я говорю? У вас абсолютно точно будут ошибки, если конечно вы – человек, а не [робот](#).

Чтобы читать их в удобном виде, а также получать массу полезной информации о выполнении скриптов, в браузерах есть *инструменты разработки*.

Для разработки рекомендуется использовать Chrome или Firefox.

Другие браузеры, как правило, находятся в положении «догоняющих» по возможностям встроенных инструментов разработки. Если ошибка, к примеру, именно в Internet Explorer, тогда уже смотрим конкретно в нём, но обычно – Chrome/Firefox.

В инструментах разработчика предусмотрена масса возможностей, но на текущем этапе мы просто посмотрим, как их открывать, смотреть в консоли ошибки и запускать команды JavaScript.

Google Chrome

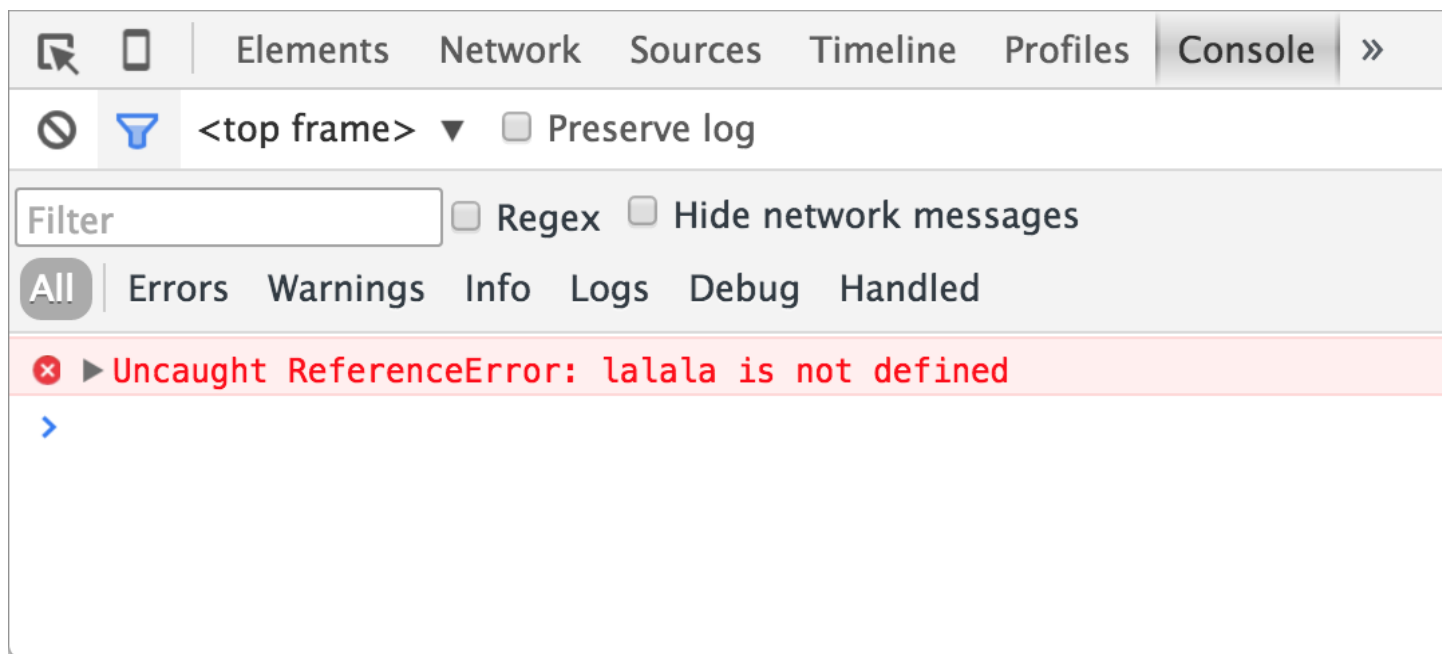
Откройте страницу [bug.html](#).

В её JavaScript-коде есть ошибка. Конечно, обычному посетителю она не видна, нужно открыть инструменты разработчика.

Для этого используйте клавишу `F12` под Windows, а если у вас Mac, то `Cmd+Opt+J`.

При этом откроются инструменты разработчика и вкладка Console, в которой будет ошибка.

Выглядеть будет примерно так:



- При клике на [bug.html](#) вы перейдёте во вкладку с кодом к месту ошибки, там будет и краткое описание ошибки. В данном случае ошибка вызвана строкой `lalala`, которая интерпретатору непонятна.
- В этом же окошке вы можете набирать команды на JavaScript. Например, наберите `alert("Hello")` – команду вывода сообщения и запустите её нажатием `Enter`. Мы познакомимся с этой и другими командами далее.
- Для перевода курсора на следующую строку (если команда состоит из нескольких строк) – используется сочетание `Shift+Enter`.

Далее в учебнике мы подробнее рассмотрим отладку в Chrome в главе [Отладка в браузере Chrome](#).

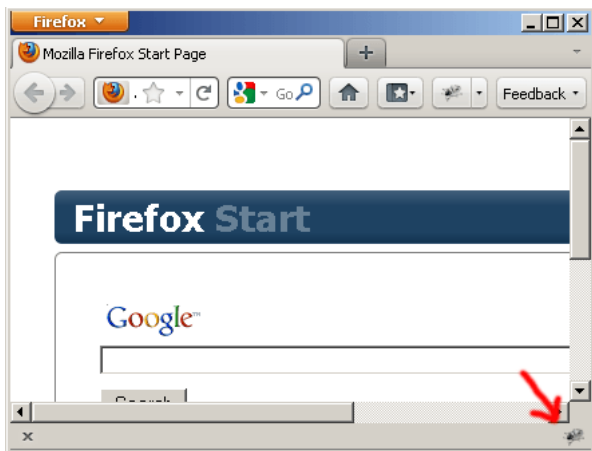
Firefox

Для разработки в Firefox используется расширение Firebug.

1. Первым делом его надо установить.

Это можно сделать со страницы <https://addons.mozilla.org/ru/firefox/addon/firebug/>.

Перезапустите браузер. Firebug появится в правом-нижнем углу окна:

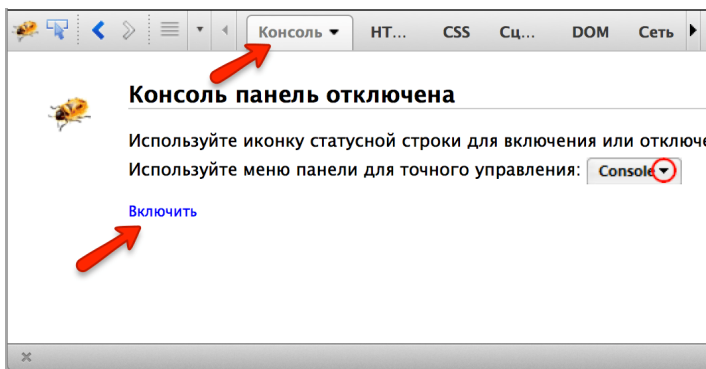


Если иконки не видно – возможно, у вас выключена панель расширений. Нажмите `Ctrl+X` для её отображения.

Ну а если её нет и там, то нажмите `F12` – это горячая клавиша для запуска Firebug, расширение появится, если установлено.

2. Далее, для того чтобы консоль заработала, её надо включить.

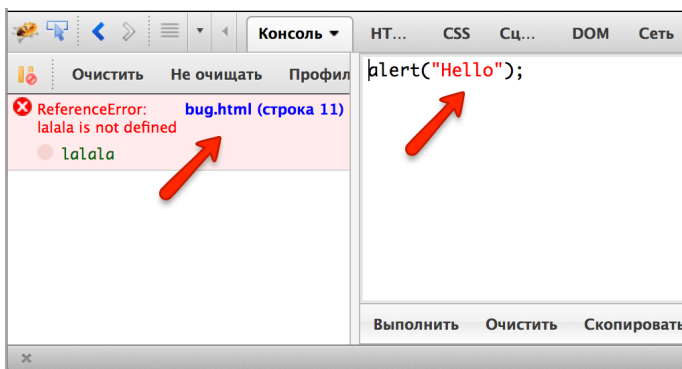
Если консоль уже была включена ранее, то этот шаг не нужен, но если она серая – выберите в меню **Консоль** и включите её:



3. Для того, чтобы Firebug работал без глюков, желательно сначала открыть Firebug, а уже потом – зайти на страницу.

С открытым Firebug зайдите на страницу с ошибкой: [bug.html](#).

Консоль покажет ошибку:



Кликните на строчке с ошибкой и браузер покажет исходный код. При необходимости включайте дополнительные панели.

Как и в Chrome, можно набирать и запускать команды. Область для команд на рисунке находится справа, запуск команд осуществляется нажатием `Ctrl+Enter` (для Mac – `Cmd+Enter`).

Можно перенести её вниз, нажав на кнопку (на рисунке её не видно, но она присутствует в правом нижнем углу панели разработки).

Об основных возможностях можно прочитать на сайте [firebug.ru](#).

Internet Explorer

Панель разработчика запускается нажатием `F12`.

Откройте её и зайдите на страницу с ошибкой: [bug.html](#). Если вы разобрались с Chrome/Firefox, то дальнейшее будет вам более-менее понятно, так как инструменты IE построены позже и по аналогии с Chrome/Firefox.

Safari

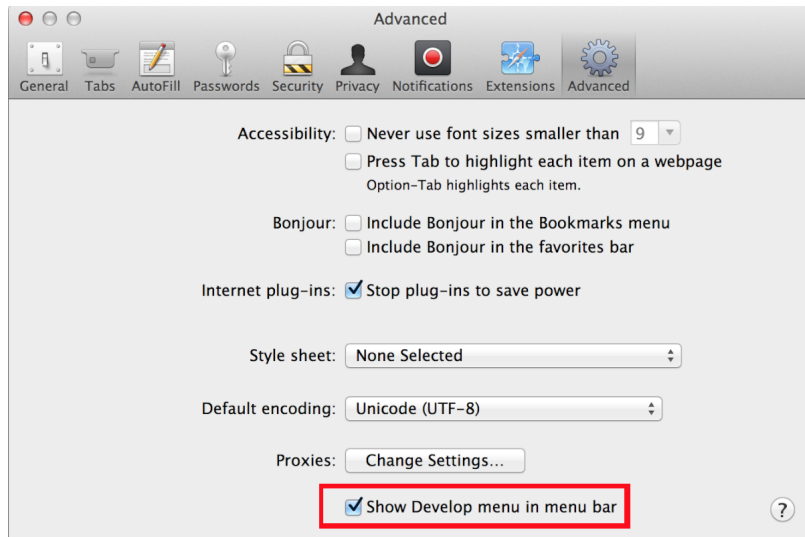
Горячие клавиши: `Ctrl+Shift+I`, `Ctrl+Alt+C` для Mac – `Cmd` вместо `Ctrl`.

Для доступа к функционалу разработки через меню:

1. В Safari первым делом нужно активировать меню разработки.

Откройте меню, нажав на колесико справа-сверху и выберите **Настройки**.

Затем вкладка **Дополнительно**:



Отметьте **Показывать меню "Разработка"** в строке меню. Закройте настройки.

2. Нажмите на колесико и выберите **Показать строку меню**.

Инструменты будут доступны в появившейся строке меню, в пункте **Разработка**.

Итого

Мы разобрали, как открывать инструменты разработчика и смотреть ошибки, а также запускать простые команды, не отходя от браузера.

Далее мы приступим к изучению JavaScript.

Основы JavaScript

Основные кирпичики, из которых состоят скрипты.

Привет, мир!

В этой статье мы создадим простой скрипт и посмотрим, как он работает.

Тег SCRIPT

i А побыстрее?

В том (и только в том!) случае, если читатель нетерпелив и уже разрабатывал на JavaScript или имеет достаточно опыта в другом языке программировании, он может не читать каждую статью этого раздела, а перепрыгнуть сразу к главе **Всё вместе: особенности JavaScript**. Там будет кратко самое основное.

Если же у вас есть достаточно времени и желание начать с азов, то читайте дальше :)

Программы на языке JavaScript можно вставить в любое место HTML при помощи тега `SCRIPT`. Например:

```
<!DOCTYPE HTML>
<html>

<head>
  <!-- Тег meta для указания кодировки -->
  <meta charset="utf-8">
</head>

<body>

  <p>Начало документа...</p>
  <script>
    alert( 'Привет, Мир!' );
  </script>

  <p>...Конец документа</p>

</body>
```

</html>

Этот пример использует следующие элементы:

`<script> ... </script>`

Тег `script` содержит исполняемый код. Предыдущие стандарты HTML требовали обязательного указания атрибута `type`, но сейчас он уже не нужен. Достаточно просто `<script>`.

Браузер, когда видит `<script>`:

1. Начинает отображать страницу, показывает часть документа до `script`
2. Встретив тег `script`, переключается в JavaScript-режим и не показывает, а исполняет его содержимое.
3. Закончив выполнение, возвращается обратно в HTML-режим и *только тогда* отображает оставшуюся часть документа.

Попробуйте этот пример в действии, и вы сами всё увидите.

alert(сообщение)

Отображает окно с сообщением и ждёт, пока посетитель не нажмёт «Ок».

i Кодировка и тег **META**

При попытке сделать такой же файл у себя на диске и запустить, вы можете столкнуться с проблемой – выведутся «кракозяблы», «квадратики» и «вопросики» вместо русского текста.

Чтобы всё было хорошо, нужно:

1. Убедиться, что в HEAD есть строка `<meta charset="utf-8">`. Если вы будете открывать файл с диска, то именно он укажет браузеру кодировку.
2. Убедиться, что редактор сохранил файл именно в кодировке UTF-8, а не, скажем, в `windows-1251`.

Указание кодировки – часть обычного HTML, я упоминаю об этом «на всякий случай», чтобы не было сюрпризов при запуске примеров локально.

Современная разметка для SCRIPT

В старых скриптах оформление тега `SCRIPT` было немного сложнее. В устаревших руководствах можно встретить следующие элементы:

Атрибут `<script type=...>`

В отличие от HTML5, стандарт HTML 4 требовал обязательного указания этого атрибута. Выглядел он так: `type="text/javascript"`. Если указать другое значение `type`, то скрипт выполнен не будет.

В современной разработке атрибут `type` не обязателен.

Атрибут `<script language=...>`

Этот атрибут предназначен для указания языка, на котором написан скрипт. По умолчанию, язык – JavaScript, так что и этот атрибут ставить не обязательно.

Комментарии до и после скриптов

В совсем старых руководствах и книгах иногда рекомендуют использовать HTML-комментарии внутри `SCRIPT`, чтобы спрятать Javascript от браузеров, которые не поддерживают его.

Выглядит это примерно так:

```
<script type="text/javascript"><!--
  ...
  //--></script>
```

Браузер, для которого предназначались такие трюки, очень старый Netscape, давно умер. Поэтому в этих комментариях нет нужды.

Итак, для вставки скрипта мы просто пишем `<script>`, без дополнительных атрибутов и комментариев.

✓ Задачи

Выведите alert

важность: 5

Сделайте страницу, которая выводит «Я – JavaScript!».

Создайте её на диске, откройте в браузере, убедитесь, что всё работает.

[Демо в новом окне](#)

Внешние скрипты, порядок исполнения

Если JavaScript-кода много – его выносят в отдельный файл, который подключается в HTML:

```
<script src="/path/to/script.js"></script>
```

Здесь /path/to/script.js – это абсолютный путь к файлу, содержащему скрипт (из корня сайта).

Браузер сам скачает скрипт и выполнит.

Можно указать и полный URL, например:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"></script>
```

Вы также можете использовать путь относительно текущей страницы. Например, src="lodash.js" обозначает файл из текущей директории.

Чтобы подключить несколько скриптов, используйте несколько тегов:

```
<script src="/js/script1.js"></script>
<script src="/js/script2.js"></script>
...
```

На заметку:

Как правило, в HTML пишут только самые простые скрипты, а сложные выносят в отдельный файл.

Браузер скачает его только первый раз и в дальнейшем, при правильной настройке сервера, будет брать из своего [кеша](#).

Благодаря этому один и тот же большой скрипт, содержащий, к примеру, библиотеку функций, может использоваться на разных страницах без полной перезагрузки с сервера.

Если указан атрибут src, то содержимое тега игнорируется.

В одном теге SCRIPT нельзя одновременно подключить внешний скрипт и указать код.

Вот так не сработает:

```
<script src="file.js">
  alert(1); // так как указан src, то внутренняя часть тега игнорируется
</script>
```

Нужно выбрать: либо SCRIPT идёт с src, либо содержит код. Тег выше следует разбить на два: один – с src, другой – с кодом, вот так:

```
<script src="file.js"></script>
<script>
  alert( 1 );
</script>
```

Асинхронные скрипты: defer/async

Браузер загружает и отображает HTML постепенно. Особенно это заметно при медленном интернет-соединении: браузер не ждёт, пока страница загрузится целиком, а показывает ту часть, которую успел загрузить.

Если браузер видит тег <script>, то он по стандарту обязан сначала выполнить его, а потом показать оставшуюся часть страницы.

Например, в примере ниже – пока все кролики не будут посчитаны – нижний <p> не будет показан:

```
<!DOCTYPE HTML>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>

  <p>Начинаем считать:</p>

  <script>
    alert( 'Первый кролик!' );
    alert( 'Второй кролик!' );
    alert( 'Третий кролик!' );
  </script>

  <p>Кролики посчитаны!</p>
```

```
</body>
</html>
```

Такое поведение называют «синхронным». Как правило, оно вполне нормально, но есть важное следствие.

Если скрипт – внешний, то пока браузер не выполнит его, он не покажет часть страницы под ним.

То есть, в таком документе, пока не загрузится и не выполнится `big.js`, содержимое `<body>` будет скрыто:

```
<html>
<head>
  <script src="big.js"></script>
</head>
<body>
  Этот текст не будет показан, пока браузер не выполнит big.js.
</body>
</html>
```

И здесь вопрос – действительно ли мы этого хотим? То есть, действительно ли оставшуюся часть страницы нельзя показывать до загрузки скрипта?

Есть ситуации, когда мы не только НЕ хотим такой задержки, но она даже опасна.

Например, если мы подключаем внешний скрипт, который показывает рекламу или вставляет счётчик посещений, а затем идёт наша страница. Конечно, неправильно, что пока счётчик или реклама не подгрузятся – оставшаяся часть страницы не показывается. Счётчик посещений не должен никак задерживать отображение страницы сайта. Реклама тоже не должна тормозить сайт и нарушать его функционал.

А что, если сервер, с которого загружается внешний скрипт, перегружен? Посетитель в этом случае может ждать очень долго!

Вот пример, с подобным скриптом (стоит искусственная задержка загрузки):

```
<p>Важная информация не покажется, пока не загрузится скрипт.</p>
<script src="https://js.cx/hello/ads.js?speed=0"></script>
<p>...Важная информация!</p>
```

Что делать?

Можно поставить все подобные скрипты в конец страницы – это уменьшит проблему, но не избавит от неё полностью, если скриптов несколько. Допустим, в конце страницы 3 скрипта, и первый из них тормозит – получается, другие два его будут ждать – тоже нехорошо.

Кроме того, браузер дойдёт до скриптов, расположенных в конце страницы, они начнут грузиться только тогда, когда вся страница загрузится. А это не всегда правильно. Например, счётчик посещений наиболее точно сработает, если загрузить его пораньше.

Поэтому «расположить скрипты внизу» – не лучший выход.

Кардинально решить эту проблему помогут атрибуты `async` или `defer` :

Атрибут `async`

Поддерживается всеми браузерами, кроме IE9-. Скрипт выполняется полностью асинхронно. То есть, при обнаружении `<script async src="...">` браузер не останавливает обработку страницы, а спокойно работает дальше. Когда скрипт будет загружен – он выполнится.

Атрибут `defer`

Поддерживается всеми браузерами, включая самые старые IE. Скрипт также выполняется асинхронно, не заставляет ждать страницу, но есть два отличия от `async` .

Первое – браузер гарантирует, что относительный порядок скриптов с `defer` будет сохранён.

То есть, в таком коде (с `async`) первым сработает тот скрипт, который раньше загрузится:

```
<script src="1.js" async></script>
<script src="2.js" async></script>
```

А в таком коде (с `defer`) первым сработает всегда `1.js` , а скрипт `2.js` , даже если загрузился раньше, будет его ждать.

```
<script src="1.js" defer></script>
<script src="2.js" defer></script>
```

Поэтому атрибут `defer` используют в тех случаях, когда второй скрипт `2.js` зависит от первого `1.js` , к примеру – использует что-то, описанное первым скриптом.

Второе отличие – скрипт с `defer` сработает, когда весь HTML-документ будет обработан браузером.

Например, если документ достаточно большой...

```
<script src="async.js" async></script>
<script src="defer.js" defer></script>
```

Много много много букв

...То скрипт `async.js` выполнится, как только загрузится – возможно, до того, как весь документ готов. А `defer.js` подождёт готовности всего документа.

Это бывает удобно, когда мы в скрипте хотим работать с документом, и должны быть уверены, что он полностью получен.

i `async` вместе с `defer`

При одновременном указании `async` и `defer` в современных браузерах будет использован только `async`, в IE9 – только `defer` (не понимает `async`).

! Атрибуты `async/defer` – только для внешних скриптов

Атрибуты `async/defer` работают только в том случае, если назначены на внешние скрипты, т.е. имеющие `src`.

При попытке назначить их на обычные скрипты `<script>...</script>`, они будут проигнорированы.

Тот же пример с `async`:

```
<p>Важная информация теперь не ждёт, пока загрузится скрипт...</p>
<script async src="https://js.cx/hello/ads.js?speed=0"></script>
<p>...Важная информация!</p>
```

При запуске вы увидите, что вся страница отобразилась тут же, а `alert` из внешнего скрипта появится позже, когда загрузится скрипт.

i Эти атрибуты давно «в ходу»

Большинство современных систем рекламы и счётчиков знают про эти атрибуты и используют их.

Перед вставкой внешнего тега `<script>` понимающий программист всегда проверит, есть ли у него подобный атрибут. Иначе медленный скрипт может задержать загрузку страницы.

i Забегая вперёд

Для продвинутого читателя, который знает, что теги `<script>` можно добавлять на страницу в любой момент при помощи самого javascript, заметим, что скрипты, добавленные таким образом, ведут себя так же, как `async`. То есть, выполняются как только загрузятся, без сохранения относительного порядка.

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'); // загрузятся эти скрипты начнут сразу
addScript('2.js'); // выполнятся, как только загрузятся
addScript('3.js'); // но, гарантированно, в порядке 1 -> 2 -> 3
```

Более подробно работу со страницей мы разберём во второй части учебника.

Итого

- Скрипты вставляются на страницу как текст в теге `<script>`, либо как внешний файл через `<script src="путь"></script>`
- Специальные атрибуты `async` и `defer` используются для того, чтобы пока грузится внешний скрипт – браузер показал остальную (следующую за ним) часть страницы. Без них этого не происходит.
- Разница между `async` и `defer`: атрибут `defer` сохраняет относительную последовательность скриптов, а `async` – нет. Кроме того, `defer` всегда ждёт, пока весь HTML-документ будет готов, а `async` – нет.

Очень важно не только читать учебник, но делать что-то самостоятельно.

Решите задачки, чтобы удостовериться, что вы всё правильно поняли.

✓ Задачи

Вывести `alert` внешним скриптом

важность: 5

Возьмите решение предыдущей задачи [Выведите alert](#) и вынесите скрипт во внешний файл `alert.js`, который расположите в той же директории.

Откройте страницу и проверьте, что вывод сообщения всё ещё работает.

[К решению](#)

Какой скрипт выполнится первым?

важность: 4

В примере ниже подключены два скрипта `small.js` и `big.js`.

Если предположить, что `small.js` загружается гораздо быстрее, чем `big.js` – какой выполнится первым?

```
<script src="big.js"></script>
<script src="small.js"></script>
```

А вот так?

```
<script async src="big.js"></script>
<script async src="small.js"></script>
```

А так?

```
<script defer src="big.js"></script>
<script defer src="small.js"></script>
```

[К решению](#)

Структура кода

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

Команды

Раньше мы уже видели пример команды: `alert('Привет, мир!')` выводит сообщение.

Для того, чтобы добавить в код ещё одну команду – можно поставить её после точки с запятой.

Например, вместо одного вызова `alert` сделаем два:

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

Как правило, каждая команда пишется на отдельной строке – так код лучше читается:

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

Точка с запятой

Точку с запятой *во многих случаях* можно не ставить, если есть переход на новую строку.

Так тоже будет работать:

```
alert('Привет')
alert('Мир')
```

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

Однако, важно то, что «во многих случаях» не означает «всегда»!

Например, запустите этот код:

```
alert(3 +
1
+ 2);
```

Выведет 6.

То есть, точка с запятой не ставится. Почему? Интуитивно понятно, что здесь дело в «незавершённом выражении», конца которого JavaScript ждёт с первой строки и поэтому не ставит точку с запятой. И здесь это, пожалуй, хорошо и приятно.

Но в некоторых важных ситуациях JavaScript «забывает» вставить точку с запятой там, где она нужна.

Таких ситуаций не так много, но ошибки, которые при этом появляются, достаточно сложно обнаруживать и исправлять.

Чтобы не быть голословным, вот небольшой пример.

Такой код работает:

```
[1, 2].forEach(alert)
```

Он выводит по очереди 1, 2. Почему он работает – сейчас не важно, позже разберёмся.

Важно, что вот такой код уже работать не будет:

```
alert("Сейчас будет ошибка")
[1, 2].forEach(alert)
```

Выведется только первый alert, а дальше – ошибка. Потому что перед квадратной скобкой JavaScript точку с запятой не ставит, а как раз здесь она нужна (упс!).

Если её поставить, то всё будет в порядке:

```
alert( "Сейчас будет ошибка" );
[1, 2].forEach(alert)
```

Поэтому в JavaScript рекомендуется точки с запятой ставить. Сейчас это, фактически, стандарт, которому следуют все большие проекты.

Комментарии

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

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

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

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

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

Многострочные комментарии начинаются слешем-звездочкой «/*» и заканчиваются звездочкой-слешем «*/», вот так:

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

Всё содержимое комментария игнорируется. Если поместить код внутрь /* ... */ или после // – он не выполнится.

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

Используйте горячие клавиши!

В большинстве редакторов комментариев можно поставить горячей клавишей, обычно это `Ctrl+/` для однострочных и что-то вроде `Ctrl+Shift+/` – для многострочных комментариев (нужно выделить блок и нажать сочетание клавиш). Детали смотрите в руководстве по редактору.

Вложенные комментарии не поддерживаются!

В этом коде будет ошибка:

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

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

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

Современный стандарт, "use strict"

Очень долго язык JavaScript развивался без потери совместимости. Новые возможности добавлялись в язык, но старые – никогда не менялись, чтобы не «сломать» уже существующие HTML/JS-страницы с их использованием.

Однако, это привело к тому, что любая ошибка в дизайне языка становилась «вмороженной» в него навсегда.

Так было до появления стандарта ECMAScript 5 (ES5), который одновременно добавил новые возможности и внёс в язык ряд исправлений, которые могут привести к тому, что старый код, который был написан до его появления, перестанет работать.

Чтобы этого не случилось, решили, что по умолчанию эти опасные изменения будут выключены, и код будет работать по-старому. А для того, чтобы перевести код в режим полного соответствия современному стандарту, нужно указать специальную директиву `use strict`.

Эта директива не поддерживается IE9-.

Директива `use strict`

Директива выглядит как строка `"use strict";` или `'use strict';` и ставится в начале скрипта.

Например:

```
"use strict";  
// этот код будет работать по современному стандарту ES5  
...
```

Отменить действие `use strict` никак нельзя

Не существует директивы `no use strict` или подобной, которая возвращает в старый режим.

Если уж вошли в современный режим, то это дорога в один конец.

`use strict` для функций

Через некоторое время мы будем проходить [функции](#). На будущее заметим, что `use strict` также можно указывать в начале функций, тогда строгий режим будет действовать только внутри функции.

В следующих главах мы будем подробно останавливаться на отличиях в работе языка при `use strict` и без него.

Нужен ли мне «`use strict`»?

Если говорить абстрактно, то – да, нужен. В строгом режиме исправлены некоторые ошибки в дизайне языка, и вообще, современный стандарт – это хорошо.

Однако, есть и две проблемы.

Поддержка браузеров IE9-, которые игнорируют `"use strict"`.

Предположим, что мы, используя `"use strict"`, разработали код и протестировали его в браузере Chrome. Всё работает... Однако, вероятность ошибок при этом в IE9- выросла! Он-то всегда работает по старому стандарту, а значит, иногда по-другому. Возникающие ошибки придётся отлаживать уже в IE9-, и это намного менее приятно, нежели в Chrome.

Впрочем, проблема не так страшна. Несовместимостей мало. И если их знать (а в учебнике мы будем останавливаться на них) и писать правильный код, то всё будет в порядке и `"use strict"` станет нашим верным помощником.

Библиотеки, написанные без учёта `"use strict"`.

Некоторые библиотеки, которые написаны без `"use strict"`, не всегда корректно работают, если вызывающий код содержит `"use strict"`.

В первую очередь имеются в виду сторонние библиотеки, которые писали не мы, и которые не хотелось бы переписывать или править.

Таких библиотек мало, но при переводе давно существующих проектов на `"use strict"` эта проблема возникает с завидной регулярностью.

Вывод?

Писать код с `use strict` следует лишь в том случае, если вы уверены, что описанных выше проблем не будет.

Конечно же, весь код, который находится в этом учебнике, корректно работает в режиме `"use strict"`.

ES5-shim

Браузер IE8 поддерживает только совсем старую версию стандарта JavaScript, а именно ES3.

К счастью, многие возможности современного стандарта можно добавить в этот браузер, подключив библиотеку [ES5 shim](#), а именно – скрипты `es5-shim.js` и `es5-sham.js` из неё.

Итого

В этой главе мы познакомились с понятием «строгий режим».

Далее мы будем предполагать, что разработка ведётся либо в современном браузере, либо в IE8- с подключённым [ES5 shim](#). Это позволит нам использовать большинство возможностей современного JavaScript во всех браузерах.

Очень скоро, буквально в следующей главе, мы увидим особенности строгого режима на конкретных примерах.

Переменные

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

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

Чтобы хранить информацию, используются *переменные*.

Переменная

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

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

```
var message;
```

После объявления, можно записать в переменную данные:

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

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

```
var message;  
message = 'Hello!';  
  
alert( message ); // выведет содержимое переменной
```

Для краткости можно совместить объявление переменной и запись данных:

```
var message = 'Hello!';
```

Можно даже объявить несколько переменных сразу:

```
var user = 'John', age = 25, message = 'Hello';
```

Аналогия из жизни

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

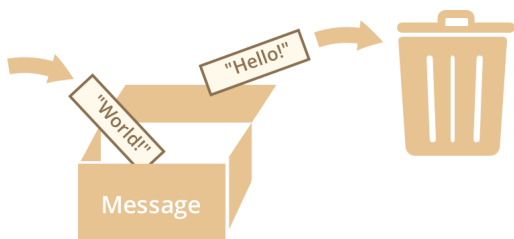
Например, переменная `message` – это коробка, в которой хранится значение `"Hello!"` :



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

```
var message;  
message = 'Hello!';  
message = 'World!'; // заменили значение  
alert( message );
```

При изменении значения старое содержимое переменной удаляется.



Можно объявить две переменные и копировать данные из одной в другую:

```
var hello = 'Hello world!';  
  
var message;  
  
// скопировали значение  
message = hello;  
  
alert( hello ); // Hello world!  
alert( message ); // Hello world!
```

На заметку:

Существуют [функциональные](#) языки программирования, в которых значение переменной менять нельзя. Например, [Scala](#) или [Erlang](#).

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

С виду – не очень удобно, но, как ни странно, и на таких языках вполне можно успешно программировать. Более того, оказывается, что в ряде областей, например в распараллеливании вычислений, они имеют преимущества. Изучение какого-нибудь функционального языка рекомендуется для расширения кругозора.

Имена переменных

На имя переменной в JavaScript наложены всего два ограничения.

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

Например:

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

Что особенно интересно – доллар '\$' и знак подчеркивания '_' являются такими же обычными символами, как буквы:

```
var $ = 1; // объявили переменную с именем '$'  
var _ = 2; // переменная с именем '_'  
  
alert( $ + _ ); // 3
```

А такие переменные были бы неправильными:

```
var 1a; // начало не может быть цифрой  
var my-name; // дефис '-' не является разрешенным символом
```

Регистр букв имеет значение

Переменные apple и AppLE – две разные переменные.

Русские буквы допустимы, но не рекомендуются

В названии переменных можно использовать и русские буквы, например:

```
var имя = "Вася";  
alert( имя ); // "Вася"
```

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

Зарезервированные имена

Существует список зарезервированных слов, которые нельзя использовать для переменных, так как они используются самим языком, например: var, class, return, export и др.

Например, такой пример выдаст синтаксическую ошибку:

```
var return = 5; // ошибка  
alert(return);
```

Важность директивы var

В старом стандарте JavaScript разрешалось создавать переменную и без var, просто присвоив ей значение:

```
num = 5; // переменная num будет создана, если ее не было
```

В режиме "use strict" так делать уже нельзя.

Следующий код выдаст ошибку:

```
"use strict";  
num = 5; // error: num is not defined
```

Обратим внимание, директиву use strict нужно ставить до кода, иначе она не сработает:

```
var something;  
"use strict"; // слишком поздно  
num = 5; // ошибки не будет, так как строгий режим не активирован
```

⚠ Ошибка в IE8- без var

Если же вы собираетесь поддерживать IE8-, то у меня для вас ещё одна причина всегда использовать var .

Следующий документ в IE8- ничего не выведет, будет ошибка:

```
<div id="test"></div>  
<script>  
  test = 5; // здесь будет ошибка!  
  alert( test ); // не сработает  
</script>
```

Это потому, что переменная test не объявлена через var и совпадает с id элемента <div> . Даже не спрашивайте почему – это ошибка в браузере IE до версии 9.

Самое «забавное» то, что такая ошибка присвоения значений будет только в IE8- и только если на странице присутствует элемент с совпадающим с именем id .

Такие ошибки особенно «весело» исправлять и отлаживать.

Вывод простой – всегда объявляем переменные через var , и сюрпризов не будет. Даже в старых IE.

Константы

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

```
var COLOR_RED = "#F00";  
var COLOR_GREEN = "#0F0";  
var COLOR_BLUE = "#00F";  
var COLOR_ORANGE = "#FF7F00";  
  
var color = COLOR_ORANGE;  
alert( color ); // #FF7F00
```

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

Зачем нужны константы? Почему бы просто не писать var color = "#FF7F00" ?

- Во-первых, константа COLOR_ORANGE – это понятное имя. По присвоению var color="#FF7F00" непонятно, что цвет – оранжевый. Иными словами, константа COLOR_ORANGE является «понятным псевдонимом» для значения #FF7F00 .
- Во-вторых, опечатка в строке, особенно такой сложной как #FF7F00 , может быть не замечена, а в имени константы её допустить куда сложнее.

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

Итого

- В JavaScript можно объявлять переменные для хранения данных. Это делается при помощи var .
- Технически, можно просто записать значение и без объявления переменной, однако по ряду причин это не рекомендуется.
- Вместе с объявлением можно сразу присвоить значение: var x = 10 .
- Переменные, которые названы БОЛЬШИМИ_БУКВАМИ , являются константами, то есть никогда не меняются. Как правило, они используются для удобства, чтобы было меньше ошибок.

✔ Задачи

Работа с переменными

важность: 2

1. Объявите две переменные: `admin` и `name`.
2. Запишите в `name` строку "Василий".
3. Скопируйте значение из `name` в `admin`.
4. Выведите `admin` (должно вывести «Василий»).

[К решению](#)

Правильный выбор имени переменной

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

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

Возможно, эти слова не очевидны, если вы пока что ничего большого не писали или пишете код «только для чтения» (написал 5 строк, отдал заказчику и забыл). Но чем более серьёзные проекты вы будете делать, тем более актуальны они будут для вас.

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

Гораздо проще найти нужные данные, если они правильно помечены, то есть когда переменная названа *правильно*.

Правила именования

- Правило 1.

Никакого транслита. Только английский.

Неприемлемы:

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

Подойдут:

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

Чем плох транслит?

Во-первых, среди разработчиков всего мира принято использовать английский язык для имён переменных. И если ваш код потом попадёт к кому-то другому, например вы будете в команде больше чем из одного человека, то велик шанс, что транслит ему не понравится.

Во-вторых, русский транслит хуже читается и длиннее, чем названия на английском.

В-третьих, в проектах вы наверняка будете применять библиотеки, написанные другими людьми. Многие уже готовы, в распоряжении современного разработчика есть масса инструментов, все они используют названия переменных и функций на английском языке, и вы, конечно, будете их использовать. А от кода, где транслит перемешан с английским – волосы могут встать дыбом, и не только на голове.

Если вы вдруг не знаете английский – самое время выучить.

- Правило 2.

Использовать короткие имена только для переменных «местного значения».

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

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

- Правило 3.

Переменные из нескольких слов пишутся **вместе** Вот Так .

Например:

```
var borderLeftWidth;
```

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

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

```
var border_left_width;
```

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

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

- Правило последнее, главное.

Имя переменной должно максимально чётко соответствовать хранимым в ней данным.

Придумывание таких имен – одновременно коротких и точных, при которых всегда понятно, что где лежит, приходит с опытом, но только если сознательно стремиться к этому.

Позвольте поделиться одним небольшим секретом, который очень прост, но позволит улучшить названия переменных и сэкономит время.

Бывает так, что, написав код, мы через некоторое время к нему возвращаемся, надо что-то поправить. И мы примерно помним, что переменная, в которой хранится нужное вам значение, называется... Ну, скажем, `borderLeftWidth`. Мы ищем её в коде, не находим, но, разобравшись, обнаруживаем, что на самом деле переменная называлась вот так: `leftBorderWidth`.

Если мы ищем переменную с одним именем, а находим – с другим, то зачастую самый лучший ход – это *переименовать* переменную, чтобы имя было тем, которое вы искали.

То есть, в коде `leftBorderWidth`, а мы её переименуем на ту, которую искали: `borderLeftWidth`.

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

Есть причина и поважнее. Поскольку именно это имя переменной пришло в голову – скорее всего, оно больше соответствует хранимым там данным, чем то, которое было мы придумали изначально. Исключения бывают, но в любом случае – такое несоответствие – это повод задуматься.

Чтобы удобно переименовывать переменную, нужно использовать [хороший редактор JavaScript](#), тогда этот процесс будет очень простым и быстрым.

Если коротко...

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

Не нужно бояться переименовывать переменные, если вы придумали имя получше.

Современные редакторы позволяют делать это очень удобно и быстро. Это в конечном счете сэкономит вам время.

Храните в переменной то, что следует

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

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

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

«Лишняя» переменная – добро, а не зло.

Задачи

Объявление переменных

важность: 3

1. Создайте переменную для названия нашей планеты и присвойте ей значение "Земля". *Правильное имя выберите сами.*
2. Создайте переменную для имени посетителя со значением "Петя". Имя также на ваш вкус.

[К решению](#)

Шесть типов данных, typeof

В JavaScript существует несколько основных типов данных.

В этой главе мы получим о них общее представление, а позже, в соответствующих главах подробно познакомимся с использованием каждого типа в отдельности.

Число «number»

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

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

Существуют специальные числовые значения `Infinity` (бесконечность) и `NaN` (ошибка вычислений).

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


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

Ошибка вычислений NaN будет результатом некорректной математической операции, например:

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

Эти значения формально принадлежат типу «число», хотя, конечно, числами в их обычном понимании не являются.

Особенности работы с числами в JavaScript разобраны в главе [Числа](#).

Строка «string»

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

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

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

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

Более подробно со строками мы познакомимся в главе [Строки](#).

Булевый (логический) тип «boolean»

У него всего два значения: `true` (истина) и `false` (ложь).

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

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

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

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;
```

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

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

Специальное значение «undefined»

Значение `undefined`, как и `null`, образует свой собственный тип, состоящий из одного этого значения. Оно имеет смысл «значение не присвоено».

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

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

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

```
var x = 123;  
x = undefined;  
alert( x ); // "undefined"
```

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

Объекты «object»

Первые 5 типов называют «*примитивными*».

Особняком стоит шестой тип: «*объекты*».

Он используется для коллекций данных и для объявления более сложных сущностей.

Объявляются объекты при помощи фигурных скобок `{...}`, например:

```
var user = { name: "Вася" };
```

Мы подробно разберём способы объявления объектов и, вообще, работу с объектами, позже, в главе [Объекты как ассоциативные массивы](#).

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)
```

Последние две строки помечены, потому что `typeof` ведёт 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 мы пройдем чуть позже. Пока лишь заметим, что 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.

К работе с типами мы также вернёмся более подробно в будущем, после изучения основных структур данных.

Итого

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

Очень скоро мы изучим их во всех деталях.

Оператор `typeof x` позволяет выяснить, какой тип находится в `x`, возвращая его в виде строки.

Основные операторы

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

Несколько операторов мы знаем со школы – это обычные сложение `+`, умножение `*`, вычитание и так далее.

В этой главе мы сконцентрируемся на операторах, которые в курсе математики не проходят, и на их особенностях в JavaScript.

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

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

Прежде, чем мы двинемся дальше – несколько терминов, чтобы понимать, о чём речь.

- *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
```

- **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 +

Обычно при помощи плюса '+' складывают числа.

Но 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"
```

Это приведение к строке – особенность исключительно бинарного оператора "+".

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

Например:

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

Преобразование к числу, унарный плюс +

Унарный, то есть применённый к одному значению, плюс ничего не делает с числами:

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

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

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

Например, 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
```

С точки зрения математики такое изобилие плюсов может показаться странным. С точки зрения программирования – никаких разночтений: сначала выполнятся унарные плюсы, приведут строки к числам, а затем – бинарный '+' их сложит.

Почему унарные плюсы выполнились до бинарного сложения? Как мы сейчас увидим, дело в их приоритете.

Приоритет

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

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

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

Отрывок из таблицы:

Приоритет	Название	Обозначение
-----------	----------	-------------

...
15	унарный плюс	+
15	унарный минус	-
14	умножение	*
14	деление	/
13	сложение	+
13	вычитание	-
...
3	присваивание	=
...

Так как «унарный плюс» имеет приоритет 15, выше, чем 13 у обычного «сложения», то в выражении `+apples + +oranges` сначала сработали плюсы у `apples` и `oranges`, а затем уже обычное сложение.

Присваивание

Обратим внимание, в таблице приоритетов также есть оператор присваивания `=`.

У него – один из самых низких приоритетов: 3.

Именно поэтому, когда переменную чему-либо присваивают, например, `x = 2 * 2 + 1` сначала выполнится арифметика, а уже затем – произойдёт присваивание `=`.

```
var x = 2 * 2 + 1;
alert( x ); // 5
```

Возможно присваивание по цепочке:

```
var a, b, c;
a = b = c = 2 + 2;

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

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

Оператор "=" возвращает значение

Все операторы возвращают значение. Вызов `x = выражение` 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
```

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

Забавное применение присваивания, не так ли?

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

Взятие остатка %

Оператор взятия остатка `%` интересен тем, что, несмотря на обозначение, никакого отношения к процентам не имеет.

Его результат `a % b` – это остаток от деления `a` на `b`.

Например:

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

Инкремент/декремент: ++, --

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


Для этого существуют даже специальные операторы:

- Инкремент `++` увеличивает на 1:

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

- Декремент `--` уменьшает на 1:

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

 **Важно:**

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 её 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
```

i 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, выполнен раньше но значение вернул старое
```

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 переменная увеличивается.

Три строки, по одному действию в каждой – длиннее, зато нагляднее:

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

Побитовые операторы

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

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

Поддерживаются следующие побитовые операторы:

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

Они используются редко, поэтому вынесены в отдельную главу [Побитовые операторы](#).

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)
```

Оператор запятой

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, которое возвращается.

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

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

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

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

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

✔ Задачи

Инкремент, порядок срабатывания

важность: 5

Посмотрите, понятно ли вам, почему код ниже работает именно так?

```
var a = 1, b = 1, c, d;

c = ++a; alert(c); // 2
d = b++; alert(d); // 1

c = (2+ ++a); alert(c); // 5
d = (2+ b++); alert(d); // 4

alert(a); // 3
alert(b); // 3
```

[К решению](#)

Результат присваивания

важность: 3

Чему будет равен `x` в примере ниже?

```
var a = 2;

var x = 1 + (a *= 2);
```

[К решению](#)

Операторы сравнения и логические значения

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

Многие операторы сравнения знакомы нам из математики:

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

Логические значения

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

Существует всего два логических значения:

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

Например:

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

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

```
var a = true; // присваивать явно

var b = 3 > 4; // или как результат сравнения
alert( b ); // false

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

Сравнение строк

Строки сравниваются побуквенно:

```
alert( 'Б' > 'А' ); // true
```

Осторожно, Unicode!

Аналогом «алфавита» во внутреннем представлении строк служит кодировка, у каждого символа – свой номер (код). JavaScript использует кодировку [Unicode](#).

При этом сравниваются *численные коды символов*. В частности, код у символа Б больше, чем у А, поэтому и результат сравнения такой.

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

Поэтому регистр имеет значение:

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

Для корректного сравнения символы должны быть в одинаковом регистре.

Если строка состоит из нескольких букв, то сравнение осуществляется как в телефонной книжке или в словаре. Сначала сравниваются первые буквы, потом вторые, и так далее, пока одна не будет больше другой.

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

Например:

- Если первая буква первой строки больше – значит первая строка больше, независимо от остальных символов:

```
alert( 'Банан' > 'Аят' );
```

- Если одинаковы – сравнение идёт дальше. Здесь оно дойдёт до третьей буквы:

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

- При этом любая буква больше отсутствия буквы:

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

Такое сравнение называется *лексикографическим*.

Важно:

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

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

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

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

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

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

Сравнение разных типов

При сравнении 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.
```

Тема преобразований типов будет продолжена далее, в главе [Преобразование типов для примитивов](#).

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`.

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

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.

Сравнение с `null` и `undefined`

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

Интуитивно кажется, что `null/undefined` эквивалентны нулю, но это не так.

Они ведут себя по-другому.

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

Посмотрим забавные следствия.

Некорректный результат сравнения `null` с `0`

Сравним `null` с нулём:

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

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

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

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

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

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

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

Несравнимый `undefined`

Значение `undefined` вообще нельзя сравнивать:

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

- Сравнения (1) и (2) дают `false` потому, что `undefined` при преобразовании к числу даёт `NaN`. А значение `NaN` по стандарту устроено так, что сравнения `==`, `<`, `>`, `<=`, `>=` и даже `===` с ним возвращают `false`.

- Проверка равенства (3) даёт false , потому что в стандарте явно прописано, что undefined равно лишь null и ничему другому.

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

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

Итого

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

Мы ещё вернёмся к теме сравнения позже, когда лучше изучим различные типы данных в JavaScript.

Побитовые операторы

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

Эта глава требует дополнительных знаний в программировании и не очень важная, при первом чтении вы можете пропустить её и вернуться потом, когда захотите понять, как побитовые операторы работают.

Формат 32-битного целого числа со знаком

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

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

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

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

Примеры представления чисел в двоичной системе:

```
a = 0; // 00000000000000000000000000000000
a = 1; // 00000000000000000000000000000001
a = 2; // 00000000000000000000000000000010
a = 3; // 00000000000000000000000000000011
a = 255; // 0000000000000000000000000011111111
```

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

- *Дополнение до двойки* – это название способа поддержки отрицательных чисел.

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

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

Например, вот число 314 :

```
0000000000000000000000000100111010
```

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

```
111111111111111111111111011000101
```

Второй шаг – к полученному двоичному числу прибавить единицу, обычным двоичным сложением: 111111111111111111111111011000101 + 1 = 111111111111111111111111011000110 .

Итак, мы получили:

```
-314 = 111111111111111111111111011000110
```

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

Список операторов

i Исключающее ИЛИ в шифровании

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

Иначе говоря, верна формула: $a \oplus b \oplus b == a$.

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

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

Алгоритм:

- Вася берёт двоичное представление `data` и делает операцию $data \oplus key$. При необходимости `data` бьётся на части, равные по длине `key`, чтобы можно было провести побитовое ИЛИ \oplus для каждой части. В JavaScript оператор \oplus работает с 32-битными целыми числами, так что `data` нужно разбить на последовательность таких чисел.
- Результат $data \oplus key$ отправляется Пете, это шифровка.

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

Данные: 9 в двоичном виде
`000000000000000000000000000001001`

Ключ: 1220461917 в двоичном виде
`0100100010111110110001010101101`

Результат операции $9 \oplus key$:
`010010001011111011000101010100`
Результат в 10-ой системе (шифровка):
`1220461908`

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

В нашем случае:

Полученная шифровка в двоичной системе:
 $9 \oplus key = 1220461908$
`010010001011111011000101010100`

Ключ: 1220461917 в двоичном виде:
`0100100010111110110001010101101`

Результат операции $1220461908 \oplus key$:
`000000000000000000000000000001001`
Результат в 10-ой системе (исходное сообщение):
`9`

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

~ (Побитовое НЕ)

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

Таблица истинности для НЕ:

a	~a
0	1
1	0

Пример:

```
9 (по осн. 10)
= 000000000000000000000000000001001 (по осн. 2)
~9 (по осн. 10)
= 1111111111111111111111111110110 (по осн. 2)
= -10 (по осн. 10)
```

Из-за внутреннего представления отрицательных чисел получается так, что $\sim n == -(n+1)$.

Например:

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



```
var guest = ACCESS_ARTICLE_VIEW | ACCESS_GOODS_VIEW; // 10100
var editor = guest | ACCESS_ARTICLE_EDIT | ACCESS_GOODS_EDIT; // 11110
var admin = editor | ACCESS_ADMIN; // 11111
```

Теперь, чтобы понять, есть ли в доступе `editor` нужный доступ, например управление правами – достаточно применить к нему побитовый оператор И (`&`) с соответствующей константой.

Ненулевой результат будет означать, что доступ есть:

```
alert(editor & ACCESS_ADMIN); // 0, доступа нет
alert(editor & ACCESS_ARTICLE_EDIT); // 8, доступ есть
```

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

Можно проверить один из нескольких доступов.

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

```
var check = ACCESS_GOODS_VIEW | ACCESS_GOODS_EDIT; // 6, 00110
alert( admin & check ); // не 0, значит есть доступ к просмотру ИЛИ изменению
```

Битовой маской называют как раз комбинацию двоичных значений (`check` в примере выше), которая используется для проверки и выборки единиц на нужных позициях.

Маски могут быть весьма удобны.

В частности, их используют в функциях, чтобы одним параметром передать несколько «флагов», т.е. однобитных значений.

Пример вызова функции с маской:

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

Это довольно-таки коротко и элегантно, но, вместе с тем, применение масок налагает определённые ограничения. В частности, побитовые операторы в JavaScript работают только с 32-битными числами, а значит, к примеру, 33 доступа уже в число не упакуешь. Да и работа с двоичной системой счисления – как ни крути, менее удобна, чем с десятичной или с обычными логическими значениями `true/false`.

Поэтому основная сфера применения масок – это быстрые вычисления, экономия памяти, низкоуровневые операции, связанные с рисованием из JavaScript (3d-графика), интеграция с некоторыми функциями ОС (для серверного JavaScript), и другие ситуации, когда уже существуют функции, требующие битовую маску.

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 $-n = \sim n + 1$. Или, если перенести единицу: $\sim n = -(n+1)$.

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

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



```
var n = -1;

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

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

```
var str = "Проверка";

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

Умножение и деление на степени 2

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

Например:

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

При этом следует иметь в виду, что максимальный верхний порог такого умножения меньше, чем обычно, так как побитовый оператор оперирует 32-битными целыми, в то время как обычные операторы оперируют числами длиной 64 бита.

Оператор сдвига в другую сторону `a >> b`, производит обратную операцию – целочисленное деление `a` на 2^b .

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

Итого

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

Как правило, битовое представление числа используется для:

- Округления числа: $(12.34^{\wedge}0) = 12$.
- Проверки на равенство `-1`: `if (~n) { n не -1 }` .
- Упаковки нескольких битовых значений («флагов») в одно значение. Это экономит память и позволяет проверять наличие комбинации флагов одним оператором `&` .
- Других ситуаций, когда нужны битовые маски.

✔ Задачи

Побитовый оператор и значение

важность: 5

Почему побитовые операции в примерах ниже не меняют число? Что они делают внутри?

```
alert( 123 ^ 0 ); // 123
alert( 0 ^ 123 ); // 123
alert( ~~123 ); // 123
```

[К решению](#)

Проверка, целое ли число

важность: 3

Напишите функцию `isInteger(num)`, которая возвращает `true`, если `num` – целое число, иначе `false` .

Например:

```
alert( isInteger(1) ); // true
alert( isInteger(1.5) ); // false
alert( isInteger(-0.5) ); // false
```

[К решению](#)

Симметричны ли операции ^, |, &?

важность: 5

Верно ли, что для любых a и b выполняются равенства ниже?

- $a \wedge b == b \wedge a$
- $a \& b == b \& a$
- $a | b == b | a$

Иными словами, при перемене мест – всегда ли результат остаётся тем же?

[К решению](#)

Почему результат разный?

важность: 5

Почему результат второго `alert`'а такой странный?

```
alert( 123456789 ^ 0 ); // 123456789
alert( 12345678912345 ^ 0 ); // 1942903641
```

[К решению](#)

Взаимодействие с пользователем: alert, prompt, confirm

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

alert

Синтаксис:

```
alert(сообщение)
```

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

prompt

Функция `prompt` принимает два аргумента:

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

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

Пользователь должен либо что-то ввести и нажать ОК, 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.

Safari 5.1+ не возвращает `null`

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?', 100);
alert('Вам ' + years + ' лет!')
```

⚠ Всегда указывайте **default**

Второй параметр может отсутствовать. Однако при этом IE вставит в диалог значение по умолчанию "undefined" .

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

```
var test = prompt("Тест");
```

Поэтому рекомендуется *всегда* указывать второй аргумент:

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

confirm

Синтаксис:

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

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

Результатом будет **true** при нажатии OK и **false** – при CANCEL (**Esc**).

Например:

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

Особенности встроенных функций

Конкретное место, где выводится модальное окно с вопросом – обычно это центр браузера, и внешний вид окна выбирает браузер. Разработчик не может на это влиять.

С одной стороны – это недостаток, так как нельзя вывести окно в своем, особо красивом, дизайне.

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

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

Резюме

- alert выводит сообщение.
- prompt выводит сообщение и ждёт, пока пользователь введёт текст, а затем возвращает введённое значение или null , если ввод отменён (CANCEL/ Esc).
- confirm выводит сообщение и ждёт, пока пользователь нажмёт «OK» или «CANCEL» и возвращает true/false .

✔ Задачи

Простая страница

важность: 4

Создайте страницу, которая спрашивает имя и выводит его.

[Запустить демо](#)

[К решению](#)

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;
```

«Тернарный оператор»

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

Несколько операторов „?“

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

if (строка с нулём)

важность: 5

Выведется ли alert ?

```
if ("0") {  
  alert( 'Привет' );  
}
```

[К решению](#)

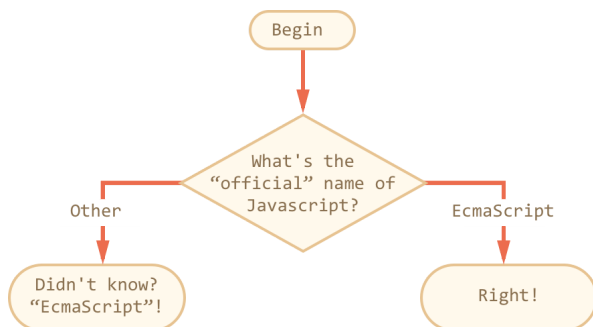
Проверка стандарта

важность: 2

Используя конструкцию if..else , напишите код, который будет спрашивать: «Каково «официальное» название JavaScript?».

Если посетитель вводит «ECMAScript», то выводить «Верно!», если что-то другое – выводить «Не знаете? «ECMAScript»!».

Блок-схема:



[Демо в новом окне ↗](#)

[К решению](#)

Получить знак числа

важность: 2

Используя конструкцию if..else , напишите код, который получает значение prompt , а затем выводит alert :

- 1 , если значение больше нуля,
- -1 , если значение меньше нуля,

- 0, если значение равно нулю.

[Демо в новом окне](#)

[К решению](#)

Проверка логина

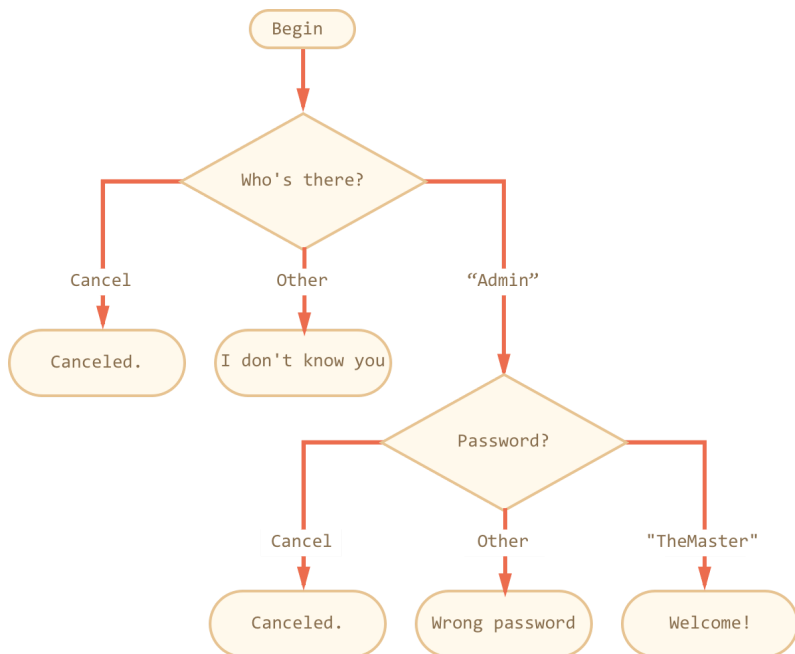
важность: 3

Напишите код, который будет спрашивать логин (prompt).

Если посетитель вводит «Админ», то спрашивать пароль, если нажал отмена (escape) – выводить «Вход отменён», если вводит что-то другое – «Я вас не знаю».

Пароль проверять так. Если введён пароль «Чёрный Властелин», то выводить «Добро пожаловать!», иначе – «Пароль неверен», при отмене – «Вход отменён».

Блок-схема:



Для решения используйте вложенные блоки if. Обращайте внимание на стиль и читаемость кода.

[Запустить демо](#)

[К решению](#)

Перепишите 'if' в '?'

важность: 5

Перепишите if с использованием оператора '?' :

```
if (a + b < 4) {
  result = 'Мало';
} else {
  result = 'Много';
}
```

[К решению](#)

Перепишите 'if..else' в '?'

важность: 5

Перепишите if..else с использованием нескольких операторов '?' .

Для читаемости – оформляйте код в несколько строк.

```
var message;
if (login == 'Вася') {
  message = 'Привет';
} else if (login == 'Директор') {
  message = 'Здравствуйте';
}
```

```
} else if (login == '') {
  message = 'Нет логина';
} else {
  message = '';
}
```

[К решению](#)

Логические операторы

Для операций над логическими значениями в JavaScript есть `||` (ИЛИ), `&&` (И) и `!` (НЕ).

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

`||` (ИЛИ)

Оператор ИЛИ выглядит как двойной символ вертикальной черты:

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

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

Значение ИЛИ

```
// запинаятся на «правде»,  
&& запинаятся на «лжи».
```

“ Илья Канатов, участник курса JavaScript

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 `||` запинаятся на правде".

&& (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 работает, 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 у `&&` больше, чем у `||`

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.

!(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
```

Задачи

Что выведет alert (ИЛИ)?

важность: 5

Что выведет код ниже?

```
alert( null || 2 || undefined );
```

[К решению](#)

Что выведет alert (ИЛИ)?

важность: 3

Что выведет код ниже?

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

[К решению](#)

Что выведет alert (И)?

важность: 5

Что выведет код ниже?

```
alert( 1 && null && 2 );
```

[К решению](#)

Что выведет alert (И)?

важность: 3

Что выведет код ниже?

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

[К решению](#)

Что выведет этот код?

важность: 5

Что выведет код ниже?

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

[К решению](#)

Проверка if внутри диапазона

важность: 3

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

«Включительно» означает, что концы промежутка включены, то есть `age` может быть равна 14 или 90.

[К решению](#)

Проверка if вне диапазона

важность: 3

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.

[К решению](#)

Вопрос про "if"

важность: 5

Какие из этих `if` верны, т.е. выполняются?

Какие конкретно значения будут результатами выражений в условиях `if(...)` ?

```
if (-1 || 0) alert( 'первое' );
if (-1 && 0) alert( 'второе' );
if (null || -1 && 1) alert( 'третье' );
```

[К решению](#)

Преобразование типов для примитивов

Система преобразования типов в JavaScript очень проста, но отличается от других языков. Поэтому она часто служит «камнем преткновения» для приходящих из других языков программистов.

Всего есть три преобразования:

1. Строковое преобразование.
2. Числовое преобразование.
3. Преобразование к логическому значению.

Эта глава описывает преобразование только примитивных значений, объекты разбираются далее.

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" ); // "truestest"
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...
<code>undefined</code>	<code>NaN</code>
<code>null</code>	<code>0</code>
<code>true</code> / <code>false</code>	<code>1</code> / <code>0</code>
Строка	Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то <code>0</code> , иначе из непустой строки "считывается" число, при ошибке результат <code>NaN</code> .

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 (пункт [11.9.3](#)). 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 даст false , правая – 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.

Итого

В JavaScript есть три преобразования:

1. Строковое: String(value) – в строковом контексте или при сложении со строкой. Работает очевидным образом.
2. Численное: Number(value) – в численном контексте, включая унарный плюс +value . Происходит при сравнении разных типов, кроме строгого равенства.
3. Логическое: Boolean(value) – в логическом контексте, можно также сделать двойным НЕ: !!value .

Точные таблицы преобразований даны выше в этой главе.

Особым случаем является проверка равенства с null и undefined . Они равны друг другу, но не равны чему бы то ни было ещё, этот случай прописан особо в спецификации.

✔️ Задачи

Вопросник по преобразованиям, для примитивов

важность: 5

Подумайте, какой результат будет у выражений ниже. Тут не только преобразования типов. Когда закончите – сверьтесь с решением.

```
" " + 1 + 0
" " - 1 + 0
true + false
6 / "3"
"2" * "3"
4 + 5 + "px"
"$" + 4 + 5

"4" - 2

"4px" - 2

7 / 0

" -9\n" + 5
" -9\n" - 5
5 && 2

2 && 5

5 || 0

0 || 5
null + 1
undefined + 1
null == "\n0\n"
+null == +"\n0\n"
```

[К решению](#)

Циклы while, for

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

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

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

Цикл while

Цикл `while` имеет вид:

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

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

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

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

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);
```

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

Цикл for

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

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

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.

i На заметку:

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.

i for..in

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

Мы познакомимся с ней позже, когда будем [говорить об объектах](#).

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.

Следующая итерация: `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);
}
```

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) {
    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.

Чтобы организовать бесконечный цикл, используют конструкцию `while(true)`. При этом он, как и любой другой цикл, может быть прерван директивой `break`.

Если на данной итерации цикла делать больше ничего не надо, но полностью прекращать цикл не следует – используют директиву `continue`.

Обе этих директивы поддерживают «метки», которые ставятся перед циклом. Метки – единственный способ для `break/continue` повлиять на выполнение внешнего цикла.

Заметим, что метки не позволяют прыгнуть в произвольное место кода, в JavaScript нет такой возможности.

✔ Задачи

Последнее значение цикла

важность: 3

Какое последнее значение выведет этот код? Почему?

```
var i = 3;
while (i) {
  alert( i-- );
}
```

[К решению](#)

Какие значения `i` выведет цикл `while`?

важность: 4

Для каждого цикла запишите, какие значения он выведет. Потом сравните с ответом.

1. Префиксный вариант

```
var i = 0;
while (++i < 5) alert( i );
```

2. Постфиксный вариант

```
var i = 0;
while (i++ < 5) alert( i );
```

[К решению](#)

Какие значения `i` выведет цикл `for`?

важность: 4

Для каждого цикла запишите, какие значения он выведет. Потом сравните с ответом.

1. Постфиксная форма:

```
for (var i = 0; i < 5; i++) alert( i );
```

2. Префиксная форма:

```
for (var i = 0; i < 5; ++i) alert( i );
```

[К решению](#)

Выведите чётные числа

важность: 5

При помощи цикла `for` выведите чётные числа от 2 до 10.

[Запустить демо](#)

[К решению](#)

Замените `for` на `while`

важность: 5

Перепишите код, заменив цикл `for` на `while`, без изменения поведения цикла.

```
for (var i = 0; i < 3; i++) {
  alert( "номер " + i + "!" );
}
```

[К решению](#)

Повторять цикл, пока ввод неверен

важность: 5

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).

Предполагается, что посетитель вводит только числа. Предусматривать обработку нечисловых строк в этой задаче необязательно.

[Запустить демо](#)

[К решению](#)

Вывести простые числа

важность: 3

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.

[К решению](#)

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:
    alert( 1 );
    break;

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

```

Группировка case

Несколько значений `case` можно группировать.

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

```

var a = 2+2;

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

  case 3: // (*)
  case 5: // (**)
    alert('Неверно!');
    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", аналогичный "switch"

важность: 5

Напишите `if..else`, соответствующий следующему `switch`:

```
switch (browser) {
  case 'IE':
    alert( '0, 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!' );
}
```

[К решению](#)

Переписать if'ы в switch

важность: 4

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' );
}
```

[К решению](#)

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.

Объявление

Пример объявления функции:

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

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

Объявленная функция доступна по имени, например:

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

```
showMessage();
showMessage();
```

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

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

Локальные переменные

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

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

  alert( message );
}

showMessage(); // 'Привет, я - Вася!'

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

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

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

Например:

```
function count() {
  // 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.

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

Параметры

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

Например, этот код выводит два сообщения:

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

  from = "*** " + from + " ***"; // здесь может быть сложный код оформления

  alert(from + ': ' + 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");

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 $b^2 - 4ac$:

```
function calcD(a, b, c) {  
  return b*b - 4*a*c;  
}  
  
var test = calcD(-4, 2, 1);  
alert(test); // 20
```

Для возврата значения используется директива `return`.

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

Вызовов `return` может быть и несколько, например:

```
function checkAge(age) {  
  if (age > 18) {  
    return true;  
  } else {  
    return confirm('Родители разрешили?');  
  }  
}  
  
var age = prompt('Ваш возраст?');  
  
if (checkAge(age)) {  
  alert( 'Доступ разрешен' );  
} else {  
  alert( 'В доступе отказано' );  
}
```

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

Например:

```
function showMovie(age) {  
  if (!checkAge(age)) {  
    return;  
  }  
  
  alert( "Фильм не для всех" ); // (*)
```

```
} // ...
```

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

i 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.

i 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.

i Сверхкороткие имена функций

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

Например, во фреймворке [jQuery](#) есть функция `$`, во фреймворке [Prototype](#) – функция `$$`, а в библиотеке [LoDash](#) очень активно используется функция с названием из одного символа подчеркивания `_`.

Итого

Объявление функции имеет вид:

```
function имя(параметры, через, запятой) {  
  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.

Именованние 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.

✔ Задачи

Обязателен ли "else"?

важность: 4

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?');
  }
}
```

Будет ли эта функция работать как-то иначе, если убрать `else`?

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

Есть ли хоть одно отличие в поведении этого варианта?

[К решению](#)

Перепишите функцию, используя оператор '?' или '||'

важность: 4

Следующая функция возвращает `true`, если параметр `age` больше 18. В ином случае она задаёт вопрос `confirm` и возвращает его результат.

```
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. Сделайте два варианта функции `checkAge`:

1. Используя оператор `'?'`
2. Используя оператор `'||'`

[К решению](#)

Функция `min`

важность: 1

Задача «Hello World» для функций :)

Напишите функцию `min(a,b)`, которая возвращает меньшее из чисел `a,b`.

Пример вызовов:

```
min(2, 5) == 2
min(3, -1) == -1
min(1, 1) == 1
```

[К решению](#)

Функция pow(x,n)

важность: 4

Напишите функцию `pow(x,n)`, которая возвращает `x` в степени `n`. Иначе говоря, умножает `x` на себя `n` раз и возвращает результат.

```
pow(3, 2) = 3 * 3 = 9
pow(3, 3) = 3 * 3 * 3 = 27
pow(1, 100) = 1 * 1 * ... * 1 = 1
```

Создайте страницу, которая запрашивает `x` и `n`, а затем выводит результат `pow(x,n)`.

[Запустить демо](#)

P.S. В этой задаче функция обязана поддерживать только натуральные значения `n`, т.е. целые от 1 и выше.

[К решению](#)

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.

A Function Expression создаются в процессе выполнении 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( 'Прощу Вас!' );
  }
} else {
  sayHi = function() {
    alert( 'До 18 нельзя' );
  }
}

sayHi();
```

Или даже так:

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

var sayHi = (age >= 18) ?
  function() { alert('Прощу 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.

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 <code>if</code>	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.

Рекурсия, стек

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.

Сейчас мы посмотрим примеры.

Рекурсия – общая тема программирования, не относящаяся напрямую к JavaScript. Если вы разрабатывали на других языках или изучали программирование раньше в ВУЗе, то наверняка уже знаете, что это такое.

Эта глава предназначена для читателей, которые пока с этой темой незнакомы и хотят лучше разобраться в том, как работают функции.

Степень $\text{pow}(x, n)$ 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:

```
 $\text{pow}(x, n) = x * \text{pow}(x, n - 1)$ 
```

То есть, $x^n = x * x^{n-1}$.

Например, вычислим `pow(2, 4)`, последовательно переходя к более простой задаче:

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

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 рекурсии 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.

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 вложенного 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.

Здесь мы постарались рассмотреть происходящее достаточно подробно, однако, если пожелаете, допустимо временно забежать вперёд и открыть главу [Отладка в браузере Chrome](#), с тем чтобы при помощи отладчика построчно пробежаться по коду и посмотреть стек на каждом шаге. Отладчик даёт к нему доступ.

✔ Задачи

Вычислить сумму чисел до данного

важность: 5

Напишите функцию `sumTo(n)`, которая для данного `n` вычисляет сумму чисел от 1 до `n`, например:

```
sumTo(1) = 1
sumTo(2) = 2 + 1 = 3
sumTo(3) = 3 + 2 + 1 = 6
sumTo(4) = 4 + 3 + 2 + 1 = 10
...
sumTo(100) = 100 + 99 + ... + 2 + 1 = 5050
```

Сделайте три варианта решения:

1. С использованием цикла.
2. Через рекурсию, т.к. $sumTo(n) = n + sumTo(n-1)$ для $n > 1$.
3. С использованием формулы для суммы [арифметической прогрессии](#).

Пример работы вашей функции:

```
function sumTo(n) { /*... ваш код ... */ }
alert( sumTo(100) ); // 5050
```

Какой вариант решения самый быстрый? Самый медленный? Почему?

Можно ли при помощи рекурсии посчитать `sumTo(100000)`? Если нет, то почему?

[К решению](#)

Вычислить факториал

важность: 4

Факториал 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`

[К решению](#)

Числа Фибоначчи

важность: 5

Последовательность [чисел Фибоначчи](#) имеет формулу $F_n = F_{n-1} + F_{n-2}$. 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...

Числа Фибоначчи тесно связаны с [золотым сечением](#) и множеством природных явлений вокруг нас.

Напишите функцию `fib(n)`, которая возвращает `n`-е число Фибоначчи. Пример работы:

```
function fib(n) { /* ваш код */ }

alert( fib(3) ); // 2
alert( fib(7) ); // 13
alert( fib(77) ); // 5527939700884757
```

Все запуски функций из примера выше должны срабатывать быстро.

[К решению](#)

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.

i 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](#).

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.

✔ Задачи

Проверка на NFE

важность: 5

Каков будет результат выполнения кода?

```
function g() { return 1; }
alert(g);
```

А такого? Будет ли разница, если да – почему?

```
(function g() { return 1; });
alert(g);
```

[К решению](#)

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 /* ... */:

Подробнее: [Структура кода](#).

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.

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

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

Подробнее: [Переменные](#), [Шесть типов данных](#), [typeof](#).

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 вернуть 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 );
```

Подробнее: [Взаимодействие с пользователем: alert, prompt, confirm](#).

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 (см. далее).

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( 'a' > 'Я' ); // true
```

Подробнее: [Основные операторы](#), [Операторы сравнения и логические значения](#).

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 [битовыми 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
```

Подробнее: [Логические операторы](#).

Циклы

- Поддерживаются три вида циклов:

```
// 1
while (условие) {
  ...
}

// 2
do {
  ...
} while (условие);

// 3
for (var i = 0; i < 10; i++) {
  ...
}
```

- Переменную можно объявлять прямо в цикле, но видна она будет и за его пределами.
- Поддерживаются директивы `break/continue` для выхода из цикла/перехода на следующую итерацию.

Для выхода одновременно из нескольких уровней цикла можно задать метку.

Синтаксис: « имя_метки: », ставится она только перед циклами и блоками, например:

```
outer:
for(;;) {
```

```

...
for(;;) {
...
break outer;
}
}

```

Переход на метку возможен только изнутри цикла, и только на внешний блок по отношению к данному циклу. В произвольное место программы перейти нельзя.

Подробнее: [Циклы while, for.](#)

Конструкция switch

При сравнениях в конструкции `switch` используется оператор `===`.

Например:

```

var age = prompt('Ваш возраст', 18);

switch (age) {
  case 18:
    alert( 'Никогда не сработает' ); // результат prompt - строка, а не число

  case "18": // вот так - сработает!
    alert( 'Вам 18 лет!' );
    break;

  default:
    alert( 'Любое значение, не совпавшее с case' );
}

```

Подробнее: [Конструкция switch.](#)

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

```

Подробнее: [Функции.](#)

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`.

Детали: [Функциональные выражения](#).

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.

Более развёрнуто: [Именованные функциональные выражения](#).

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

Для того, чтобы код был качественным, необходимы как минимум:

- Умение отладить код и поправить ошибки.
- Хороший стиль кода.
- Тестировать код, желательно – в автоматическом режиме.

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).

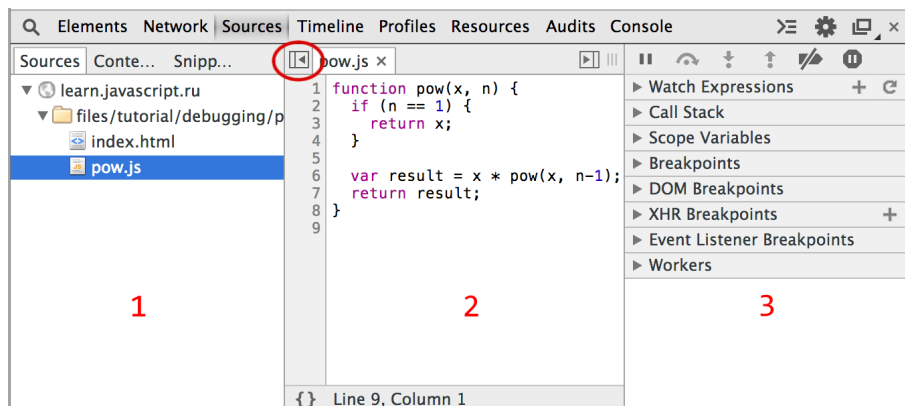
Общий вид панели Sources

В вашей версии Chrome панель может выглядеть несколько по-иному, но что где находится, должно быть понятно.

Зайдите на [страницу с примером](#) браузером Chrome.

Откройте инструменты разработчика: `F12` или в меню Инструменты > Инструменты Разработчика.

Выберите сверху Sources.

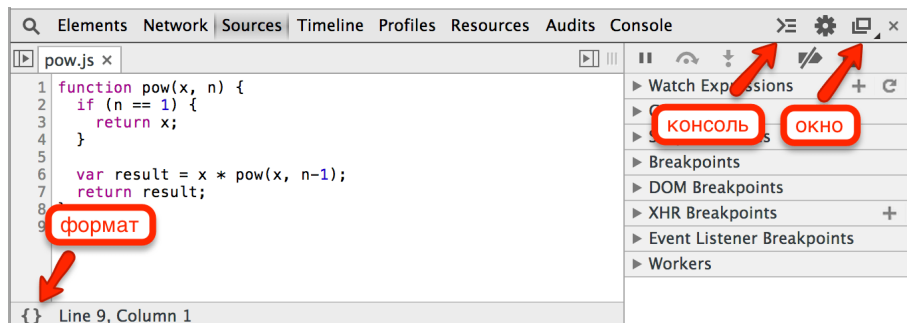


Вы видите три зоны:

1. Зона исходных файлов. В ней находятся все подключённые к странице файлы, включая JS/CSS. Выберите `pow.js`, если он не выбран.
2. Зона текста. В ней находится текст файлов.
3. 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

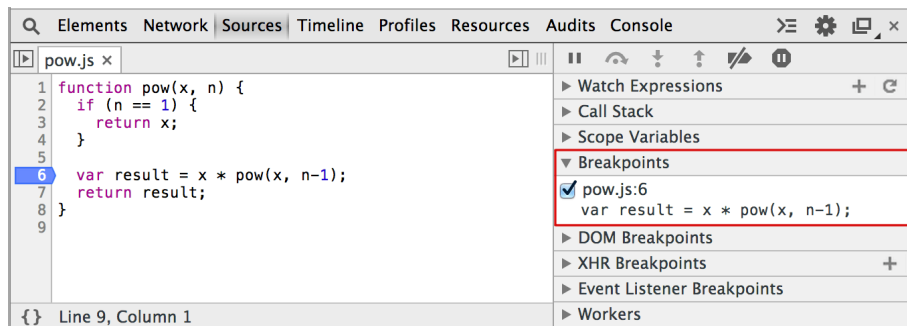
Если код очень большой, то можно вынести инструменты разработки вбок или в отдельное окно, зажав эту кнопку и выбрав соответствующий вариант из списка.

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».

Выглядит 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

- Останов можно инициировать и напрямую из кода скрипта, командой `debugger` :

```
function pow(x, n) {  
  ...  
  debugger; // <-- отладчик остановится тут  
  ...  
}
```

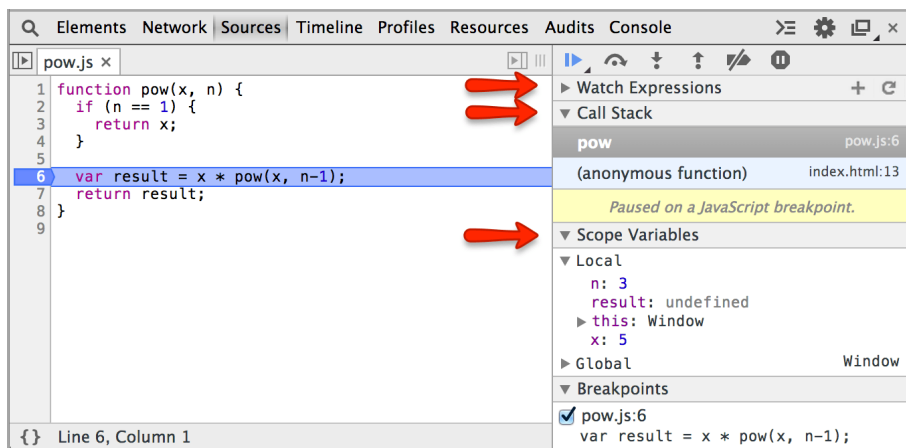
- Правый клик на номер строки `pow.js` позволит создать условную точку останова (conditional breakpoint), т.е. задать условие, при котором точка останова сработает.

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 выполнялась, 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:

 – 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.

 – 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 `row`, так как внутри `row` находится брейкпойнт, а на включённых брейкпойнтах отладчик останавливается всегда.

 – сделать шаг, горячая клавиша **F11**.


Выполняет одну команду скрипта и переходит к следующей. Если есть вложенный вызов, то заходит внутрь функции.

Эта кнопка позволяет подробнее образом пройти по очереди по командам скрипта.


 – 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.

 – отключить/включить все точки останова.

Эта кнопка никак не двигает нас по коду, она позволяет временно отключить все точки останова в файле.

 – 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» сверху-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](#) и [Chrome CommandLine API](#). Почти все команды также действуют в Firebug (отладчик для браузера Firefox).

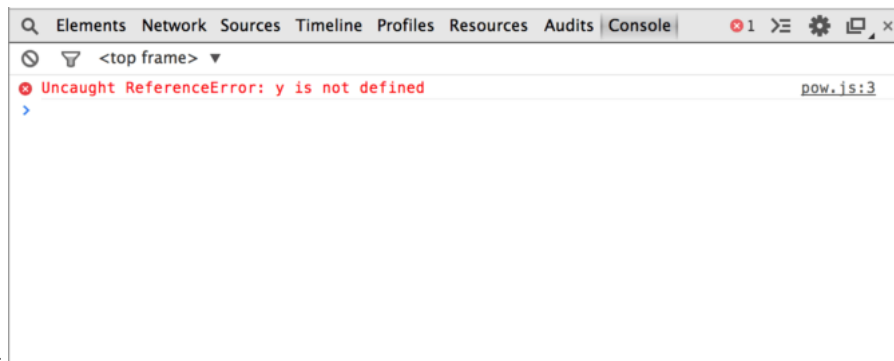
Консоль поддерживают все браузеры, и, хотя IE10- поддерживает далеко не все функции, но `console.log` работает везде. Используйте его для вывода отладочной информации по ходу работы скрипта.

Ошибки

Ошибки JavaScript выводятся в консоли.

Например, прервите отладку – для этого достаточно закрыть инструменты разработчика – и откройте [страницу с ошибкой](#).

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.

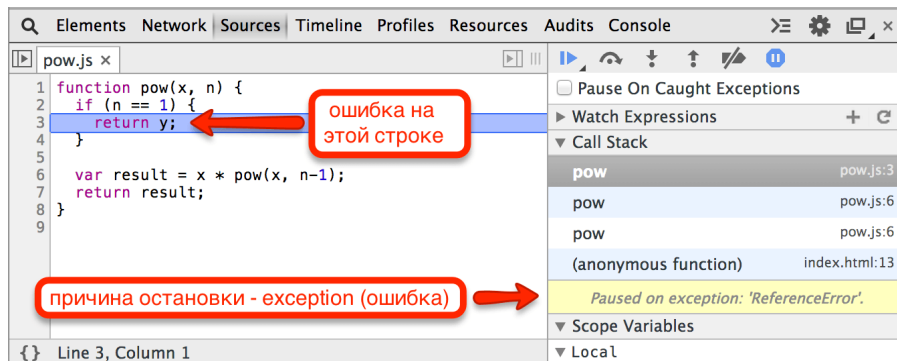
Однако почему она возникла?

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:

1. Перейдите на вкладку Sources.
2. Включите останов при ошибке, кликнув на кнопку 
3. 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.

Осваивать можно двумя путями:

1. [Официальная документация](#) [↗](#) (на англ.)
2. Кликать в разных местах и смотреть, что получается. Не забывать о клике правой кнопкой мыши.

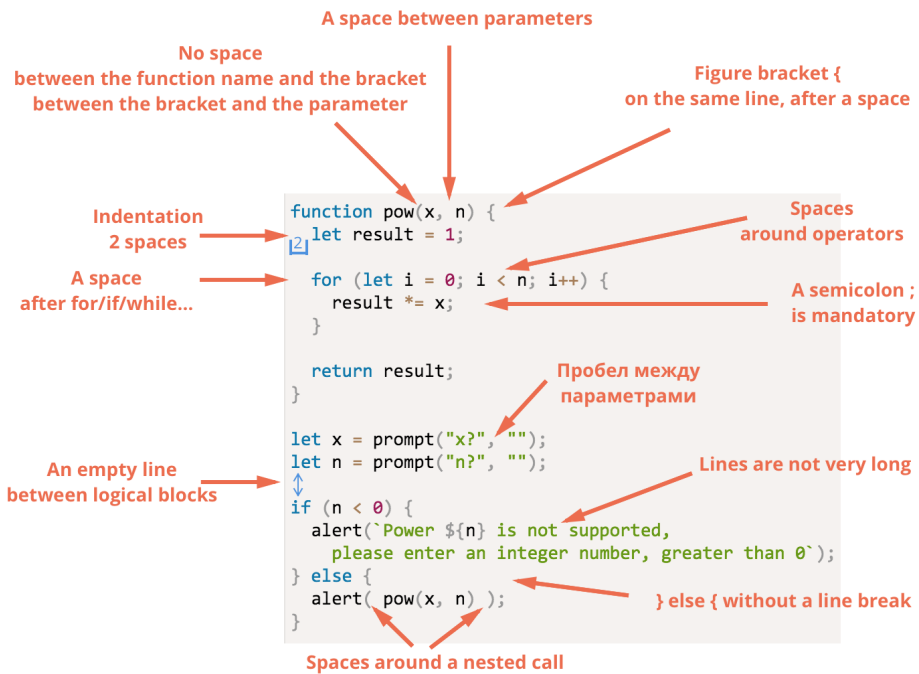
Мы ещё вернёмся к отладчику позже, когда будем работать с HTML.

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.

Beginner sometimes do that. Bad!
Figure brackets are not needed

```
if (n < 0) {alert(`Power ${n} is not supported`);}
```

Split to a separate line without brackets. Never do that!
Source of nasty errors.

```
if (n < 0)
  alert(`Power ${n} is not supported`);
```

One line without brackets.
Acceptable if it's short.

```
if (n < 0) alert(`Power ${n} is not supported`);
```

The best variant.

```
if (n < 0) {
  alert(`Power ${n} is not supported`);
}
```

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, как обсуждалось [ранее](#).

Именованное

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.

Более подробно – читайте про [имена функций](#) и [имена переменных](#).

Уровни вложенности

Уровней вложенности должно быть немного.

Например, [проверки в циклах можно делать через «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.

1. Функции над кодом, который их использует:

```
// объявить функции
function createElement() {
  ...
}

function setHandler(elem) {
  ...
}

function walkAround() {
  ...
}

// код, использующий функции
var elem = createElement();
setHandler(elem);
walkAround();
```

2. 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 идет 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» [↗](#): «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](#) [↗](#), 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](#) [↗](#).

```
/**
 * 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](#) [↗](#) и редакторами от [JetBrains](#) [↗](#), которые учитывают их при автодополнении, а также выводят их в автоподсказках при наборе кода.

Кроме того, есть инструменты, например [JSDoc 3](#) [↗](#), 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, прокомментируйте.

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, и которому, по желанию, всегда можно что-то добавить.

Большинство есть на английском, сообщите мне, если найдёте хороший перевод:

- [Google JavaScript Style Guide](#) ↗
- [jQuery Core Style Guidelines](#) ↗
- [Airbnb JavaScript Style Guide](#) ↗
- [Idiomatic.JS](#) ↗ (есть [перевод](#) ↗)
- [Dojo Style Guide](#) ↗

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](#) ↗ – проверяет код на соответствие [стилю JSLint](#) ↗, в онлайн-интерфейсе вверху можно ввести код, а внизу различные настройки проверки, чтобы сделать её более мягкой.
- [JSHint](#) ↗ – вариант JSLint с большим количеством настроек.
- [Closure Linter](#) ↗ – проверка на соответствие [Google JavaScript Style Guide](#) ↗.

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](#) ↗.

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.

Задачи

Ошибки в стиле

важность: 4

Какие недостатки вы видите в стиле этого примера?

```
function pow(x,n)
{
```

```

var result=1;
for(var i=0;i<n;i++) {result*=x;}
return result;
}

x=prompt("x?","")
n=prompt("n?","")
if (n<0)
{
  alert('Степень '+n+'не поддерживается, введите целую степень, большую 0');
}
else
{
  alert(pow(x,n))
}

```

[К решению](#)

Как писать неподдерживаемый код?

⚠ Познай свой код

Эта статья представляет собой мой вольный перевод [How To Write Unmaintainable Code](#) («как писать неподдерживаемый код») с дополнениями, актуальными для JavaScript.

Возможно, в каких-то из этих советов вам даже удастся узнать «этого парня в зеркале».

Предлагаю вашему вниманию советы мастеров древности, следование которым создаст дополнительные рабочие места для JavaScript-разработчиков.

Если вы будете им следовать, то ваш код будет так сложен в поддержке, что у JavaScript'еров, которые придут после вас, даже простейшее изменение займет годы *оплачиваемого* труда! А сложные задачи оплачиваются хорошо, так что они, определённо, скажут вам «Спасибо».

Более того, *внимательно* следуя этим правилам, вы сохраните и своё рабочее место, так как все будут бояться вашего кода и бежать от него...

...Впрочем, всему своя мера. При написании такого кода он не должен *выглядеть* сложным в поддержке, код должен *быть* таковым.

Явно кривой код может написать любой дурак. Это заметят, и вас уволят, а код будет переписан с нуля. Вы не можете такого допустить. Эти советы учитывают такую возможность. Да здравствует дзен.

Соглашения – по настроению

Рабочий-чистильщик осматривает дом:

"...Вот только жук у вас необычный...

И чтобы с ним справиться, я должен жить как жук, стать жуком, думать как жук."

(грызёт стол Симпсонов)



Сериал "Симпсоны", серия Helter Shelter

Чтобы помешать другому программисту исправить ваш код, вы должны понять путь его мыслей.

Представьте, перед ним – ваш большой скрипт. И ему нужно поправить его. У него нет ни времени ни желания, чтобы читать его целиком, а тем более – досконально разбирать. Он хотел бы по-быстрому найти нужное место, сделать изменение и убраться восвояси без появления побочных эффектов.

Он рассматривает ваш код как бы через трубочку из туалетной бумаги. Это не даёт ему общей картины, он ищет тот небольшой фрагмент, который ему необходимо изменить. По крайней мере, он надеется, что этот фрагмент будет небольшим.

На что он попытается опереться в этом поиске – так это на соглашения, принятые в программировании, об именах переменных, названиях функций и методов...

Как затруднить задачу? Можно везде нарушать соглашения – это мешает ему, но такое могут заметить, и код будет переписан. Как поступил бы ниндзя на вашем месте?

...Правильно! Следуйте соглашениям «в общем», но иногда – нарушайте их.

Тщательно разбросанные по коду нарушения соглашений с одной стороны не делают код явно плохим при первом взгляде, а с другой – имеют в точности тот же, и даже лучший эффект, чем явное неследование им!

Пример из jQuery

⚠ jQuery / DOM

Этот пример требует знаний jQuery/DOM, если пока их у вас нет – пропустите его, ничего страшного, но обязательно вернитесь к нему позже. Подобное стоит многих часов отладки.

Во фреймворке jQuery есть метод [wrap](#), который обёртывает один элемент вокруг другого:

```

var img = $('<img/>'); // создали новые элементы (jQuery-синтаксис)
var div = $('<div/>'); // и поместили в переменную

img.wrap(div); // обернуть img в div
div.append('<span/>');

```

Результат кода после операции `wrap` – два элемента, один вложен в другой:

```
<div>
  <img/>
</div>
```

А что же после `append` ?

Можно предположить, что `` добавится в конец `div` , сразу после `img` ... Но ничего подобного!

Искусный ниндзя уже нанёс свой удар и поведение кода стало неправильным, хотя разработчик об этом даже не подозревает.

Как правило, методы jQuery работают с теми элементами, которые им переданы. Но не здесь!

Внутри вызова `img.wrap(div)` происходит клонирование `div` и вокруг `img` оборачивается не сам `div` , а его клон. При этом исходная переменная `div` не меняется, в ней как был пустой `div` , так и остался.

В итоге, после вызова получается два независимых `div`'а : первый содержит `img` (этот невидный клон никуда не присвоен), а второй – наш `span` .

Объяснения не очень понятны? Написано что-то странное? Это просто разум, привыкший, что соглашения уважаются, не допускает мысли, что вызов `wrap` – невидно клонирует элемент. Ведь другие jQuery-методы, кроме `clone` этого не делают.

Как говорил [Учитель](#) : «В древности люди учились для того, чтобы совершенствовать себя. Нынче учатся для того, чтобы удивить других».

Краткость – сестра таланта!

Пишите «как короче», а не как понятнее. «Меньше букв» – уважительная причина для нарушения любых соглашений.

Ваш верный помощник – возможности языка, использованные неочевидным образом.

Обратите внимание на оператор вопросительный знак `'?'` , например:

```
// код из jQuery
i = i ? i < 0 ? Math.max(0, len + i) : i : 0;
```

Разработчик, встретивший эту строку и попытавшийся понять, чему же всё-таки равно `i` , скорее всего придёт к вам за разъяснениями. Смело скажите ему, что короче – это всегда лучше. Посвятите и его в пути ниндзя. Не забудьте вручить [Дао дэ цзин](#) .

Именованье

Существенную часть науки о создании неподдерживаемого кода занимает искусство выбора имён.

Однбуквенные переменные

Называйте переменные коротко: `a` , `b` или `c` .

В этом случае никто не сможет найти её, используя функцию «Поиск» текстового редактора.

Более того, даже найдя – никто не сможет «расшифровать» её и догадаться, что она означает.

Не используйте `i` для цикла

В тех местах, где однбуквенные переменные общеприняты, например, в счетчике цикла – ни в коем случае не используйте стандартные названия `i` , `j` , `k` . Где угодно, только не здесь!

Остановите свой взыскательный взгляд на чём-нибудь более экзотическом. Например, `x` или `y` .

Эффективность этого подхода особенно заметна, если тело цикла занимает одну-две страницы (чем длиннее – тем лучше).

В этом случае заметить, что переменная – счетчик цикла, без пролистывания вверх, невозможно.

Русские слова и сокращения

Если вам *приходится* использовать длинные, понятные имена переменных – что поделать... Но и здесь есть простор для творчества!

Назовите переменные «калькой» с русского языка или как-то «улучшите» английское слово.

В одном месте напишите `var ssilka` , в другом `var ssylka` , в третьем `var link` , в четвёртом – `var lnk` ... Это действительно великолепно работает и очень креативно!

Количество ошибок при поддержке такого кода увеличивается во много раз.

Будьте абстрактны при выборе имени

*Лучший кувшин лепят всю жизнь.
Высокая музыка неподвластна слуху.
Великий образ не имеет формы.*

“ Лао-цзы

При выборе имени старайтесь применить максимально абстрактное слово, например `obj` , `data` , `value` , `item` , `elem` и т.п.

- Идеальное имя для переменной: `data` . Используйте это имя везде, где можно. В конце концов, каждая переменная содержит *данные* , не правда ли?

Но что делать, если имя `data` уже занято? Попробуйте `value` , оно не менее универсально. Ведь каждая переменная содержит *значение* .

Занято и это? Есть и другой вариант.

- Называйте переменную по типу данных, которые она хранит: `obj` , `num` , `arr` ...

Насколько это усложнит разработку? Как ни странно, намного!

Казалось бы, название переменной содержит информацию, говорит о том, что в переменной – число, объект или массив... С другой стороны, когда непосвящённый будет разбирать этот код – он с удивлением обнаружит, что информации нет!

Ведь как раз тип легко понять, запустив отладчик и посмотрев, что внутри. Но в чём смысл этой переменной? Что за массив/объект/число в ней хранится? Без долгой медитации над кодом тут не обойтись!

- Что делать, если и эти имена кончились? Просто добавьте цифры: `item1`, `item2`, `elem5`, `data1` ...

Похожие имена

Только истинно внимательный программист достоин понять ваш код. Но как проверить, достоин ли читающий?

Один из способов – использовать похожие имена переменных, например `data` и `date`. Бегло прочитать такой код почти невозможно. А уж заметить опечатку и поправить её... Ммммм... Мы здесь надолго, время попить чайку.

А.К.Р.О.Н.И.М

Используйте сокращения, чтобы сделать код короче.

Например `ie` (Inner Element), `mc` (Money Counter) и другие. Если вы обнаружите, что путаетесь в них сами – героически страдайте, но не переписывайте код. Вы знали, на что шли.

Хитрые синонимы

Очень трудно найти чёрную кошку в тёмной комнате, особенно когда её там нет.



Конфуций

Чтобы было не скучно – используйте *похожие названия* для обозначения *одинаковых действий*.

Например, если метод показывает что-то на экране – начните его название с `display..` (скажем, `displayElement`), а в другом месте объявите аналогичный метод как `show..` (`showFrame`).

Как бы намекайте этим, что существует тонкое различие между способами показа в этих методах, хотя на самом деле его нет.

По возможности, договоритесь с членами своей команды. Если Вася в своих классах использует `display..`, то Валера – обязательно `render..`, а Петя – `paint..`

...И напротив, если есть две функции с важными отличиями – используйте одно и то же слово для их описания! Например, с `print...` можно начать метод печати на принтере `printPage`, а также – метод добавления текста на страницу `printText`.

А теперь, пусть читающий код думает: «Куда же выводит сообщение `printMessage`?». Особый шик – добавить элемент неожиданности. Пусть `printMessage` выводит не туда, куда все, а в новое окно!

Словарь терминов – это еда!

Ни в коем случае не поддавайтесь требованиям написать словарь терминов для проекта. Если же он уже есть – не следуйте ему, а лучше проглотите и скажите, что так и было!

Пусть читающий ваш код программист напрасно ищет различия в `helloUser` и `welcomeVisitor` и пытается понять, когда что использовать. Вы-то знаете, что на самом деле различий нет, но искать их можно о-очень долго.

Для обозначения посетителя в одном месте используйте `user`, а в другом `visitor`, в третьем – просто `u`. Выбирайте одно имя или другое, в зависимости от функции и настроения.

Это воплотит сразу два ключевых принципа ниндзя-дизайна – *сокрытие информации* и *подмена понятий*!

Повторно используйте имена

По возможности, повторно используйте имена переменных, функций и свойств. Просто записывайте в них новые значения.

Добавляйте новое имя только если это абсолютно необходимо.

В функции старайтесь обойтись только теми переменными, которые были переданы как параметры.

Это не только затруднит идентификацию того, что *сейчас* находится в переменной, но и сделает почти невозможным поиск места, в котором конкретное значение было присвоено.

Цель – максимально усложнить отладку и заставить читающего код программиста построчно анализировать код и конспектировать изменения переменных для каждой ветки исполнения.

Продвинутый вариант этого подхода – незаметно (!) подменить переменную на нечто похожее, например:

```
function ninjaFunction(elem) {
  // 20 строк кода, работающего с elem

  elem = elem.cloneNode(true);

  // еще 20 строк кода, работающего с elem
}
```

Программист, пожелавший добавить действия с `elem` во вторую часть функции, будет удивлён. Лишь во время отладки, посмотрев весь код, он с удивлением обнаружит, что оказывается имел дело с клоном!

Регулярные встречи с этим приемом на практике говорят: защититься невозможно. Эффективно даже против опытного ниндзи.

Добавляйте подчеркивания

Добавляйте подчеркивания `_` и `__` к именам переменных. Желательно, чтобы их смысл был известен только вам, а лучше – вообще без явной причины.

Этим вы достигните двух целей. Во-первых, код станет длиннее и менее читаемым, а во-вторых, другой программист будет долго искать смысл в подчёркиваниях. Особенно хорошо сработает и внесет сумятицу в его мысли, если в некоторых частях проекта подчеркивания будут, а в некоторых – нет.

В процессе развития кода вы, скорее всего, будете путаться и смешивать стили: добавлять имена с подчеркиваниями там, где обычно подчеркиваний нет, и наоборот. Это нормально и полностью соответствует третьей цели – увеличить количество ошибок при внесении исправлений.

Покажите вашу любовь к разработке

Пусть все видят, какими замечательными сущностями вы оперируете! Имена `superElement`, `megaFrame` и `niceItem` при благоприятном положении звёзд могут привести к просветлению читающего.

Действительно, с одной стороны, кое-что написано: `super..`, `mega..`, `nice..`. С другой – это не несёт никакой конкретики. Читающий может решить поискать в этом глубинный смысл и замедлится на часок-другой оплаченного рабочего времени.

Перекрывайте внешние переменные

*Находясь на свету, нельзя ничего увидеть в темноте.
Пребывая же в темноте, увидишь все, что находится на свету.*

“ Гуань Инь-цзы

Почему бы не использовать одинаковые переменные внутри и снаружи функции? Это просто и не требует придумывать новых имён.

```
var user = authenticateUser();

function render() {
  var user = anotherValue();
  ...
  ...многобукв...
  ...
  ... // <-- программист захочет внести исправления сюда, и..
  ...
}
```

Зашедший в середину метода `render` программист, скорее всего, не заметит, что переменная `user` локально перекрыта и попытается работать с ней, полагая, что это результат `authenticateUser()` ... Ловушка захлопнулась! Здравствуй, отладчик.

Мощные функции!

Не ограничивайте действия функции тем, что написано в её названии. Будьте шире.

Например, функция `validateEmail(email)` может, кроме проверки e-mail на правильность, выводить сообщение об ошибке и просить заново ввести e-mail.

Выберите хотя бы пару дополнительных действий, кроме основного назначения функции.

Главное – они должны быть неочевидны из названия функции. Истинный ниндзя-девелопер сделает так, что они будут неочевидны и из кода тоже.

Объединение нескольких смежных действий в одну функцию защитит ваш код от повторного использования.

Представьте, что другому разработчику нужно только проверить адрес, а сообщение – не выводить. Ваша функция `validateEmail(email)`, которая делает и то и другое, ему не подойдёт. Работодатель будет вынужден оплатить создание новой.

Внимание... Сюр-при-из!

Есть функции, название которых говорит о том, что они ничего не меняют. Например, `isReady`, `checkPermission`, `findTags` ... Предполагается, что при вызове они произведут некие вычисления, или найдут и возвратят полезные данные, но при этом их не изменят. В трактатах это называется «отсутствие сторонних эффектов».

По-настоящему красивый приём – делать в таких функциях что-нибудь полезное, заодно с процессом проверки. Что именно – совершенно неважно.

Удивление и ошеломление, которое возникнет у вашего коллеги, когда он увидит, что функция с названием на `is..`, `check..` или `find...` что-то меняет – несомненно, расширит его границы разумного!

Ещё одна вариация такого подхода – возвращать нестандартное значение.

Ведь общеизвестно, что `is..` и `check..` обычно возвращают `true/false`. Прдемонстрируйте оригинальное мышление. Пусть вызов `checkPermission` возвращает не результат `true/false`, а объект с результатами проверки! А чего, полезно.

Те же разработчики, кто попытается написать проверку `if (checkPermission(..))`, будут весьма удивлены результатом. Ответьте им: «надо читать документацию!». И перешлите эту статью.

Заключение

Все советы выше пришли из реального кода... И в том числе от разработчиков с большим опытом.

Возможно, даже больше вашего, так что не судите опрометчиво ;)

- Следуйте нескольким из них – и ваш код станет полон сюрпризов.
- Следуйте многим – и ваш код станет истинно вашим, никто не захочет изменять его.
- Следуйте всем – и ваш код станет ценным уроком для молодых разработчиков, ищущих просветления.

Автоматические тесты при помощи chai и mocha

В этой главе мы разберём основы автоматического тестирования. Оно будет применяться далее в задачах, и вообще, входит в «образовательный минимум» программиста.

Зачем нужны тесты?

При написании функции мы обычно представляем, что она должна делать, какое значение на каких аргументах выдавать.

В процессе разработки мы время от времени проверяем, правильно ли работает функция. Самый простой способ проверить – это запустить её, например в консоли, и посмотреть результат.

Если что-то не так, поправить, опять запустить – посмотреть результат... И так «до победного конца».

Но такие ручные запуски – очень несовершенное средство проверки.

Когда проверяешь работу кода вручную – легко его «нетестировать».

Например, пишем функцию `f`. Написали, тестируем с разными аргументами. Вызов функции `f(a)` работает, а вот `f(b)` не работает. Поправили код – стало работать `f(b)`, вроде закончили. Но при этом забыли заново протестировать `f(a)` – упс, вот и возможная ошибка в коде.

Автоматизированное тестирование – это когда тесты написаны отдельно от кода, и можно в любой момент запустить их и проверить все важные случаи использования.

BDD – поведенческие тесты кода

Мы рассмотрим методику тестирования, которая входит в [BDD](#) – Behavior Driven Development. Подход BDD давно и с успехом используется во многих проектах.

BDD – это не просто тесты. Это гораздо больше.

Тесты BDD – это три в одном: И тесты, И документация, И примеры использования.

Впрочем, хватит слов. Рассмотрим примеры.

Разработка pow: спецификация

Допустим, мы хотим разработать функцию `pow(x, n)`, которая возводит `x` в целую степень `n`, для простоты $n \geq 0$.

Ещё до разработки мы можем представить себе, что эта функция будет делать, и описать это по методике BDD.

Это описание называется *спецификация* (или, как говорят в обиходе, «спека») и выглядит так:

```
describe("pow", function() {
  it("возводит в n-ю степень", function() {
    assert.equal(pow(2, 3), 8);
  });
});
```

У спецификации есть три основных строительных блока, которые вы видите в примере выше:

```
describe(название, function() { ... })
```

Задаёт, что именно мы описываем, используется для группировки «рабочих лошадок» – блоков `it`. В данном случае мы описываем функцию `pow`.

```
it(название, function() { ... })
```

В названии блока `it` *человеческим языком* описывается, что должна делать функция, далее следует *тест*, который проверяет это.

```
assert.equal(value1, value2)
```

Код внутри `it`, если реализация верна, должен выполняться без ошибок.

Различные функции вида `assert.*` используются, чтобы проверить, делает ли `pow` то, что задумано. Пока что нас интересует только одна из них – `assert.equal`, она сравнивает свой первый аргумент со вторым и выдаёт ошибку в случае, когда они не равны. В данном случае она проверяет, что результат `pow(2, 3)` равен 8.

Есть и другие виды сравнений и проверок, которые мы увидим далее.

Поток разработки

Как правило, поток разработки таков:

1. Пишется спецификация, которая описывает самый базовый функционал.
2. Делается начальная реализация.
3. Для проверки соответствия спецификации мы задействуем фреймворк (в нашем случае [Mocha](#)). Фреймворк запускает все тесты `it` и выводит ошибки, если они возникнут. При ошибках вносятся исправления.
4. Спецификация расширяется, в неё добавляются возможности, которые пока, возможно, не поддерживаются реализацией.
5. Идём на пункт 2, делаем реализацию. И так «до победного конца».

Разработка ведётся *итеративно*: один проход за другим, пока спецификация и реализация не будут завершены.

В нашем случае первый шаг уже завершён, начальная спецификация готова, хорошо бы приступить к реализации. Но перед этим проведём «нулевой» запуск спецификации, просто чтобы увидеть, что уже в таком виде, даже без реализации – тесты работают.

Пример в действии

Для запуска тестов нужны соответствующие JavaScript-библиотеки.

Мы будем использовать:

- [Mocha](#) – эта библиотека содержит общие функции для тестирования, включая `describe` и `it`.
- [Chai](#) – библиотека поддерживает разнообразные функции для проверок. Есть разные «стили» проверки результатов, с которыми мы познакомимся позже, на текущий момент мы будем использовать лишь `assert.equal`.
- [Sinon](#) – для эмуляции и хитрой подмены функций «заглушками», понадобится позднее.

Эти библиотеки позволяют тестировать JS не только в браузере, но и на сервере Node.JS. Здесь мы рассмотрим браузерный вариант, серверный использует те же функции.

Пример HTML-страницы для тестов:

```
<!DOCTYPE html>
<html>
<head>
  <!-- add mocha css, to show results -->
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.css">
  <!-- add mocha framework code -->
  <script src="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.js"></script>
  <script>
    // enable bdd-style testing
    mocha.setup('bdd');
  </script>
  <!-- add chai -->
  <script src="https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js"></script>
  <script>
    // chai has a lot of stuff, let's make assert global
    let assert = chai.assert;
  </script>
</head>

<body>

  <script>
    function pow(x, n) {
      /* function code is to be written, empty now */
    }
  </script>

  <!-- the script with tests (describe, it...) -->
  <script src="test.js"></script>

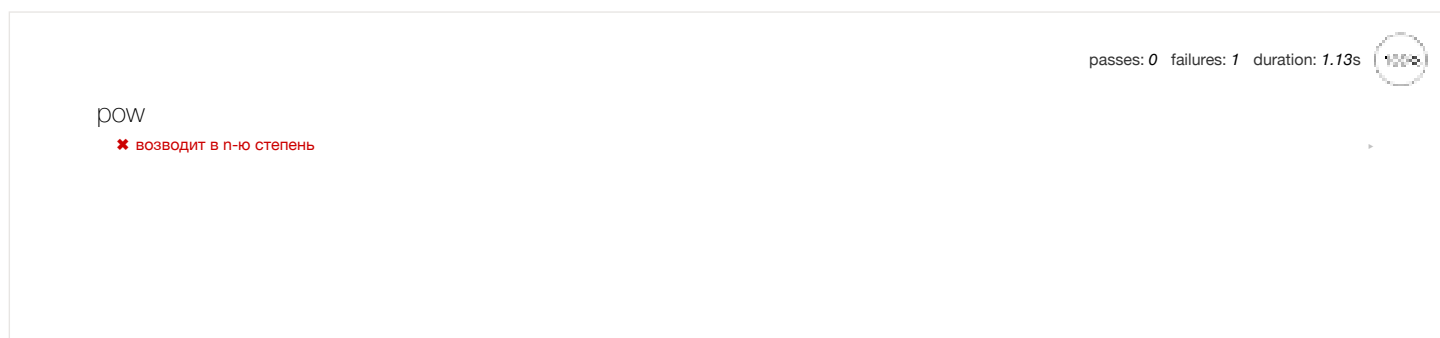
  <!-- the element with id="mocha" will contain test results -->
  <div id="mocha"></div>

  <!-- run tests! -->
  <script>
    mocha.run();
  </script>
</body>
</html>
```

Эту страницу можно условно разделить на четыре части:

1. Блок `<head>` – в нём мы подключаем библиотеки и стили для тестирования, нашего кода там нет.
2. Блок `<script>` с реализацией спецификации, в нашем случае – с кодом для `pow`.
3. Далее подключаются тесты, файл `test.js` содержит `describe("pow", ...)`, который был описан выше. Методы `describe` и `it` принадлежат библиотеке Mocha, так что важно, что она была подключена выше.
4. Элемент `<div id="mocha">` будет использоваться библиотекой Mocha для вывода результатов. Запуск тестов инициируется командой `mocha.run()`.

Результат срабатывания:



passes: 0 failures: 1 duration: 1.13s

pow

✖ возводит в n-ю степень

Пока что тесты не проходят, но это логично – вместо функции стоит «заглушка», пустой код.

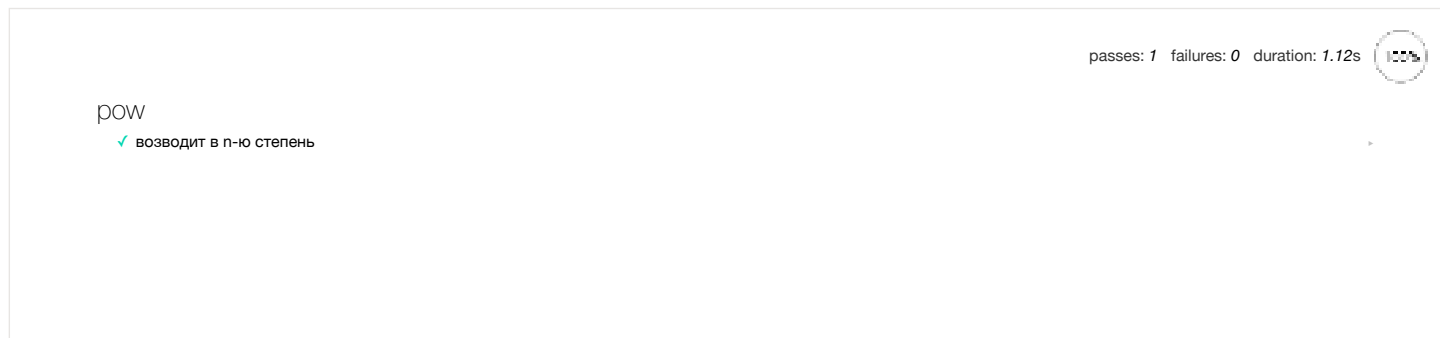
Пока что у нас одна функция и одна спецификация, но на будущее заметим, что если различных функций и тестов много – это не проблема, можно их все подключить на одной странице. Конфликта не будет, так как каждый функционал имеет свой блок `describe("что тестируем" ...)`. Сами же тесты обычно пишутся так, чтобы не влиять друг на друга, не делать лишних глобальных переменных.

Начальная реализация

Пока что, как видно, тесты не проходят, ошибка сразу же. Давайте сделаем минимальную реализацию `pow`, которая бы работала нормально:

```
function pow() {  
  return 8; // :) мы - мошенники!  
}
```

О, вот теперь работает:



Исправление спецификации

Функция, конечно, ещё не готова, но тесты проходят. Это ненадолго :)

Здесь мы видим ситуацию, которая (и не обязательно при ленивом программисте!) бывает на практике – да, есть тесты, они проходят, но функция (увы!) работает неправильно.

С точки зрения BDD, ошибка при проходящих тестах – вина спецификации.

В первую очередь не реализация исправляется, а уточняется спецификация, пишется падающий тест.

Сейчас мы расширим спецификацию, добавив проверку на `pow(3, 4) = 81`.

Здесь есть два варианта организации кода:

1. Первый вариант – добавить `assert` в тот же `it`:

```
describe("pow", function() {  
  it("возводит в n-ю степень", function() {  
    assert.equal(pow(2, 3), 8);  
    assert.equal(pow(3, 4), 81);  
  });  
});
```

2. Второй вариант – сделать два теста:

```
describe("pow", function() {  
  it("при возведении 2 в 3ю степень результат 8", function() {  
    assert.equal(pow(2, 3), 8);  
  });  
  it("при возведении 3 в 4ю степень равен 81", function() {  
    assert.equal(pow(3, 4), 81);  
  });  
});
```

Их принципиальное различие в том, что если `assert` обнаруживает ошибку, то он тут же прекращает выполнение блока `it`. Поэтому в первом варианте, если вдруг первый `assert` «провалился», то про результат второго мы никогда не узнаем.

Таким образом, разделить эти тесты может быть полезно, чтобы мы получили больше информации о происходящем.

Кроме того, есть ещё одно правило, которое желательно соблюдать.

Один тест тестирует ровно одну вещь.

Если мы явно видим, что тест включает в себя совершенно независимые проверки – лучше разбить его на два более простых и наглядных.

По этим причинам второй вариант здесь предпочтительнее.

Результат:

passes: 1 failures: 1 duration: 1.12s



pow

- ✓ при возведении 2 в 3ю степень результат 8
- ✗ при возведении 3 в 4ю степень равен 81

Как и следовало ожидать, второй тест не проходит. Ещё бы, ведь функция всё время возвращает 8 .

Уточнение реализации

Придётся написать нечто более реальное, чтобы тесты проходили:

```
function pow(x, n) {
  var result = 1;

  for (var i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}
```

Чтобы быть уверенными, что функция работает верно, желательно протестировать её на большем количестве значений. Вместо того, чтобы писать блоки `it` вручную, мы можем сгенерировать тесты в цикле `for` :

```
describe("pow", function() {
  function makeTest(x) {
    var expected = x * x * x;
    it("при возведении " + x + " в степень 3 результат: " + expected, function() {
      assert.equal(pow(x, 3), expected);
    });
  }

  for (var x = 1; x <= 5; x++) {
    makeTest(x);
  }
});
```

Результат:

passes: 5 failures: 0 duration: 1.12s



pow

- ✓ при возведении 1 в степень 3 результат: 1
- ✓ при возведении 2 в степень 3 результат: 8
- ✓ при возведении 3 в степень 3 результат: 27
- ✓ при возведении 4 в степень 3 результат: 64
- ✓ при возведении 5 в степень 3 результат: 125

Вложенный describe

Функция `makeTest` и цикл `for` , очевидно, нужны друг другу, но не нужны для других тестов, которые мы добавим в дальнейшем. Они образуют единую группу, задача которой – проверить возведение в `n`-ю степень.


Будет правильно выделить их, при помощи вложенного блока `describe` :

```
describe("pow", function() {
  describe("возводит x в степень n", function() {
    function makeTest(x) {
      var expected = x * x * x;
      it("при возведении " + x + " в степень 3 результат: " + expected, function() {
        assert.equal(pow(x, 3), expected);
      });
    }

    for (var x = 1; x <= 5; x++) {
      makeTest(x);
    }
  });
});
```

```
});  
// ... дальнейшие тесты it и подблоки describe ...  
});
```

Вложенный describe объявит новую «подгруппу» тестов, блоки it которой запускаются так же, как и обычно, но выводятся с подзаголовком, вот так:

passes: 5 failures: 0 duration: 1.12s 

pow

ВОЗВОДИТ X В СТЕПЕНЬ N

- ✓ при возведении 1 в степень 3 результат: 1
- ✓ при возведении 2 в степень 3 результат: 8
- ✓ при возведении 3 в степень 3 результат: 27
- ✓ при возведении 4 в степень 3 результат: 64
- ✓ при возведении 5 в степень 3 результат: 125

В будущем мы сможем добавить другие тесты it и блоки describe со своими вспомогательными функциями.

before/after и beforeEach/afterEach

В каждом блоке describe можно также задать функции before/after, которые будут выполнены до/после запуска тестов, а также beforeEach/afterEach, которые выполняются до/после каждого it.

Например:

```
describe("Тест", function() {  
  before(function() { alert("Начало тестов"); });  
  after(function() { alert("Конец тестов"); });  
  
  beforeEach(function() { alert("Вход в тест"); });  
  afterEach(function() { alert("Выход из теста"); });  
  
  it('тест 1', function() { alert('1'); });  
  it('тест 2', function() { alert('2'); });  
  
});
```

Последовательность будет такой:

```
Начало тестов  
Вход в тест  
1  
Выход из теста  
Вход в тест  
2  
Выход из теста  
Конец тестов
```

[Открыть пример с тестами в песочнице](#)

Как правило, beforeEach/afterEach (before/after) используют, если необходимо произвести инициализацию, обнулить счётчики или сделать что-то ещё в таком духе между тестами (или их группами).

Расширение спецификации

Базовый функционал pow описан и реализован, первая итерация разработки завершена. Теперь расширим и уточним его.

Как говорилось ранее, функция pow(x, n) предназначена для работы с целыми неотрицательными n.

В JavaScript для ошибки вычислений служит специальное значение NaN, которое функция будет возвращать при некорректных n.

Добавим это поведение в спецификацию:

```
describe("pow", function() {  
  // ...  
  
  it("при возведении в отрицательную степень результат NaN", function() {  
    assert(isNaN(pow(2, -1)));  
  });  
  
  it("при возведении в дробную степень результат NaN", function() {  
    assert(isNaN(pow(2, 1.5)));  
  });  
  
});
```




pow

* при возведении в отрицательную степень результат NaN

* при возведении в дробную степень результат NaN

В коде тестов выше можно было бы добавить описание и к `assert.equal`, указав в конце: `assert.equal(value1, value2, "описание")`, но с равенством обычно и так всё понятно, поэтому мы так делать не будем.

Итого

Итак, разработка завершена, мы получили полноценную спецификацию и код, который её реализует.

Задачи выше позволяют дополнить её, и в результате может получиться что-то в таком духе:

```
describe("pow", function() {
  describe("raises x to power n", function() {
    function makeTest(x) {
      let expected = x * x * x;
      it(`${x} in the power 3 is ${expected}`, function() {
        assert.equal(pow(x, 3), expected);
      });
    }
    for (let x = 1; x <= 5; x++) {
      makeTest(x);
    }
  });
  it("if n is negative, the result is NaN", function() {
    assert.isNaN(pow(2, -1));
  });
  it("if n is not integer, the result is NaN", function() {
    assert.isNaN(pow(2, 1.5));
  });
});
```

[Открыть полный пример с реализацией в песочнице](#)

Эту спецификацию можно использовать как:

1. Тесты, которые гарантируют правильность работы кода.
2. Документацию по функции, что она конкретно делает.
3. Примеры использования функции, которые демонстрируют её работу внутри `it`.

Имея спецификацию, мы можем улучшать, менять, переписывать функцию и легко контролировать её работу, просматривая тесты.

Особенно важно это в больших проектах.

Бывает так, что изменение в одной части кода может повлечь за собой «падение» другой части, которая её использует. Так как всё-всё в большом проекте руками не перепроверишь, то такие ошибки имеют большой шанс остаться в продукте и вылезти позже, когда проект увидит посетитель или заказчик.

Чтобы избежать таких проблем, бывает, что вообще стараются не трогать код, от которого много что зависит, даже если его ну очень нужно переписать. Жизнь пробивается тонкими росточками там, где должен цвести и пахнуть новый функционал.

Код, покрытый автотестами, являет собой полную противоположность этому!

Даже если какое-то изменение потенциально может порушить всё – его совершенно не страшно сделать. Ведь есть масса тестов, которые быстро и в автоматическом режиме проверяют работу кода. И если что-то падает, то это можно будет легко локализовать и поправить.

Кроме того, код, покрытый тестами, имеет лучшую архитектуру.

Конечно, это естественное следствие того, что его легче менять и улучшать. Но не только.

Чтобы написать тесты, нужно разбить код на функции так, чтобы для каждой функции было чётко понятно, что она получает на вход, что делает с этим и что возвращает. Это означает ясную и понятную структуру с самого начала.

Конечно, в реальной жизни всё не так просто. Зачастую написать тест сложно. Или сложно поддерживать тесты, поскольку код активно меняется. Сами тесты тоже пишутся по-разному, при помощи разных инструментов.

Что дальше?

В дальнейшем условия ряда задач будут уже содержать в себе тесты. На них вы познакомитесь с дополнительными примерами.

Как правило, они будут вполне понятны, даже если немного выходят за пределы этой главы.

✔ Задачи

Сделать row по спецификации

важность: 5

Исправьте код функции `row`, чтобы тесты проходили.

Для этого ниже в задаче вы найдёте ссылку на песочницу.

Она содержит HTML с тестами. Обратите внимание, что HTML-страница в ней короче той, что обсуждалась в статье [Автоматические тесты при помощи chai и mocha](#). Это потому что библиотеки Chai, Mocha и Sinon объединены в один файл:

```
<script src="https://js.cx/test/libs.js"></script>
```

Этот файл содержит код библиотек, стили, настройки для них и запуск `mocha.run` по окончании загрузки страницы. Если нет элемента с `id="mocha"`, то результаты выводятся в `<body>`.

Сборка сделана исключительно для более компактного представления задач, без рекомендаций использовать именно её в проектах.

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

Добавьте тест к задаче

важность: 5

Добавьте к [предыдущей задаче](#) тесты, которые будут проверять, что любое число, кроме нуля, в нулевой степени равно `1`, а ноль в нулевой степени даёт `NaN` (это математически корректно, результат 0^0 не определён).

При необходимости, исправьте саму функцию `row()`, чтобы тесты проходили без ошибок.

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

Что не так в тесте?

важность: 5

Что не так в этом тесте функции `row`?

```
it("Возводит x в степень n", function() {
  var x = 5;

  var result = x;
  assert.equal(pow(x, 1), result);

  result *= x;
  assert.equal(pow(x, 2), result);

  result *= x;
  assert.equal(pow(x, 3), result);
});
```

P.S. Синтаксически он верен и работает, но спроектирован неправильно.

[К решению](#)

Структуры данных

Изучаем JavaScript: расширенное знакомство со встроенными типами данных, их особенностями.

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
```

Более подробно с различными свойствами и методами строк мы познакомимся в главе [Строки](#).

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"
```

Детали работы `toFixed` разобраны в главе [Числа](#).

⚠️ 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.

Числа

Все числа в JavaScript, как целые так и дробные, имеют тип `Number` и хранятся в 64-битном формате [IEEE-754](#), также известном как «double precision».

Здесь мы рассмотрим различные тонкости, связанные с работой с числами в JavaScript.

Способы записи

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

```
alert( 0xFF ); // 255 в шестнадцатеричной системе
```

Также доступна запись в «*научном формате*» (ещё говорят «запись с плавающей точкой»), который выглядит как `<число>e<кол-во 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
```

Деление на ноль, Infinity

Представьте, что вы собираетесь создать новый язык... Люди будут называть его «JavaScript» (или «LiveScript»... неважно).

Что должно происходить при попытке деления на ноль?

Как правило, ошибка в программе... Во всяком случае, в большинстве языков программирования это именно так.

Но создатель JavaScript решил пойти математически правильным путем. Ведь чем меньше делитель, тем больше результат. При делении на очень-очень маленькое число должно получиться очень большое. В математическом анализе это описывается через [пределы](#), и если подразумевать предел, то в качестве результата деления на `0` мы получаем «бесконечность», которая обозначается символом ∞ или, в JavaScript: "Infinity".

```
alert( 1 / 0 ); // Infinity  
alert( 12345 / 0 ); // 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
```

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)`.

Математические операции в JS безопасны

Никакие математические операции в JavaScript не могут привести к ошибке или «обрушить» программу.

В худшем случае, результат будет `NaN`.

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
```

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 шестнадцатичной и других системах счисления:

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

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`, т.е. 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 `' '`, т.к. 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, т.е. `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-чной, 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;
alert( 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.

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
alert( Math.ceil(3.1) ); // 4
alert( Math.round(3.1) ); // 3
```

i 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. Почему? Скоро узнаем!

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, т.к. числа $0.1 \cdot 10 = 1$ и $0.2 \cdot 10 = 2$ могут быть точно представлены в двоичной системе.

2. 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».

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.

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](#) ↗.

Math.sin(x)

Вычисляет синус x (в радианах)

Math.cos(x)

Вычисляет косинус x (в радианах)

Math.tan(x)

Возвращает тангенс x (в радианах)

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 x^{exp} , например $\text{Math.pow}(2,3) = 8$. Работает в том числе с дробными и отрицательными степенями, например: $\text{Math.pow}(4, -1/2) = 0.5$.

Math.abs(x)

Возвращает абсолютное значение числа

Math.exp(x)

Возвращает e^x , 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 псевдо-случайное число в интервале [0,1) – то есть между 0(включительно) и 1(не включая). Генератор случайных чисел инициализируется текущим временем.

Форматирование

Для красивого вывода чисел в стандарте [ECMA 402](#) 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](#)). Он также умеет форматировать валюту и проценты. Более подробно про устройство этого метода можно будет узнать в статье [Intl: интернационализация в JavaScript](#), когда это вам понадобится.

Итого

- Числа могут быть записаны в шестнадцатичной, восьмеричной системе, 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 10^p .
- 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](#) и [Math](#).

✔ Задачи

Интерфейс суммы

важность: 5

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

[Запустить демо](#)

P.S. Есть «подводный камень» при работе с типами.

[К решению](#)

Почему `6.35.toFixed(1) == 6.3`?

важность: 4

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
```

[К решению](#)

Сложение цен

важность: 5

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?

[К решению](#)

Бесконечный цикл по ошибке

важность: 4

Этот цикл – бесконечный. Почему?

```
var i = 0;
while (i != 10) {
  i += 0.2;
}
```

[К решению](#)

Как получить дробную часть числа?

важность: 4

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
```

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Формула Бине

важность: 4

Последовательность [чисел Фибоначчи](#) имеет формулу $F_n = F_{n-1} + F_{n-2}$. 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](#)):

```
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](#), согласно которой F_n равно ближайшему целому для $\varphi^n/\sqrt{5}$, где $\varphi=(1+\sqrt{5})/2$ – золотое сечение.

Напишите функцию `fibBinet(n)`, которая будет вычислять F_n , используя эту формулу. Проверьте её для значения F_{77} (должно получиться `fibBinet(77) = 552793970884757`).

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

[К решению](#)

Случайное из интервала (0, max)

важность: 2

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

[К решению](#)

Случайное из интервала (min, max)

важность: 2

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

[К решению](#)

Случайное целое от min до max

важность: 2

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.

[К решению](#)

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 \(Unicode\)](#).

Создание строк

Строки создаются при помощи двойных или одинарных кавычек:

```
var text = "моя строка";
var anotherText = 'еще строка';
var str = "012345";
```

В JavaScript нет разницы между двойными и одинарными кавычками.

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
<code>\b</code>	Backspace
<code>\f</code>	Form feed
<code>\n</code>	New line
<code>\r</code>	Carriage return
<code>\t</code>	Tab
<code>\uNNNN</code>	Символ в кодировке Юникод с шестнадцатеричным кодом <code>\NNNN</code> . Например, <code>\u00A9</code> -- 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, с некоторыми из которых мы знакомимся ранее, в главе [Введение в методы и свойства](#).

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
```

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 также 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

Для поиска подстроки есть метод [indexOf\(подстрока\[, начальная_позиция\]\)](#).

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

Необязательный второй аргумент позволяет искать, начиная с указанной позиции. Например, первый раз `"id"` появляется на позиции `1`. Чтобы найти его следующее появление – запустим поиск с позиции `2`:

```
var str = "Widget with id";
alert(str.indexOf("id", 2)) // 12, поиск начал с позиции 2
```

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

i 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("get")) {
  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`.

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.

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
```

2. Буква 'ё' находится в алфавите между е и ж: абвгдеёжз... . 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](#).

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](#).

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:

ЊЋЌЙЎЦАБВГДЕЖЗИЙКЛМНОПРСТУФХЦШЩЪЫЬЭЮЯабвгдежзийклмнопрстуфхцшщъыьэюяёёђѓєѕіїјљ

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 'а' > 'Z'.

2. 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: а .

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

В юникоде есть много забавных и полезных символов, например, символ ножниц: ✂ (✂), дроби: 1/2 (½) 3/4 (¾) 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, вернуть 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"
```

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](#)) поддерживают стандарт [ECMA 402](#), 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](#), 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` 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

важность: 5

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

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

P.S. В JavaScript нет встроенного метода для этого. Создайте функцию, используя `toUpperCase()` и `charAt()`.

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Проверьте спам

важность: 5

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
```

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

Усечение строки

важность: 5

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!"
```

Эта функция имеет применение в жизни. Она используется, чтобы усекать слишком длинные темы сообщений.

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

Выделить число

важность: 4

Есть стоимость в виде строки: "\$120". То есть, первым идёт знак валюты, а затем – число.

Создайте функцию `extractCurrencyValue(str)`, которая будет из такой строки выделять число-значение, в данном случае 120.

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

Объекты как ассоциативные массивы

Объекты в JavaScript сочетают в себе два важных функционала.

Первый – это ассоциативный массив: структура, пригодная для хранения любых данных. В этой главе мы рассмотрим использование объектов именно как массивов.

Второй – языковые возможности для объектно-ориентированного программирования. Эти возможности мы изучим в последующих разделах учебника.

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:

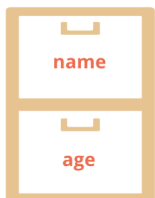
```
var person = {}; // пока пустой
```



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
```



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.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.
```

i 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](#). Она была некоторое время назад, но с тех пор мало что изменилось.

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

Первый объект

важность: 3

Мини-задача на синтаксис объектов. Напишите код, по строке на каждое действие.

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.

[К решению](#)

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.

i 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.

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 пройдем 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, добавляем в начало + , как описано в примере выше, а потом, в процессе обработки, преобразуют такие ключи в числа.

✔ Задачи

Определите, пуст ли объект

важность: 5

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
```

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

Сумма свойств

важность: 5

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.

[К решению](#)

Свойство с наибольшим значением

важность: 5

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"
```

[К решению](#)

Умножьте численные свойства на 2

важность: 3

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"
};
```

P.S. Для проверки на число используйте функцию:

```
function isNumeric(n) {
  return !isNaN(parseFloat(n)) && isFinite(n)
}
```

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

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
```

i 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 клонировать 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++; // (**)

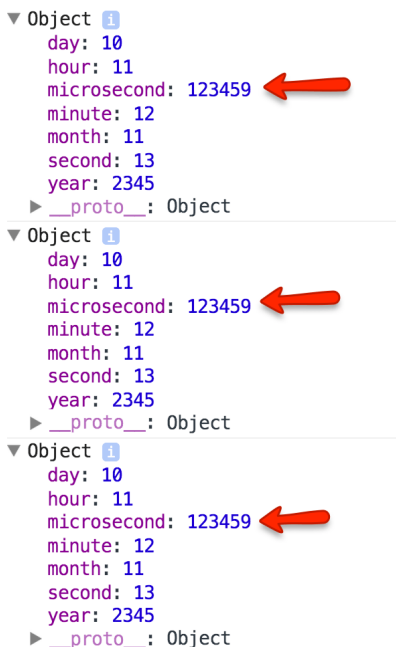
console.log(time);
time.microsecond++;

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):



```
▼ Object ⓘ
  day: 10
  hour: 11
  microsecond: 123459
  minute: 12
  month: 11
  second: 13
  year: 2345
  ► __proto__: Object

▼ Object ⓘ
  day: 10
  hour: 11
  microsecond: 123459
  minute: 12
  month: 11
  second: 13
  year: 2345
  ► __proto__: Object

▼ Object ⓘ
  day: 10
  hour: 11
  microsecond: 123459
  minute: 12
  month: 11
  second: 13
  year: 2345
  ► __proto__: Object
```

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.

Массивы с числовыми индексами

Массив – разновидность объекта, которая предназначена для хранения пронумерованных значений и предлагает дополнительные методы для удобного манипулирования такой коллекцией.

Они обычно используются для хранения упорядоченных коллекций данных, например – списка товаров на странице, студентов в группе и т.п.

Объявление

Синтаксис для создания нового массива – квадратные скобки со списком элементов внутри.

Пустой массив:

```
var arr = [];
```

Массив `fruits` с тремя элементами:

```
var fruits = ["Яблоко", "Апельсин", "Слива"];
```

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
```

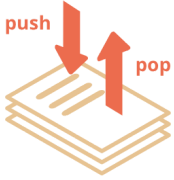
Методы `pop`/`push`, `shift`/`unshift`

Одно из применений массива – это *очередь* [↗](#). 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](#). 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
```

... 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:

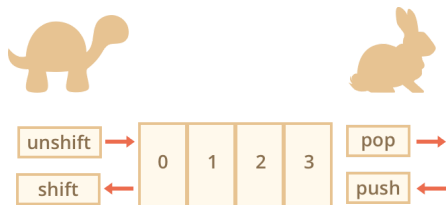
```
var a = [];
a[0] = 0;
a[5] = 5;

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.



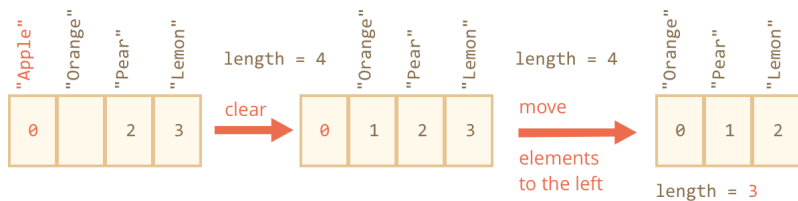
Чтобы понять, почему работать с концом массива – быстрее, чем с его началом, разберём подробнее происходящее при операции:

```
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.



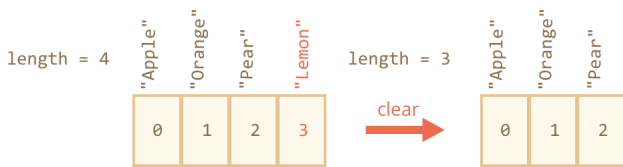
Чем больше элементов в массиве, тем дольше их перемещать, это много операций с памятью.

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 укорачивает `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.

2. 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 перевернуть.

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`.

Создание вызовом `new Array`

`new Array()`

Существует еще один синтаксис для создания массива:

```
var arr = new Array("Яблоко", "Груша", "и т.п.");
```

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(2); // создаст массив [2] ?
alert( arr[0] ); // undefined! у нас массив без элементов, длины 2
```

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.

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

важность: 5

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`.

[К решению](#)

Добавить новый элемент в массив

важность: 5

Как добавить элемент в конец произвольного массива?

У нас есть массив `goods`. Напишите код для добавления в его конец значения «Компьютер».

[К решению](#)

Создание массива

важность: 5

Задача из 5 шагов-строк:

1. Создайте массив `styles` с элементами «Джаз», «Блюз».
2. Добавьте в конец значение «Рок-н-Ролл»
3. Замените предпоследнее значение с конца на «Классика». Код замены предпоследнего значения должен работать для массивов любой длины.
4. Удалите первое значение массива и выведите его `alert`.
5. 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
```

[К решению](#)

Получить случайное значение из массива

важность: 3

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));
```

[К решению](#)

Создайте калькулятор для введённых значений

важность: 4

Напишите код, который:

- Запрашивает по очереди значения при помощи `prompt` и сохраняет их в массиве.
- Заканчивает ввод, как только посетитель введёт пустую строку, не число или нажмёт «Отмена».
- При этом ноль `0` не должен заканчивать ввод, это разрешённое число.
- Выводит сумму всех значений массива

[Запустить демо](#)

[К решению](#)

Чему равен элемент массива?

важность: 3

Что выведет этот код?

```
var arr = [1, 2, 3];  
var arr2 = arr;  
arr2[0] = 5;  
  
alert( arr[0] );  
alert( arr2[0] );
```

[К решению](#)

Поиск в массиве

важность: 3

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
```

[Открыть песочницу с тестами для задачи.](#) [↗](#)

[К решению](#)

Фильтр диапазона

важность: 3

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
```

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Решето Эратосфена

важность: 3

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 $p^2 < n$.
6. Все оставшиеся не зачеркнутыми числа – простые.

Посмотрите также [анимацию алгоритма](#).

Реализуйте «Решето Эратосфена» в JavaScript, используя массив.

Найдите все простые числа до 100 и выведите их сумму.

[К решению](#)

Подмассив наибольшей суммы

важность: 2

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 $O(n^2)$, а лучше за $O(n)$ операций.

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Массивы: методы

В этой главе мы рассмотрим встроенные методы массивов JavaScript.

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] );
}
```

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
```

i 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)` 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
```

i `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();
```

i 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 работает
```

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 $a > b$,
- Отрицательное значение, если $a < b$,
- Если равны – можно 0 , но вообще – не важно, что возвращать, их взаимный порядок не имеет значения.

i Алгоритм сортировки

В методе `sort` , внутри самого интерпретатора JavaScript, реализован универсальный алгоритм сортировки. Как правило, это «**быстрая сортировка**» [↗](#), 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.

reverse

Метод `arr.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)` [↗](#) 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
```

indexOf/lastIndexOf

Эти методы не поддерживаются в IE8-. Для их поддержки подключите библиотеку [ES5-shim](#) [↗](#).

Метод «`arr.indexOf(searchElement[, fromIndex])`» [↗](#) 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](#) [↗](#).

Метод «`arr.lastIndexOf(searchElement[, fromIndex])`» [↗](#) 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\(obj\)](#). Он поддерживается везде, кроме IE8-:

```
var user = {
  name: "Петя",
  age: 30
}

var keys = Object.keys(user);

alert( keys ); // name, age
```

Итого

Методы:

- `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-).

Изученных нами методов достаточно в 95% случаях, но существуют и другие. Для знакомства с ними рекомендуется заглянуть в справочник [Array](#) и [Array в Mozilla Developer Network](#).

Задачи

Добавить класс в строку

важность: 5

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.

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Перевести текст вида border-left-width в borderLeftWidth

важность: 3

Напишите функцию `camelize(str)`, которая преобразует строки вида «my-short-string» в «myShortString».

То есть, дефисы удаляются, а все слова после них получают заглавную букву.

Например:

```
camelize("background-color") == 'backgroundColor';
camelize("list-style-image") == 'listStyleImage';
camelize("-webkit-transition") == 'WebkitTransition';
```

Такая функция полезна при работе с CSS.

P.S. Вам пригодятся методы строк `charAt`, `split` и `toUpperCase`.

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

Функция removeClass

важность: 5

У объекта есть свойство `className`, которое хранит список «классов» – слов, разделенных пробелами:

```
var obj = {
  className: 'open menu'
};
```

Напишите функцию `removeClass(obj, cls)`, которая удаляет класс `cls`, если он есть:

```
removeClass(obj, 'open'); // obj.className='menu'
removeClass(obj, 'blabla'); // без изменений (нет такого класса)
```

P.S. Дополнительное усложнение. Функция должна корректно обрабатывать дублирование класса в строке:

```
obj = {
  className: 'my menu menu'
};
removeClass(obj, 'menu');
alert( obj.className ); // 'my'
```

Лишних пробелов после функции образовываться не должно.

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

Фильтрация массива "на месте"

важность: 4

Создайте функцию `filterRangeInPlace(arr, a, b)`, которая получает массив с числами `arr` и удаляет из него все числа вне диапазона `a..b`. То есть, проверка имеет вид `a ≤ arr[i] ≤ b`. Функция должна менять сам массив и ничего не возвращать.

Например:

```
arr = [5, 3, 8, 1];
filterRangeInPlace(arr, 1, 4); // удалены числа вне диапазона 1..4
alert( arr ); // массив изменился: остались [3, 1]
```

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

Сортировать в обратном порядке

важность: 5

Как отсортировать массив чисел в обратном порядке?

```
var arr = [5, 2, 1, -10, 8];
// отсортируйте?
alert( arr ); // 8, 5, 2, 1, -10
```

[К решению](#)

Скопировать и отсортировать массив

важность: 5

Есть массив строк `arr`. Создайте массив `arrSorted` – из тех же элементов, но отсортированный.

Исходный массив не должен меняться.

```
var arr = ["HTML", "JavaScript", "CSS"];
// ... ваш код ...

alert( arrSorted ); // CSS, HTML, JavaScript
alert( arr ); // HTML, JavaScript, CSS (без изменений)
```

Постарайтесь сделать код как можно короче.

[К решению](#)

Случайный порядок в массиве

важность: 3

Используйте функцию `sort` для того, чтобы «перетрясти» элементы массива в случайном порядке.

```
var arr = [1, 2, 3, 4, 5];
arr.sort(ваша функция);
alert( arr ); // элементы в случайном порядке, например [3,5,1,2,4]
```

[К решению](#)

Сортировка объектов

важность: 5

Напишите код, который отсортирует массив объектов `people` по полю `age`.

Например:

```
var vasya = { name: "Вася", age: 23 };
var masha = { name: "Маша", age: 18 };
var vovochka = { name: "Вовочка", age: 6 };

var people = [ vasya , masha , vovochka ];

... ваш код ...

// теперь people: [vovochka, masha, vasya]
alert(people[0].age) // 6
```

Выведите список имён в массиве после сортировки.

[К решению](#)

Вывести односвязный список

важность: 5

Односвязный список [↗](#) – это структура данных, которая состоит из *элементов*, каждый из которых хранит ссылку на следующий. Последний элемент может не иметь ссылки, либо она равна `null`.

Например, объект ниже задаёт односвязный список, в `next` хранится ссылка на следующий элемент:

```
var list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
}
```

```
}  
}  
};
```



Графическое представление этого списка:

Альтернативный способ создания:

```
var list = { value: 1 };  
list.next = { value: 2 };  
list.next.next = { value: 3 };  
list.next.next.next = { value: 4 };
```

Такая структура данных интересна тем, что можно очень быстро разбить список на части, объединить списки, удалить или добавить элемент в любое место, включая начало. При использовании массива такие действия требуют обширных перенумерований.

Задачи:

1. Напишите функцию `printList(list)`, которая выводит элементы списка по очереди, при помощи цикла.
2. Напишите функцию `printList(list)` при помощи рекурсии.
3. Напишите функцию `printReverseList(list)`, которая выводит элементы списка в обратном порядке, при помощи рекурсии. Для списка выше она должна выводить `4, 3, 2, 1`
4. Сделайте вариант `printReverseList(list)`, использующий не рекурсию, а цикл.

Как лучше – с рекурсией или без?

[К решению](#)

Отфильтровать анаграммы

важность: 3

Анаграммы – слова, состоящие из одинакового количества одинаковых букв, но в разном порядке. Например:

```
воз - зов  
киборг - гробик  
корсет - костер - сектор
```

Напишите функцию `aclean(arr)`, которая возвращает массив слов, очищенный от анаграмм.

Например:

```
var arr = ["воз", "киборг", "корсет", "ЗОВ", "гробик", "костер", "сектор"];  
alert( aclean(arr) ); // "воз,киборг,корсет" или "ЗОВ,гробик,сектор"
```

Из каждой группы анаграмм должно остаться только одно слово, не важно какое.

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

Оставить уникальные элементы массива

важность: 3

Пусть `arr` – массив строк.

Напишите функцию `unique(arr)`, которая возвращает массив, содержащий только уникальные элементы `arr`.

Например:

```
function unique(arr) {  
  /* ваш код */  
}  
  
var strings = ["кришна", "кришна", "хапе", "хапе",  
  "хапе", "хапе", "кришна", "кришна", "8-()"];  
  
alert( unique(strings) ); // кришна, хапе, 8-()
```

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

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](#).

forEach

Метод `arr.forEach(callback[, thisArg])` 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 + ")");
});
```

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 примере.

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

sum	sum	sum	sum	sum
0	0+1	0+1+2	0+1+2+3	0+1+2+3+4
current	current	current	current	current
1	2	3	4	5

1	2	3	4	5
---	---	---	---	---

 → 0+1+2+3+4+5 = 15

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 `arr.reduceRight` работает аналогично, но идёт по массиву справа-налево.

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`

важность: 5

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`.

[К решению](#)

Массив частичных сумм

важность: 2

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`.

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Псевдомассив аргументов "arguments"

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
```

i 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](#).

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);
```

Наоборот:

```
function f(x) {
  x = 5;
  alert( arguments[0] ); // 5, обновленный x
}

f(1);
```

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 \(NFE\)](#).

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

важность: 5

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
```

[К решению](#)

Сумма аргументов

важность: 5

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
```

[К решению](#)

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)`

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
```

Получение компонентов даты

Для доступа к компонентам даты-времени объекта `Date` используются следующие методы:

`getFullYear()`

Получить год(из 4 цифр)

`getMonth()`

Получить месяц, от 0 до 11.

`getDate()`

Получить число месяца, от 1 до 31.

`getHours()`, `getMinutes()`, `getSeconds()`, `getMilliseconds()`

Получить соответствующие компоненты.

⚠ Не `getFullYear()`, а `getFullYear()`

Некоторые браузеры реализуют нестандартный метод `getFullYear()`. Где-то он возвращает только две цифры из года, где-то четыре. Так или иначе, этот метод отсутствует в стандарте JavaScript. Не используйте его. Для получения года есть `getFullYear()`.

Дополнительно можно получить день недели:

`getDay()`

Получить номер дня в неделе. Неделя в JavaScript начинается с воскресенья, так что результат будет числом от 0(воскресенье) до 6(суббота).

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:

- `setFullYear(year [, month, date])`
- `setMonth(month [, date])`
- `setDate(date)`
- `setHours(hour [, min, sec, ms])`
- `setMinutes(min [, sec, ms])`
- `setSeconds(sec [, ms])`
- `setMilliseconds(ms)`
- `setTime(milliseconds)` (устанавливает всю дату по миллисекундам с 01.01.1970 UTC)

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.
```

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; // засекали 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" );
```

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' );
```

i Более точное время с `performance.now()`

В современных браузерах (кроме IE9-) вызов `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](#), 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](#), 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](#) 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)
```

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](#).

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](#).

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](#). Создать «только время» при этом нельзя, оно должно быть с датой. Список методов `Date` вы можете найти в справочнике [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

важность: 5

Создайте объект `Date` для даты: 20 февраля 2012 года, 3 часа 12 минут.

Временная зона – местная. Выведите его на экран.

[К решению](#)

Имя дня недели

важность: 5

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'
```

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

День недели в европейской нумерации

важность: 5

Напишите функцию, `getLocalDay(date)` которая возвращает день недели для даты `date`.

День нужно вернуть 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
```

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

День указанное количество дней назад

важность: 4

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`.

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Последний день месяца?

важность: 5

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).

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

Сколько секунд уже прошло сегодня?

важность: 5

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.

[К решению](#)

Сколько секунд - до завтра?

важность: 5

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.

[К решению](#)

Вывести дату в формате дд.мм.гг

важность: 3

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`.

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

Относительное форматирование даты

важность: 4

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"
```

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

Замыкания, область видимости

Понимание «области видимости» и «замыканий» – ключевое в изучении JavaScript, без них «каши не сварить».

В этом разделе мы более глубоко изучаем переменные и функции – и замыкания в том числе.

Глобальный объект

Механизм работы функций и переменных в JavaScript очень отличается от большинства языков.

Чтобы его понять, мы в этой главе рассмотрим переменные и функции в глобальной области. А в следующей – пойдём дальше.

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](#), 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`.

2. На второй фазе – собственно, выполнение.

Присваивание (`=`) значений переменных происходит, когда поток выполнения доходит до соответствующей строки 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() { /*...*/ };
```

i 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`.

i 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` .

2. 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 выполняются позже, когда выполнение дойдет до них.

Задачи

Window и переменная

важность: 5

Каков будет результат кода?

```
if ("a" in window) {
  var a = 1;
}
alert( a );
```

[К решению](#)

Window и переменная 2

важность: 5

Каков будет результат (перед `a` нет `var`)?

```
if ("a" in window) {
  a = 1;
}
alert( a );
```

[К решению](#)

Window и переменная 3

важность: 5

Каков будет результат (перед `a` нет `var`, а ниже есть)?

```
if ("a" in window) {
  a = 1;
}
var a;

alert( a );
```

[К решению](#)

Замыкания, функции изнутри

В этой главе мы продолжим рассматривать, как работают переменные, и, как следствие, познакомимся с замыканиями. От глобального объекта мы переходим к работе внутри функций.

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');
```

2. 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');
```

3. 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.

i 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.

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 «продвинутой» 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(); // (*)
```

2. В процессе выполнения `makeCounter()` создаёт функцию в строке `(**)`. При создании эта функция получает внутреннее свойство `[[Scope]]` со ссылкой на текущий `LexicalEnvironment`.

3. 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

Что выведет say в начале кода?

важность: 5

Что будет, если вызов `say('Вася')`; стоит в самом-самом начале, в первой строке кода?

```
say('Вася'); // Что выведет? Не будет ли ошибки?
```

```
var phrase = 'Привет';

function say(name) {
  alert( name + ", " + phrase );
}
```

[К решению](#)

В какую переменную будет присвоено значение?

важность: 5

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 ?

[К решению](#)

var window

важность: 5

Каков будет результат выполнения этого кода? Почему?

```
function test() {
  alert( window );
  var window = 5;
  alert( window );
}

test();
```

[К решению](#)

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.

[К решению](#)

Перекрытие переменной

важность: 4

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)?
  };
}
```

[К решению](#)

Глобальный счётчик

важность: 5

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() ); // ?
```

[К решению](#)

[[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?

Продвинутое знание

Содержимое этой секции содержит продвинутое теоретическое знание, которое прямо сейчас не обязательно 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.

Напомним, 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](#) предоставляет глобальную переменную с именем [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

важность: 4

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
```

[К решению](#)

Функция - строковый буфер

важность: 5

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.

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Строковый буфер с очисткой

важность: 5

Добавьте буферу из решения задачи [Функция - строковый буфер](#) 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() ); // ""
```

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Сортировка

важность: 5

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.

[К решению](#)

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
```

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Армия функций

важность: 5

Следующий код создает массив функций-стрелок `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 стрелки выводят одно и то же? Поправьте код, чтобы стрелки работали как задумано. Предложите несколько вариантов исправления.

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

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>
```

[Открыть в песочнице](#) ↗

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();
})();
```

[Открыть в песочнице](#) ↗

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](#), хотя могли бы и [jQuery](#), 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](#).

Пример использования:

```
<p>Подключим библиотеку</p>
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"></script>

<p>Функция _.defaults() 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` выносятся 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 `5` как 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*.

2. 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](#) (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 на объекты `vasya`, `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](#) отслеживает такие ситуации и очищает память.

Алгоритм сборки мусора

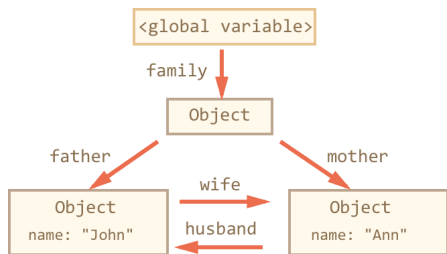
Сборщик мусора идёт от корня по ссылкам и запоминает все найденные объекты. По окончании – он смотрит, какие объекты в нём отсутствуют и удаляет их.

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:



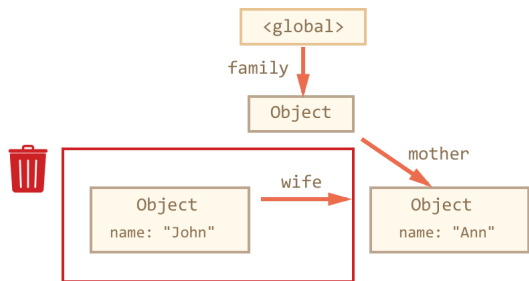
Здесь стрелочками показаны ссылки, а вот свойство `name` ссылкой не является, там хранится примитив, поэтому оно внутри самого объекта.

Чтобы запустить сборщик мусора, удалим две ссылки:

```
delete family.father;
delete family.mother.husband;
```

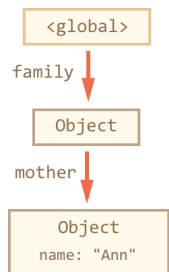
Обратим внимание, удаление только одной из этих ссылок ни к чему бы не привело. Пока до объекта можно добраться из корня `window`, объект остаётся жив.

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.

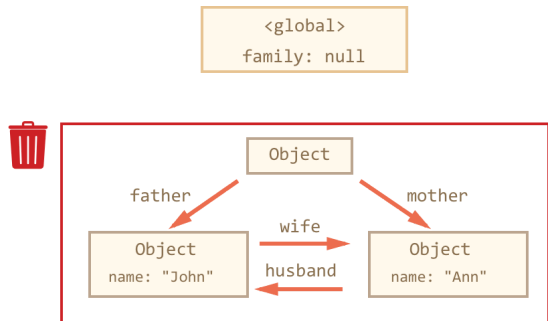


А теперь – рассмотрим более сложный случай. Что будет, если удалить главную ссылку `family` ?

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.

i 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 меняться 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](#).

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 окончании, 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() {};
}

// 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 показатель. 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 );
alert( window.size );
```

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.

2. Еще одна причина – алгоритмы сжатия JavaScript не любят with. Перед выкладкой на сервер JavaScript сжимают. Для этого есть много инструментов, например [Closure Compiler](#) и [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.
3. Ну и, наконец, производительность – усложнение поиска переменной из-за with влечет дополнительные накладные расходы.

Современные движки применяют много внутренних оптимизаций, ряд которых не могут быть применены к коду, в котором есть with.

Вот, к примеру, запустите этот код в современном браузере. Производительность функции fast существенно 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

важность: 5

Какая из функций будет вызвана?

```
function f() {  
  alert(1)  
}  
  
var obj = {  
  f: function() {  
    alert(2)  
  }  
};  
  
with(obj) {  
  f();  
}
```

[К решению](#)

With + переменные

важность: 5

Что выведет этот код?

```
var a = 1;  
  
var obj = {  
  b: 2  
};  
  
with(obj) {  
  var b;  
  alert( a + b );  
}
```

[К решению](#)

Методы объектов и контекст вызова

Начинаем изучать объектно-ориентированную разработку – как работают объекты и функции, что такое контекст вызова и способы его передачи.

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](#).

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

важность: 5

Каким будет результат? Почему?

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

[К решению](#)

Проверка синтаксиса

важность: 2

Каков будет результат этого кода?

```
var obj = {
  go: function() { alert(this) }
}

(obj.go)()
```

P.S. Есть подвох :)

[К решению](#)

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 .

[К решению](#)

Значение this в объявлении объекта

важность: 5

Что выведет alert в этом коде? Почему?

```
var user = {
  firstName: "Василий",
  export: this
};

alert( user.export.firstName );
```

[К решению](#)

Возврат this

важность: 5

Что выведет alert в этом коде? Почему?

```
var name = "";

var user = {
  name: "Василий",
  export: function() {
    return this;
  }
};
```

```
alert( user.export().name );
```

[К решению](#)

Возврат объекта с this

важность: 5

Что выведет `alert` в этом коде? Почему?

```
var name = "";  
  
var user = {  
  name: "Василий",  
  
  export: function() {  
    return {  
      value: this  
    };  
  }  
};  
  
alert( user.export().value.name );
```

[К решению](#)

Создайте калькулятор

важность: 5

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](#)

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

Цепочка вызовов

важность: 2

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.

[К решению](#)

Преобразование объектов: toString и valueOf

Ранее, в главе [Преобразование типов для примитивов](#) 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
```

i 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.

i 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
```

i Детали спецификации

Если посмотреть в стандарт, то в пункте [15.2.4.4](#) 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 вернуть 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](#), [11.9.3](#), а также [9.1](#) и [9.3](#).

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: + {}
```

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'`

важность: 5

Почему результат true ?

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

[К решению](#)

Преобразование

важность: 5

Объявлен объект с `toString` и `valueOf`.

Какими будут результаты `alert` ?

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

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

Подумайте, прежде чем ответить.

[К решению](#)

Почему `[] == []` неверно, а `[] == ![]` верно?

важность: 5

Почему первое равенство – неверно, а второе – верно?

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

Какие преобразования происходят при вычислении?

[К решению](#)

Вопросник по преобразованиям, для объектов

важность: 5

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

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

[К решению](#)

Сумма произвольного количества скобок

важность: 2

Напишите функцию `sum`, которая будет работать так:

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

Количество скобок может быть любым.

Пример такой функции для двух аргументов – есть в решении задачи [Сумма через замыкание](#).

[К решению](#)

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 Animal(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`.

Подробнее – 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`, 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.

i 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.

i 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](#).

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 вернуть свой объект, то её результатом будет `this`.
- Функции, которые предназначены для создания объектов, называются *конструкторами*. Их названия пишут с большой буквы, чтобы отличать от обычных.

✔ Задачи

Две функции один объект

важность: 2

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.

[К решению](#)

Создать Calculator при помощи конструктора

важность: 5

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() );
```

[Запустить демо](#)

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Создать Accumulator при помощи конструктора

важность: 5

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
```

[Запустить демо](#)

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Создайте калькулятор

важность: 5

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

Эта задача состоит из двух частей, которые можно решать одна за другой.

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

Пример использования:

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

2. Второй шаг – добавить 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.

[Открыть песочницу с тестами для задачи.](#)

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();
      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.defineProperty\(obj, descriptors\)](#)

Позволяет объявить несколько свойств сразу:

```
var user = {}

Object.defineProperty(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 поставим 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

важность: 5

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.

[К решению](#)

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](#), [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 "Вынык 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`. 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)`. 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

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

важность: 5

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)
```

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

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 «одаживание 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; // одалили 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.

i Почему вызов сработает?

Здесь метод `join` массива скопирован и вызван в контексте `arguments`. Не произойдёт ли что-то плохое от того, что `arguments` – не массив? Почему он, вообще, сработал?

Ответ на эти вопросы простой. В соответствии [со спецификацией](#), внутри `join` реализован примерно так:

```
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: "A",
  1: "B",
  2: "B",
  length: 3
};

obj.join = [].join;
alert( obj.join(';') ); // "A;B;B"
```

...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](#).

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, ...)`, 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!

i 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 ); // null
}

f.call(null);
```

Без `use strict`:

```
function f() {
  alert( this ); // window
}

f.call(null);
```

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

важность: 5

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.

[К решению](#)

Примените функцию к аргументам

важность: 5

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 вернуть 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
```

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

Привязка контекста и карринг: "bind"

Функции в JavaScript никак не привязаны к своему контексту `this`, с одной стороны, здорово – это позволяет быть максимально гибкими, одалживать методы и так далее.

Но с другой стороны – в некоторых случаях контекст может быть потерян. То есть мы вроде как вызываем метод объекта, а на самом деле он получает `this = undefined`.

Такая ситуация является типичной для начинающих разработчиков, но бывает и у «зубров» тоже. Конечно, «зубры» при этом знают, что с ней делать.

Пример потери контекста

В браузере есть встроенная функция `setTimeout(func, ms)`, которая вызывает выполнение функции `func` через `ms` миллисекунд (=1/1000 секунды).

Мы подробно остановимся на ней и её тонкостях позже, в главе [setTimeout и setInterval](#), а пока просто посмотрим пример.

Этот код выведет «Привет» через 1000 мс, то есть 1 секунду:

```
setTimeout(function() {
  alert( "Привет" );
}, 1000);
```

Попробуем сделать то же самое с методом объекта, следующий код должен выводить имя пользователя через 1 секунду:

```
var user = {
  firstName: "Вася",
  sayHi: function() {
    alert( this.firstName );
  }
};
```

```
setTimeout(user.sayHi, 1000); // undefined (не Вася!)
```

При запуске кода выше через секунду выводится вовсе не "Вася", а `undefined`!

Это произошло потому, что в примере выше `setTimeout` получил функцию `user.sayHi`, но не её контекст. То есть, последняя строчка аналогична двум таким:

```
var f = user.sayHi;
setTimeout(f, 1000); // контекст user потеряли
```

Ситуация довольно типична – мы хотим передать метод объекта куда-то в другое место кода, откуда он потом может быть вызван. Как бы прикрепить к нему контекст, желательно, с минимумом плясок с бубном и при этом надёжно?

Есть несколько способов решения, среди которых мы, в зависимости от ситуации, можем выбирать.

Решение 1: сделать обёртку

Самый простой вариант решения – это обернуть вызов в анонимную функцию:

```
var user = {
  firstName: "Вася",
  sayHi: function() {
    alert( this.firstName );
  }
};
```

```
setTimeout(function() {
  user.sayHi(); // Вася
}, 1000);
```

Теперь код работает, так как `user` достаётся из замыкания.

Это решение также позволяет передать дополнительные аргументы:

```
var user = {
  firstName: "Вася",
  sayHi: function(who) {
    alert( this.firstName + ": Привет, " + who );
  }
};
```

```
setTimeout(function() {
  user.sayHi("Петя"); // Вася: Привет, Петя
}, 1000);
```

Но тут же появляется и уязвимое место в структуре кода!

А что, если до срабатывания `setTimeout` (ведь есть целая секунда) в переменную `user` будет записано другое значение? К примеру, в другом месте кода будет присвоено `user=(другой пользователь) ...` В этом случае вызов неожиданно будет совсем не тот!

Хорошо бы гарантировать правильность контекста.

Решение 2: bind для привязки контекста

Напишем вспомогательную функцию `bind(func, context)`, которая будет жёстко фиксировать контекст для `func`:

```
function bind(func, context) {
  return function() { // (*)
    return func.apply(context, arguments);
  };
}
```

Посмотрим, что она делает, как работает, на таком примере:

```
function f() {
  alert( this );
}

var g = bind(f, "Context");
g(); // Context
```

То есть, `bind(f, "Context")` привязывает "Context" в качестве `this` для `f`.

Посмотрим, за счёт чего это происходит.

Результатом `bind(f, "Context")`, как видно из кода, будет анонимная функция (*).

Вот она отдельно:

```
function() { // (*)
  return func.apply(context, arguments);
};
```

Если подставить наши конкретные аргументы, то есть `f` и "Context", то получится так:

```
function() { // (*)
  return f.apply("Context", arguments);
};
```

Эта функция запишется в переменную `g`.

Далее, если вызвать `g`, то вызов будет передан в `f`, причём `f.apply("Context", arguments)` передаст в качестве контекста "Context", который и будет выведен.

Если вызвать `g` с аргументами, то также будет работать:

```
function f(a, b) {
  alert( this );
  alert( a + b );
}

var g = bind(f, "Context");
g(1, 2); // Context, затем 3
```

Аргументы, которые получила `g(...)`, передаются в `f` также благодаря методу `.apply`.

Иными словами, в результате вызова `bind(func, context)` мы получаем «функцию-обёртку», которая прозрачно передаёт вызов в `func`, с теми же аргументами, но фиксированным контекстом `context`.

Вернёмся к `user.sayHi`. Вариант с `bind`:

```
function bind(func, context) {
  return function() {
    return func.apply(context, arguments);
  };
}

var user = {
  firstName: "Бая",
  sayHi: function() {
    alert( this.firstName );
  }
};

setTimeout(bind(user.sayHi, user), 1000);
```

Теперь всё в порядке!

Вызов `bind(user.sayHi, user)` возвращает такую функцию-обёртку, которая привязывает `user.sayHi` к контексту `user`. Она будет вызвана через 1000 мс.

Полученную обёртку можно вызвать и с аргументами – они пойдут в `user.sayHi` без изменений, фиксирован лишь контекст.

```
var user = {
  firstName: "Бая",
  sayHi: function(who) { // здесь у sayHi есть один аргумент
    alert( this.firstName + ": Привет, " + who );
  }
};

var sayHi = bind(user.sayHi, user);
```

```
// контекст Вася, а аргумент передаётся "как есть"
sayHi("Петя"); // Вася: Привет, Петя
sayHi("Маша"); // Вася: Привет, Маша
```

В примере выше продемонстрирована другая частая цель использования `bind` – «привязать» функцию к контексту, чтобы в дальнейшем «не таскать за собой» объект, а просто вызывать `sayHi`.

Результат `bind` можно передавать в любое место кода, вызывать как обычную функцию, он «помнит» свой контекст.

Решение 3: встроенный метод `bind`

В современном JavaScript (или при подключении библиотеки [es5-shim](#) для IE8-) у функций уже есть встроенный метод `bind`, который мы можем использовать.

Он работает примерно так же, как `bind`, который описан выше.

Изменения очень небольшие:

```
function f(a, b) {
  alert( this );
  alert( a + b );
}

// вместо
// var g = bind(f, "Context");
var g = f.bind("Context");
g(1, 2); // Context, затем 3
```

Синтаксис встроенного `bind`:

```
var wrapper = func.bind(context[, arg1, arg2...])
```

func

Произвольная функция

context

Контекст, который привязывается к `func`

arg1, arg2, ...

Если указаны аргументы `arg1, arg2...` – они будут прибавлены к каждому вызову новой функции, причем встанут *перед* теми, которые указаны при вызове.

Результат вызова `func.bind(context)` аналогичен вызову `bind(func, context)`, описанному выше. То есть, `wrapper` – это обёртка, фиксирующая контекст и передающая вызовы в `func`. Также можно указать аргументы, тогда и они будут фиксированы, но об этом чуть позже.

Пример со встроенным методом `bind`:

```
var user = {
  firstName: "Вася",
  sayHi: function() {
    alert( this.firstName );
  }
};

// setTimeout( bind(user.sayHi, user), 1000 );
setTimeout(user.sayHi.bind(user), 1000); // аналог через встроенный метод
```

Получили простой и надёжный способ привязать контекст, причём даже встроенный в JavaScript.

Далее мы будем использовать именно встроенный метод `bind`.

bind не похож на call/apply

Методы `bind` и `call/apply` близки по синтаксису, но есть важнейшее отличие.

Методы `call/apply` вызывают функцию с заданным контекстом и аргументами.

А `bind` не вызывает функцию. Он только возвращает «обёртку», которую мы можем вызвать позже, и которая передаст вызов в исходную функцию, с привязанным контекстом.

i Привязать всё: `bindAll`

Если у объекта много методов и мы планируем их активно передавать, то можно привязать контекст для них всех в цикле:

```
for (var prop in user) {
  if (typeof user[prop] == 'function') {
    user[prop] = user[prop].bind(user);
  }
}
```

В некоторых JS-фреймворках есть даже встроенные функции для этого, например `_.bindAll(obj)` [↗](#).

Карринг

До этого мы говорили о привязке контекста. Теперь пойдём на шаг дальше. Привязывать можно не только контекст, но и аргументы. Используется это реже, но бывает полезно.

Карринг [↗](#) (currying) или *карирование* – термин **функционального программирования** [↗](#), который означает создание новой функции путём фиксации аргументов существующей.

Как было сказано выше, метод `func.bind(context, ...)` может создавать обёртку, которая фиксирует не только контекст, но и ряд аргументов функции.

Например, есть функция умножения двух чисел `mul(a, b)`:

```
function mul(a, b) {
  return a * b;
};
```

При помощи `bind` создадим функцию `double`, удваивающую значения. Это будет вариант функции `mul` с фиксированным первым аргументом:

```
// double умножает только на два
var double = mul.bind(null, 2); // контекст фиксируем null, он не используется

alert( double(3) ); // = mul(2, 3) = 6
alert( double(4) ); // = mul(2, 4) = 8
alert( double(5) ); // = mul(2, 5) = 10
```

При вызове `double` будет передавать свои аргументы исходной функции `mul` после тех, которые указаны в `bind`, то есть в данном случае после зафиксированного первого аргумента `2`.

Говорят, что `double` является «частичной функцией» (partial function) от `mul`.

Другая частичная функция `triple` утраивает значения:

```
var triple = mul.bind(null, 3); // контекст фиксируем null, он не используется

alert( triple(3) ); // = mul(3, 3) = 9
alert( triple(4) ); // = mul(3, 4) = 12
alert( triple(5) ); // = mul(3, 5) = 15
```

При помощи `bind` мы можем получить из функции её «частный вариант» как самостоятельную функцию и дальше передать в `setTimeout` или сделать с ней что-то ещё.

Наш выигрыш состоит в том, что эта самостоятельная функция, во-первых, имеет понятное имя (`double`, `triple`), а во-вторых, повторные вызовы позволяют не указывать каждый раз первый аргумент, он уже фиксирован благодаря `bind`.

Функция `ask` для задач

В задачах этого раздела предполагается, что объявлена следующая «функция вопросов» `ask`:

```
function ask(question, answer, ok, fail) {
  var result = prompt(question, '');
  if (result.toLowerCase() == answer.toLowerCase()) ok();
  else fail();
}
```

Её назначение – задать вопрос `question` и, если ответ совпадёт с `answer`, то запустить функцию `ok()`, а иначе – функцию `fail()`.

Несмотря на внешнюю простоту, функции такого вида активно используются в реальных проектах. Конечно, они будут сложнее, вместо `alert/prompt` – вывод красивого JavaScript-диалога с рамочками, кнопками и так далее, но это нам сейчас не нужно.

Пример использования:

```
ask("Выпустить птичку?", "да", fly, die);

function fly() {
  alert( 'улетела :)' );
}
```

```
function die() {
  alert( 'птичку жалко :(' );
}
```

Итого

- Функция сама по себе не запоминает контекст выполнения.
- Чтобы гарантировать правильный контекст для вызова `obj.func()`, нужно использовать функцию-обёртку, задать её через анонимную функцию:

```
setTimeout(function() {
  obj.func();
})
```

- ...Либо использовать `bind`:

```
setTimeout(obj.func.bind(obj));
```

- Вызов `bind` часто используют для привязки функции к контексту, чтобы затем присвоить её в обычную переменную и вызывать уже без явного указания объекта.
- Вызов `bind` также позволяет фиксировать первые аргументы функции («каррировать» её), и таким образом из общей функции получить её «частные» варианты – чтобы использовать их многократно без повтора одних и тех же аргументов каждый раз.

✔ Задачи

Кросс-браузерная эмуляция `bind`

важность: 3

Если вы вдруг захотите копнуть поглубже – аналог `bind` для IE8- и старых версий других браузеров будет выглядеть следующим образом:

```
function bind(func, context /*, args*/) {
  var bindArgs = [].slice.call(arguments, 2); // (1)
  function wrapper() { // (2)
    var args = [].slice.call(arguments);
    var unshiftArgs = bindArgs.concat(args); // (3)
    return func.apply(context, unshiftArgs); // (4)
  }
  return wrapper;
}
```

Использование – вместо `mul.bind(null, 2)` вызывать `bind(mul, null, 2)`.

Не факт, что он вам понадобится, но в качестве упражнения попробуйте разобраться, как это работает.

[К решению](#)

Запись в объект после `bind`

важность: 5

Что выведет функция?

```
function f() {
  alert( this );
}

var user = {
  g: f.bind("Hello")
}

user.g();
```

[К решению](#)

Повторный `bind`

важность: 5

Что выведет этот код?

```
function f() {
  alert(this.name);
}

f = f.bind( {name: "Вася"} ).bind( {name: "Петя"} );

f();
```

[К решению](#)

Свойство функции после bind

важность: 5

В свойство функции записано значение. Изменится ли оно после применения `bind`? Обоснуйте ответ.

```
function sayHi() {  
  alert( this.name );  
}  
sayHi.test = 5;  
alert( sayHi.test ); // 5
```

```
var bound = sayHi.bind({  
  name: "Вася"  
});  
  
alert( bound.test ); // что выведет? почему?
```

[К решению](#)

Использование функции вопросов

важность: 5

Вызов `user.checkPassword()` в коде ниже должен, при помощи `ask`, спрашивать пароль и вызывать `loginOk/loginFail` в зависимости от правильности ответа.

Однако, его вызов приводит к ошибке. Почему?

Исправьте выделенную строку, чтобы всё работало (других строк изменять не надо).

```
"use strict";  
  
function ask(question, answer, ok, fail) {  
  var result = prompt(question, '');  
  if (result.toLowerCase() == answer.toLowerCase()) ok();  
  else fail();  
}  
  
var user = {  
  login: 'Василий',  
  password: '12345',  
  
  loginOk: function() {  
    alert( this.login + ' вошёл в сайт' );  
  },  
  
  loginFail: function() {  
    alert( this.login + ': ошибка входа' );  
  },  
  
  checkPassword: function() {  
    ask("Ваш пароль?", this.password, this.loginOk, this.loginFail);  
  }  
};  
  
user.checkPassword();
```

P.S. Ваше решение должно также срабатывать, если переменная `user` будет перезаписана, например вместо `user.checkPassword()` в конце будут строки:

```
var vasya = user;  
user = null;  
vasya.checkPassword();
```

[К решению](#)

Использование функции вопросов с каррингом

важность: 5

Эта задача – усложнённый вариант задачи [Использование функции вопросов](#). В ней объект `user` изменён.

Теперь заменим две функции `user.loginOk()` и `user.loginFail()` на единый метод: `user.loginDone(true/false)`, который нужно вызвать с `true` при верном ответе и `fail` – при неверном.

Код ниже делает это, соответствующий фрагмент выделен.

Сейчас он обладает важным недостатком: при записи в `user` другого значения объект перестанет корректно работать, вы увидите это, запустив пример ниже (будет ошибка).

Как бы вы написали правильно?

Исправьте выделенный фрагмент, чтобы код заработал.

```
"use strict";

function ask(question, answer, ok, fail) {
  var result = prompt(question, '');
  if (result.toLowerCase() == answer.toLowerCase()) ok();
  else fail();
}

var user = {
  login: 'Василий',
  password: '12345',

  // метод для вызова из ask
  loginDone: function(result) {
    alert( this.login + (result ? ' вошёл в сайт' : ' ошибка входа' ) );
  },

  checkPassword: function() {
    ask("Ваш пароль?", this.password,
      function() {
        user.loginDone(true);
      },
      function() {
        user.loginDone(false);
      }
    );
  }
};

var vasya = user;
user = null;
vasya.checkPassword();
```

Изменения должны касаться только выделенного фрагмента.

Если возможно, предложите два решения, одно – с использованием `bind`, другое – без него. Какое решение лучше?

[К решению](#)

Функции-обёртки, декораторы

JavaScript предоставляет удивительно гибкие возможности по работе с функциями: их можно передавать, в них можно записывать данные как в объекты, у них есть свои встроенные методы...

Конечно, этим нужно уметь пользоваться. В этой главе, чтобы более глубоко понимать работу с функциями, мы рассмотрим создание функций-обёрток или, иначе говоря, «декораторов».

Декоратор [↗](#) – приём программирования, который позволяет взять существующую функцию и изменить/расширить ее поведение.

Декоратор получает функцию и возвращает обертку, которая делает что-то своё «вокруг» вызова основной функции.

`bind` – привязка контекста

Один простой декоратор вы уже видели ранее – это функция `bind`:

```
function bind(func, context) {
  return function() {
    return func.apply(context, arguments);
  };
}
```

Вызов `bind(func, context)` возвращает обёртку, которая ставит `this` и передаёт основную работу функции `func`.

Декоратор-таймер

Создадим более сложный декоратор, измеряющий время выполнения функции.

Он будет называться `timingDecorator` и получать функцию вместе с «названием таймера», а возвращать – функцию-обёртку, которая измеряет время и прибавляет его в специальный объект `timer` по свойству-названию.

Использование:

```
function f(x) {} // любая функция

var timers = {}; // объект для таймеров

// отдекорировали
f = timingDecorator(f, "myFunc");

// запускаем
f(1);
f(2);
f(3); // функция работает как раньше, но время подсчитывается

alert( timers.myFunc ); // общее время выполнения всех вызовов f
```

При помощи декоратора `timingDecorator` мы сможем взять произвольную функцию и одним движением руки прикрутить к ней измеритель времени.

Его реализация:

```
var timers = {};  
  
// прибавит время выполнения f к таймеру timers[timer]  
function timingDecorator(f, timer) {  
  return function() {  
    var start = performance.now();  
  
    var result = f.apply(this, arguments); // (*)  
  
    if (!timers[timer]) timers[timer] = 0;  
    timers[timer] += performance.now() - start;  
  
    return result;  
  }  
}  
  
// функция может быть произвольной, например такой:  
var fibonacci = function f(n) {  
  return (n > 2) ? f(n - 1) + f(n - 2) : 1;  
}  
  
// использование: завернём fibonacci в декоратор  
fibonacci = timingDecorator(fibonacci, "fibo");  
  
// неоднократные вызовы...  
alert( fibonacci(10) ); // 55  
alert( fibonacci(20) ); // 6765  
// ...  
  
// в любой момент можно получить общее количество времени на вызовы  
alert( timers.fibo + 'мс' );
```

Обратим внимание на строку `(*)` внутри декоратора, которая и осуществляет передачу вызова:

```
var result = f.apply(this, arguments); // (*)
```

Этот приём называется «форвардинг вызова» (от англ. forwarding): текущий контекст и аргументы через `apply` передаются в функцию `f`, так что изнутри `f` всё выглядит так, как была вызвана она напрямую, а не декоратор.

Декоратор для проверки типа

В JavaScript, как правило, пренебрегают проверками типа. В функцию, которая должна получать число, может быть передана строка, булево значение или даже объект.

Например:

```
function sum(a, b) {  
  return a + b;  
}  
  
// передадим в функцию для сложения чисел нечисловые значения  
alert( sum(true, { name: "Вася", age: 35 }) ); // true[Object object]
```

Функция «как-то» отработала, но в реальной жизни передача в `sum` подобных значений, скорее всего, будет следствием программной ошибки. Всё-таки `sum` предназначена для суммирования чисел, а не объектов.

Многие языки программирования позволяют прямо в объявлении функции указать, какие типы данных имеют параметры. И это удобно, поскольку повышает надёжность кода.

В JavaScript же проверку типов приходится делать дополнительным кодом в начале функции, который во-первых обычно лень писать, а во-вторых он увеличивает общий объем текста, тем самым ухудшая читаемость.

Декораторы способны упростить рутинные, повторяющиеся задачи, вынести их из кода функции.

Например, создадим декоратор, который принимает функцию и массив, который описывает для какого аргумента какую проверку типа применять:

```
// вспомогательная функция для проверки на число  
function checkNumber(value) {  
  return typeof value == 'number';  
}  
  
// декоратор, проверяющий типы для f  
// второй аргумент checks - массив с функциями для проверки  
function typeCheck(f, checks) {  
  return function() {  
    for (var i = 0; i < arguments.length; i++) {  
      if (!checks[i](arguments[i])) {  
        alert( "Некорректный тип аргумента номер " + i );  
        return;  
      }  
    }  
    return f.apply(this, arguments);  
  }  
}  
  
function sum(a, b) {  
  return a + b;  
}
```



```
// обернём декоратор для проверки
sum = typeCheck(sum, [checkNumber, checkNumber]); // оба аргумента - числа
```

```
// пользуемся функцией как обычно
alert( sum(1, 2) ); // 3, все хорошо
```

```
// а вот так - будет ошибка
sum(true, null); // некорректный аргумент номер 0
sum(1, ["array", "in", "sum?!"]); // некорректный аргумент номер 1
```

Конечно, этот декоратор можно ещё расширять, улучшать, дописывать проверки, но... Вы уже поняли принцип, не правда ли?

Один раз пишем декоратор и дальше просто применяем этот функционал везде, где нужно.

Декоратор проверки доступа

И наконец посмотрим ещё один, последний пример.

Предположим, у нас есть функция `isAdmin()`, которая возвращает `true`, если у посетителя есть права администратора.

Можно создать декоратор `checkPermissionDecorator`, который добавляет в любую функцию проверку прав:

Например, создадим декоратор `checkPermissionDecorator(f)`. Он будет возвращать обертку, которая передает вызов `f` в том случае, если у посетителя достаточно прав:

```
function checkPermissionDecorator(f) {
  return function() {
    if (isAdmin()) {
      return f.apply(this, arguments);
    }
    alert( 'Недостаточно прав' );
  }
}
```

Использование декоратора:

```
function save() { ... }

save = checkPermissionDecorator(save);
// Теперь вызов функции save() проверяет права
```

Итого

Декоратор – это обёртка над функцией, которая модифицирует её поведение. При этом основную работу по-прежнему выполняет функция.

Декораторы можно не только повторно использовать, но и комбинировать!

Это кардинально повышает их выразительную силу. Декораторы можно рассматривать как своего рода «фили» или возможности, которые можно «нацепить» на любую функцию. Можно один, а можно несколько.

Скажем, используя декораторы, описанные выше, можно добавить к функции возможности по проверке типов данных, замеру времени и проверке доступа буквально одной строкой, не залезая при этом в её код, то есть (!) не увеличивая его сложность.

Предлагаю вашему вниманию задачи, которые помогут выяснить, насколько вы разобрались в декораторах. Далее в учебнике мы ещё встретимся с ними.

✔ Задачи

Логирующий декоратор (1 аргумент)

важность: 5

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

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

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

Работать должно так:

```
function work(a) {
  /* ... */ // work - произвольная функция, один аргумент
}

function makeLogging(f, log) { /* ваш код */ }

var log = [];
work = makeLogging(work, log);

work(1); // 1, добавлено в log
work(5); // 5, добавлено в log

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

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Логирующий декоратор (много аргументов)

важность: 3

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

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

Работать должно так:

```
function work(a, b) {
  alert( a + b ); // work - произвольная функция
}

function makeLogging(f, log) { /* ваш код */ }

var log = [];
work = makeLogging(work, log);

work(1, 2); // 3
work(4, 5); // 9

for (var i = 0; i < log.length; i++) {
  var args = log[i]; // массив из аргументов i-го вызова
  alert( 'Лог:' + args.join() ); // "Лог:1,2", "Лог:4,5"
}
```

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Кеширующий декоратор

важность: 5

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

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

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

Должно работать так:

```
function f(x) {
  return Math.random() * x; // random для удобства тестирования
}

function makeCaching(f) { /* ваш код */ }

f = makeCaching(f);

var a, b;

a = f(1);
b = f(1);
alert( a == b ); // true (значение закешировано)

b = f(2);
alert( a == b ); // false, другой аргумент => другое значение
```

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Некоторые другие возможности

Различные возможности JavaScript, которые достаточно важны, но не заслужили отдельного раздела.

Типы данных: `[[Class]]`, `instanceof` и утки

Время от времени бывает удобно создавать так называемые «полиморфные» функции, то есть такие, которые по-разному обрабатывают аргументы, в зависимости от их типа. Например, функция вывода может по-разному форматировать числа и даты.

Для реализации такой возможности нужен способ определить тип переменной.

Оператор `typeof`

Мы уже знакомы с простейшим способом – оператором `typeof`.

Оператор `typeof` надёжно работает с примитивными типами, кроме `null`, а также с функциями. Он возвращает для них тип в виде строки:

```
alert( typeof 1 );           // 'number'
alert( typeof true );      // 'boolean'
alert( typeof "Текст" );   // 'string'
alert( typeof undefined ); // 'undefined'
alert( typeof null );      // 'object' (ошибка в языке)
alert( typeof alert );     // 'function'
```

...Но все объекты, включая массивы и даты для `typeof` – на одно лицо, они имеют один тип `'object'`:

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

Поэтому различить их при помощи `typeof` нельзя, и в этом его основной недостаток.

Секретное свойство `[[Class]]`

Для встроенных объектов есть одна «секретная» возможность узнать их тип, которая связана с методом `toString`.

Во всех встроенных объектах есть специальное свойство `[[Class]]`, в котором хранится информация о его типе или конструкторе.

Оно взято в квадратные скобки, так как это свойство – внутреннее. Явно получить его нельзя, но можно прочитать его «в обход», воспользовавшись методом `toString` стандартного объекта `Object`.

Его внутренняя реализация выводит `[[Class]]` в небольшом обрамлении, как `"[object значение]"`.

Например:

```
var toString = {}.toString;

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

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

var user = { name: "Вася" };
alert( toString.call(user) ); // [object Object]
```

В первой строке мы взяли метод `toString`, принадлежащий именно стандартному объекту `{}`. Нам пришлось это сделать, так как у `Date` и `Array` – свои собственные методы `toString`, которые работают иначе.

Затем мы вызываем этот `toString` в контексте нужного объекта `obj`, и он возвращает его внутреннее, невидимое другими способами, свойство `[[Class]]`.

Для получения `[[Class]]` нужна именно внутренняя реализация `toString` стандартного объекта `Object`, другая не подойдёт.

К счастью, методы в JavaScript – это всего лишь функции-свойства объекта, которые можно скопировать в переменную и применить на другом объекте через `call/apply`. Что мы и делаем для `{}.toString`.

Метод также можно использовать с примитивами:

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

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

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

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

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

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

Для большего удобства можно сделать функцию `getClass`, которая будет возвращать только сам `[[Class]]`:

```
function getClass(obj) {
    return {}.toString.call(obj).slice(8, -1);
}
```

```
alert( getClass(new Date) ); // Date
alert( getClass([1, 2, 3]) ); // Array
```

Заметим, что свойство `[[Class]]` есть и доступно для чтения указанным способом – у всех *встроенных* объектов. Но его нет у объектов, которые создают *наши функции*. Точнее, оно есть, но равно всегда `"Object"`.

Например:

```
function User() {}
var user = new User();
alert( {}.toString.call(user) ); // [object Object], не [object User]
```

Поэтому узнать тип таким образом можно только для встроенных объектов.

Метод `Array.isArray()`

Для проверки типа на массив есть специальный метод: `Array.isArray(arr)`. Он возвращает `true` только если `arr` – массив:

```
alert( Array.isArray([1,2,3]) ); // true
alert( Array.isArray("not array") ); // false
```

Но этот метод – единственный в своём роде.

Других аналогичных, типа `Object.isObject`, `Date.isDate` – нет.

Оператор `instanceof`

Оператор `instanceof` позволяет проверить, создан ли объект данной функцией, причём работает для любых функций – как встроенных, так и наших.

```
function User() {}
var user = new User();
alert( user instanceof User ); // true
```

Таким образом, `instanceof`, в отличие от `[[Class]]` и `typeof` может помочь выяснить тип для новых объектов, созданных нашими конструкторами.

Заметим, что оператор `instanceof` – сложнее, чем кажется. Он учитывает наследование, которое мы пока не проходили, но скоро изучим, и затем вернёмся к `instanceof` в главе [Проверка класса: "instanceof"](#).

Утиная типизация

Альтернативный подход к типу – «утиная типизация», которая основана на одной известной поговорке: *«If it looks like a duck, swims like a duck and quacks like a duck, then it probably is a duck (who cares what it really is)»*.

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

Смысл утиной типизации – в проверке необходимых методов и свойств.

Например, мы можем проверить, что объект – массив, не вызывая `Array.isArray`, а просто уточнив наличие важного для нас метода, например `splice`:

```
var something = [1, 2, 3];
if (something.splice) {
  alert( 'Это утка! То есть, массив!' );
}
```

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

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

```
var x = new Date();
if (x.getTime) {
  alert( 'Дата!' );
  alert( x.getTime() ); // работаем с датой
}
```

С виду такая проверка хрупка, ее можно «сломать», передав похожий объект с тем же методом.

Но как раз в этом и есть смысл утиной типизации: если объект похож на дату, у него есть методы даты, то будем работать с ним как с датой (какая разница, что это на самом деле).

То есть, мы намеренно позволяем передать в код нечто менее конкретное, чем определённый тип, чтобы сделать его более универсальным.

Проверка интерфейса

Если говорить словами «классического программирования», то «duck typing» – это проверка реализации объектом требуемого интерфейса. Если реализует – ок, используем его. Если нет – значит это что-то другое.

Пример полиморфной функции

Пример полиморфной функции – `sayHi(who)`, которая будет говорить «Привет» своему аргументу, причём если передан массив – то «Привет» каждому:

```
function sayHi(who) {
  if (Array.isArray(who)) {
    who.forEach(sayHi);
  } else {
    alert( 'Привет, ' + who );
  }
}

// Вызов с примитивным аргументом
sayHi("Вася"); // Привет, Вася

// Вызов с массивом
sayHi(["Саша", "Петя"]); // Привет, Саша... Петя

// Вызов с вложенными массивами - тоже работает!
sayHi(["Саша", "Петя", ["Маша", "Юля"]]); // Привет Саша..Петя..Маша..Юля
```

Проверку на массив в этом примере можно заменить на «утиную» – нам ведь нужен только метод `forEach`:

```
function sayHi(who) {
  if (who.forEach) { // если есть forEach
    who.forEach(sayHi); // предполагаем, что он ведёт себя "как надо"
  } else {
    alert( 'Привет, ' + who );
  }
}
```

Итого

Для написания полиморфных (это удобно!) функций нам нужна проверка типов.

- Для примитивов с ней отлично справляется оператор `typeof`.
У него две особенности:
 - Он считает `null` объектом, это внутренняя ошибка в языке.
 - Для функций он возвращает `function`, по стандарту функция не считается базовым типом, но на практике это удобно и полезно.
- Для встроенных объектов мы можем получить тип из скрытого свойства `[[Class]]`, при помощи вызова `{}.toString.call(obj).slice(8, -1)`. Для конструкторов, которые объявлены нами, `[[Class]]` всегда равно `"Object"`.
- Оператор `obj instanceof Func` проверяет, создан ли объект `obj` функцией `Func`, работает для любых конструкторов. Более подробно мы разберём его в главе [Проверка класса: "instanceof"](#).
- И, наконец, зачастую достаточно проверить не сам тип, а просто наличие нужных свойств или методов. Это называется «утиная типизация».

Задачи

Полиморфная функция `formatDate`

важность: 5

Напишите функцию `formatDate(date)`, которая возвращает дату в формате `dd.mm.yy`.

Ее первый аргумент должен содержать дату в одном из видов:

1. Как объект `Date`.
2. Как строку, например `уууу-мм-дд` или другую в стандартном формате даты.
3. Как число секунд с `01.01.1970`.
4. Как массив `[гггг, мм, дд]`, месяц начинается с нуля

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

Пример работы:

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

alert( formatDate('2011-10-02') ); // 02.10.11
alert( formatDate(1234567890) ); // 14.02.09
```

```
alert( formatDate([2014, 0, 1]) ); // 01.01.14
alert( formatDate(new Date(2014, 0, 1)) ); // 01.01.14
```

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Формат JSON, метод toJSON

В этой главе мы рассмотрим работу с форматом [JSON](#), который используется для представления объектов в виде строки.

Это один из наиболее удобных форматов данных при взаимодействии с JavaScript. Если нужно с сервера взять объект с данными и передать на клиенте, то в качестве промежуточного формата – для передачи по сети, почти всегда используют именно его.

В современных браузерах есть замечательные методы, знание тонкостей которых делает операции с JSON простыми и комфортными.

Формат JSON

Данные в формате JSON ([RFC 4627](#)) представляют собой:

- JavaScript-объекты { ... } или
- Массивы [...] или
- Значения одного из типов:
 - строки в двойных кавычках,
 - число,
 - логическое значение true / false ,
 - null .

Почти все языки программирования имеют библиотеки для преобразования объектов в формат JSON.

Основные методы для работы с JSON в JavaScript – это:

- `JSON.parse` – читает объекты из строки в формате JSON.
- `JSON.stringify` – превращает объекты в строку в формате JSON, используется, когда нужно из JavaScript передать данные по сети.

Метод JSON.parse

Вызов `JSON.parse(str)` превратит строку с данными в формате JSON в JavaScript-объект/массив/значение.

Например:

```
var numbers = "[0, 1, 2, 3]";
numbers = JSON.parse(numbers);
alert( numbers[1] ); // 1
```

Или так:

```
var user = '{ "name": "Вася", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';
user = JSON.parse(user);
alert( user.friends[1] ); // 1
```

Данные могут быть сколь угодно сложными, объекты и массивы могут включать в себя другие объекты и массивы. Главное чтобы они соответствовали формату.

⚠ JSON-объекты ≠ JavaScript-объекты

Объекты в формате JSON похожи на обычные JavaScript-объекты, но отличаются от них более строгими требованиями к строкам – они должны быть именно в двойных кавычках.

В частности, первые два свойства объекта ниже – некорректны:

```
{
  name: "Вася", // ошибка: ключ name без кавычек!
  "surname": 'Петров', // ошибка: одинарные кавычки у значения 'Петров'!
  "age": 35, // .. а тут всё в порядке.
  "isAdmin": false // и тут тоже всё ок
}
```

Кроме того, в формате JSON не поддерживаются комментарии. Он предназначен только для передачи данных.

Есть нестандартное расширение формата JSON, которое называется [JSON5](#) и как раз разрешает ключи без кавычек, комментарии и т.п, как в обычном JavaScript. На данном этапе, это отдельная библиотека.

Умный разбор: JSON.parse(str, reviver)

Метод `JSON.parse` поддерживает и более сложные алгоритмы разбора.

Например, мы получили с сервера объект с данными события `event`.

Он выглядит так:

```
// title: название события, date: дата события
var str = '{"title":"Конференция","date":"2014-11-30T12:00:00.000Z"}';
```

...И теперь нужно *восстановить* его, то есть превратить в JavaScript-объект.

Попробуем вызвать для этого `JSON.parse`:

```
var str = '{"title":"Конференция","date":"2014-11-30T12:00:00.000Z"}';
var event = JSON.parse(str);

alert( event.date.getDate() ); // ошибка!
```

...Увы, ошибка!

Дело в том, что значением `event.date` является строка, а отнюдь не объект `Date`. Откуда методу `JSON.parse` знать, что нужно превратить строку именно в дату?

Для интеллектуального восстановления из строки у `JSON.parse(str, reviver)` есть второй параметр `reviver`, который является функцией `function(key, value)`.

Если она указана, то в процессе чтения объекта из строки `JSON.parse` передаёт ей по очереди все создаваемые пары ключ-значение и может вернуть либо преобразованное значение, либо `undefined`, если его нужно пропустить.

В данном случае мы можем создать правило, что ключ `date` всегда означает дату:

```
// дата в строке - в формате UTC
var str = '{"title":"Конференция","date":"2014-11-30T12:00:00.000Z"}';

var event = JSON.parse(str, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});

alert( event.date.getDate() ); // теперь работает!
```

Кстати, эта возможность работает и для вложенных объектов тоже:

```
var schedule = '{
  \ "events": [ \
    { "title": "Конференция", "date": "2014-11-30T12:00:00.000Z", \
      "title": "День рождения", "date": "2015-04-18T12:00:00.000Z" } \
    ] \
}';

schedule = JSON.parse(schedule, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});

alert( schedule.events[1].date.getDate() ); // работает!
```

Сериализация, метод JSON.stringify

Метод `JSON.stringify(value, replacer, space)` преобразует («сериализует») значение в JSON-строку.

Пример использования:

```
var event = {
  title: "Конференция",
  date: "сегодня"
};

var str = JSON.stringify(event);
alert( str ); // {"title":"Конференция","date":"сегодня"}

// Обратное преобразование.
event = JSON.parse(str);
```

При сериализации объекта вызывается его метод `toJSON`.

Если такого метода нет – перечисляются его свойства, кроме функций.

Посмотрим это в примере посложнее:

```
var room = {
  number: 23,
  occupy: function() {
    alert( this.number );
  }
};

event = {
  title: "Конференция",
  date: new Date(Date.UTC(2014, 0, 1)),
  room: room
};

alert( JSON.stringify(event) );
/*
  {
    "title": "Конференция",
    "date": "2014-01-01T00:00:00.000Z", // (1)
    "room": {"number": 23}          // (2)
  }
*/
```

Обратим внимание на два момента:

1. Дата превратилась в строку. Это не случайно: у всех дат есть встроенный метод `toJSON`. Его результат в данном случае – строка в таймзоне UTC.
2. У объекта `room` нет метода `toJSON`. Поэтому он сериализуется перечислением свойств.

Мы, конечно, могли бы добавить такой метод, тогда в итог попал бы его результат:

```
var room = {
  number: 23,
  toJSON: function() {
    return this.number;
  }
};

alert( JSON.stringify(room) ); // 23
```

Исключение свойств

Попытаемся преобразовать в JSON объект, содержащий ссылку на DOM.

Например:

```
var user = {
  name: "Вася",
  age: 25,
  window: window
};

alert( JSON.stringify(user) ); // ошибка!
// TypeError: Converting circular structure to JSON (текст из Chrome)
```

Произошла ошибка! В чём же дело, неужели некоторые объекты запрещены? Как видно из текста ошибки – дело совсем в другом. Глобальный объект `window` – сложная структура с кучей встроенных свойств и круговыми ссылками, поэтому его преобразовать невозможно. Да и нужно ли?

Во втором параметре `JSON.stringify(value, replacer)` можно указать массив свойств, которые подлежат сериализации.

Например:

```
var user = {
  name: "Вася",
  age: 25,
  window: window
};

alert( JSON.stringify(user, ["name", "age"]) );
// {"name":"Вася","age":25}
```

Для более сложных ситуаций вторым параметром можно передать функцию `function(key, value)`, которая возвращает сериализованное `value` либо `undefined`, если его не нужно включать в результат:


```
var user = {
  name: "Вася",
  age: 25,
  window: window
};
```

```
var str = JSON.stringify(user, function(key, value) {
  if (key == 'window') return undefined;
  return value;
});
```

```
alert( str ); // {"name":"Вася","age":25}
```

В примере выше функция пропустит свойство с названием `window`. Для остальных она просто возвращает значение, передавая его стандартному алгоритму. А могла бы и как-то обработать.

Функция `replacer` работает рекурсивно

То есть, если объект содержит вложенные объекты, массивы и т.п., то все они пройдут через `replacer`.

Красивое форматирование

В методе `JSON.stringify(value, replacer, space)` есть ещё третий параметр `space`.

Если он является числом – то уровни вложенности в JSON оформляются указанным количеством пробелов, если строкой – вставляется эта строка.

Например:

```
var user = {
  name: "Вася",
  age: 25,
  roles: {
    isAdmin: false,
    isEditor: true
  }
};
```

```
var str = JSON.stringify(user, "", 4);
```

```
alert( str );
/* Результат -- красиво сериализованный объект:
{
  "name": "Вася",
  "age": 25,
  "roles": {
    "isAdmin": false,
    "isEditor": true
  }
}
*/
```

Итого

- JSON – формат для представления объектов (и не только) в виде строки.
- Методы [JSON.parse](#) и [JSON.stringify](#) позволяют интеллектуально преобразовать объект в строку и обратно.

Задачи

Превратите объект в JSON

важность: 3

Превратите объект `leader` из примера ниже в JSON:

```
var leader = {
  name: "Василий Иванович",
  age: 35
};
```

После этого прочитайте получившуюся строку обратно в объект.

[К решению](#)

Превратите объекты со ссылками в JSON

важность: 3

Превратите объект `team` из примера ниже в JSON:

```
var leader = {
  name: "Василий Иванович"
};

var soldier = {
  name: "Петька"
```

```
};  
  
// эти объекты ссылаются друг на друга!  
leader.soldier = soldier;  
soldier.leader = leader;  
  
var team = [leader, soldier];
```

1. Может ли это сделать прямой вызов `JSON.stringify(team)` ? Если нет, то почему?
2. Какой подход вы бы предложили для чтения и восстановления таких объектов?

[К решению](#)

setTimeout и setInterval

Почти все реализации JavaScript имеют внутренний таймер-планировщик, который позволяет задавать вызов функции через заданный период времени.

В частности, эта возможность поддерживается в браузерах и в сервере Node.JS.

setTimeout

Синтаксис:

```
var timerId = setTimeout(func / code, delay[, arg1, arg2...])
```

Параметры:

func/code

Функция или строка кода для исполнения. Строка поддерживается для совместимости, использовать её не рекомендуется.

delay

Задержка в миллисекундах, 1000 миллисекунд равны 1 секунде.

arg1 , arg2 ...

Аргументы, которые нужно передать функции. Не поддерживаются в IE9-.

Исполнение функции произойдёт спустя время, указанное в параметре `delay` .

Например, следующий код вызовет `func()` через одну секунду:

```
function func() {  
  alert( 'Привет' );  
}
```

```
setTimeout(func, 1000);
```

С передачей аргументов (не сработает в IE9-):

```
function func(phrase, who) {  
  alert( phrase + ', ' + who );  
}
```

```
setTimeout(func, 1000, "Привет", "Вася"); // Привет, Вася
```

Если первый аргумент является строкой, то интерпретатор создаёт анонимную функцию из этой строки.

То есть такая запись тоже сработает:

```
setTimeout("alert('Привет')", 1000);
```

Однако, использование строк не рекомендуется, так как они могут вызвать проблемы при минимизации кода, и, вообще, сама возможность использовать строку сохраняется лишь для совместимости.

Вместо них используйте анонимные функции, вот так:

```
setTimeout(function() { alert('Привет') }, 1000);
```

Отмена исполнения clearTimeout

Функция `setTimeout` возвращает числовой идентификатор таймера `timerId` , который можно использовать для отмены действия.

Синтаксис:

```
var timerId = setTimeout(...);  
clearTimeout(timerId);
```

В следующем примере мы ставим таймаут, а затем удаляем (передумали). В результате ничего не происходит.

```
var timerId = setTimeout(function() { alert(1) }, 1000);
alert(timerId); // число - идентификатор таймера

clearTimeout(timerId);
alert(timerId); // всё ещё число, оно не обнуляется после отмены
```

Как видно из `alert`, в браузере идентификатор таймера является обычным числом. Другие JavaScript-окружения, например Node.JS, могут возвращать объект таймера, с дополнительными методами.

Такие разночтения вполне соответствуют стандарту просто потому, что в спецификации JavaScript про таймеры нет ни слова.

Таймеры – это надстройка над JavaScript, которая описана в [секции Timers](#) стандарта HTML5 для браузеров и в [документации к Node.JS](#) – для сервера.

setInterval

Метод `setInterval` имеет синтаксис, аналогичный `setTimeout`.

```
var timerId = setInterval(func / code, delay[, arg1, arg2...])
```

Смысл аргументов – тот же самый. Но, в отличие от `setTimeout`, он запускает выполнение функции не один раз, а регулярно повторяет её через указанный интервал времени. Остановить исполнение можно вызовом `clearInterval(timerId)`.

Следующий пример при запуске станет выводить сообщение каждые две секунды, пока не пройдет 5 секунд:

```
// начать повторы с интервалом 2 сек
var timerId = setInterval(function() {
  alert( "тик" );
}, 2000);

// через 5 сек остановить повторы
setTimeout(function() {
  clearInterval(timerId);
  alert( 'стоп' );
}, 5000);
```

Модальные окна замораживают время в Chrome/Opera/Safari

Что будет, если долго не жать OK на появившемся `alert`? Это зависит от браузера.

В браузерах Chrome, Opera и Safari внутренний таймер «заморожен» во время показа `alert/confirm/prompt`. А вот в IE и Firefox внутренний таймер продолжит идти.

Поэтому, если закрыть `alert` после небольшой паузы, то в Firefox/IE следующий `alert` будет показан сразу же (время подошло), а в Chrome/Opera/Safari – только через 2 секунды после закрытия.

Рекурсивный setTimeout

Важная альтернатива `setInterval` – рекурсивный `setTimeout`:

```
/** вместо:
var timerId = setInterval(function() {
  alert( "тик" );
}, 2000);
*/

var timerId = setTimeout(function tick() {
  alert( "тик" );
  timerId = setTimeout(tick, 2000);
}, 2000);
```

В коде выше следующее выполнение планируется сразу после окончания предыдущего.

Рекурсивный `setTimeout` – более гибкий метод тайминга, чем `setInterval`, так как время до следующего выполнения можно запланировать по-разному, в зависимости от результатов текущего.

Например, у нас есть сервис, который в 5 секунд опрашивает сервер на предмет новых данных. В случае, если сервер перегружен, можно увеличивать интервал опроса до 10, 20, 60 секунд... А потом вернуть обратно, когда всё нормализуется.

Если у нас регулярно проходят грузящие процессор задачи, то мы можем оценивать время, потраченное на их выполнение, и планировать следующий запуск раньше или позже.

Рекурсивный `setTimeout` гарантирует паузу между вызовами, `setInterval` – нет.

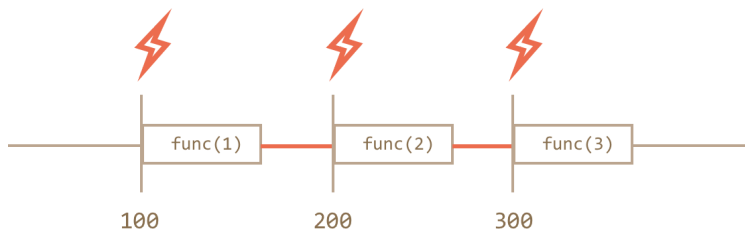
Давайте сравним два кода. Первый использует `setInterval`:

```
var i = 1;
setInterval(function() {
  func(i);
}, 100);
```

Второй использует рекурсивный `setTimeout` :

```
var i = 1;
setTimeout(function run() {
  func(i);
  setTimeout(run, 100);
}, 100);
```

При `setInterval` внутренний таймер будет срабатывать чётко каждые 100 мс и вызывать `func(i)` :



Вы обратили внимание?...

Реальная пауза между вызовами `func` при `setInterval` меньше, чем указана в коде!

Это естественно, ведь время работы функции никак не учитывается, оно «съедает» часть интервала.

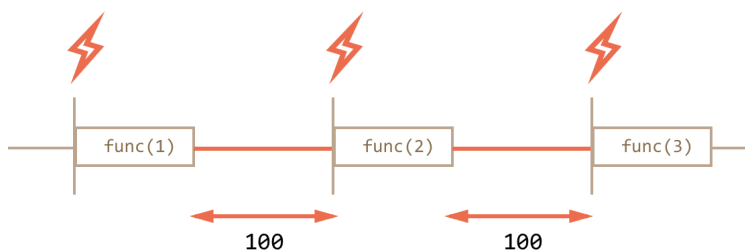
Возможно и такое что `func` оказалась сложнее, чем мы рассчитывали и выполнялась дольше, чем 100 мс.

В этом случае интерпретатор будет ждать, пока функция завершится, затем проверит таймер и, если время вызова `setInterval` уже подошло (или прошло), то следующий вызов произойдёт сразу же.

Если функция и выполняется дольше, чем пауза `setInterval`, то вызовы будут происходить вообще без перерыва.

Исключением является IE, в котором таймер «застывает» во время выполнения JavaScript.

А так будет выглядеть картинка с рекурсивным `setTimeout` :



При рекурсивном `setTimeout` задержка всегда фиксирована и равна 100 мс.

Это происходит потому, что каждый новый запуск планируется только после окончания текущего.

i Управление памятью

Сборщик мусора в JavaScript не чистит функции, назначенные в таймерах, пока таймеры актуальны.

При передаче функции в `setInterval/setTimeout` создаётся внутренняя ссылка на неё, через которую браузер её будет запускать, и которая препятствует удалению из памяти, даже если функция анонимна.

```
// Функция будет жить в памяти, пока не сработал (или не был очищен) таймер
setTimeout(function() {}, 100);
```

- Для `setTimeout` – внутренняя ссылка исчезнет после исполнения функции.
- Для `setInterval` – ссылка исчезнет при очистке таймера.

Так как функция также тянет за собой всё замыкание, то ставшие неактуальными, но не отменённые `setInterval` могут приводить к излишним тратам памяти.

Минимальная задержка таймера

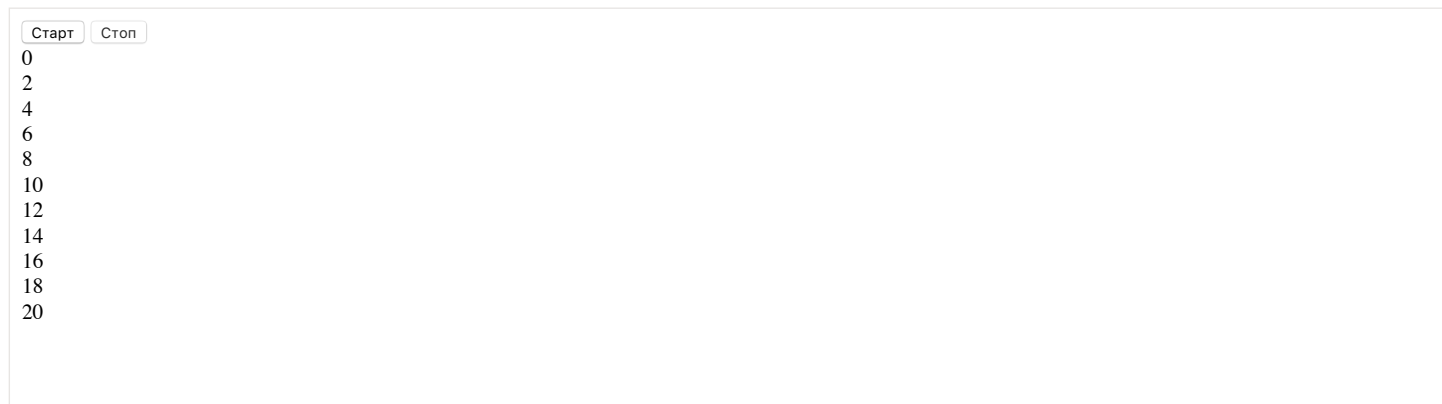
У браузерного таймера есть минимальная возможная задержка. Она меняется от примерно нуля до 4 мс в современных браузерах. В более старых она может быть больше и достигать 15 мс.

По [стандарту](#), минимальная задержка составляет 4 мс. Так что нет разницы между `setTimeout(..,1)` и `setTimeout(..,4)`.

Посмотреть минимальное разрешение «вживую» можно на следующем примере.

В примере ниже каждая полоска удлиняется вызовом `setInterval` с указанной на ней задержкой – от 0 мс (сверху) до 20 мс (внизу).

Позапускайте его в различных браузерах. Вы заметите, что несколько первых полосок анимируются с одинаковой скоростью. Это как раз потому, что слишком маленькие задержки таймер не различает.



⚠ Важно:

В Internet Explorer, нулевая задержка `setInterval(..., 0)` не работает. Это касается именно `setInterval`, т.е. `setTimeout(..., 0)` работает нормально.

ℹ Откуда взялись эти 4 мс?

Почему минимальная задержка – 4 мс, а не 1 мс? Зачем она вообще существует?

Это – «привет» от прошлого. Браузер Chrome как-то пытался убрать минимальную задержку в своих ранних версиях, но оказалось, что существуют сайты, которые используют `setTimeout(..., 0)` рекурсивно, создавая тем самым «асинхронный цикл». И, если задержку совсем убрать, то будет 100% загрузка процессора, такой сайт «подвесит» браузер.

Поэтому, чтобы не ломать существующие скрипты, решили сделать задержку. По возможности, небольшую. На время создания стандарта оптимальным числом показалось 4 мс.

Реальная частота срабатывания

В ряде ситуаций таймер будет срабатывать реже, чем обычно. Задержка между вызовами `setInterval(..., 4)` может быть не 4 мс, а 30 мс или даже 1000 мс.

- Большинство браузеров (десктопных в первую очередь) продолжают выполнять `setTimeout/setInterval`, даже если вкладка неактивна.
При этом ряд из них (Chrome, FF, IE10) снижают минимальную частоту таймера, до 1 раза в секунду. Получается, что в «фоновой» вкладке будет срабатывать таймер, но редко.
- При работе от батареи, в ноутбуке – браузеры тоже могут снижать частоту, чтобы реже выполнять код и экономить заряд батареи. Особенно этим известен IE. Снижение может достигать нескольких раз, в зависимости от настроек.
- При слишком большой загрузке процессора JavaScript может не успевать обрабатывать таймеры вовремя. При этом некоторые запуски `setInterval` будут пропущены.

Вывод: на частоту 4 мс стоит ориентироваться, но не стоит рассчитывать.

Разбивка долгих скриптов

Нулевой или небольшой таймаут также используют, чтобы разорвать поток выполнения «тяжелых» скриптов.

Например, скрипт для подсветки синтаксиса должен проанализировать код, создать много цветных элементов для подсветки и добавить их в документ – на большом файле это займёт много времени, браузер может даже подвиснуть, что неприемлемо.

Для того, чтобы этого избежать, сложная задача разбивается на части, выполнение каждой части запускается через мини-интервал после предыдущей, чтобы дать браузеру время.

Например, осуществляется анализ и подсветка первых 100 строк, затем через 20 мс – следующие 100 строк и так далее. При этом можно подстраиваться под CPU посетителя: замерять время на анализ 100 строк и, если процессор хороший, то в следующий раз обработать 200 строк, а если плохой – то 50. В итоге подсветка будет работать с адекватной быстротой и без тормозов на любых текстах и компьютерах.

Итого

- Методы `setInterval(func, delay)` и `setTimeout(func, delay)` позволяют запускать `func` регулярно/один раз через `delay` миллисекунд.
- Оба метода возвращают идентификатор таймера. Его используют для остановки выполнения вызовом `clearInterval/clearTimeout`.
- В случаях, когда нужно гарантировать задержку между регулярными вызовами или гибко её менять, вместо `setInterval` используют рекурсивный `setTimeout`.
- Минимальная задержка по стандарту составляет 4 мс. Браузеры соблюдают этот стандарт, но некоторые другие среды для выполнения JS, например Node.js, могут предоставить и меньше задержки.

- В реальности срабатывания таймера могут быть гораздо реже, чем назначено, например если процессор перегружен, вкладка находится в фоновом режиме, ноутбук работает от батареи или по какой-то иной причине.

Браузерных особенностей почти нет, разве что вызов `setInterval(..., 0)` с нулевой задержкой в IE недопустим, нужно указывать `setInterval(..., 1)`.

✔ Задачи

Вывод чисел каждые 100 мс

важность: 5

Напишите функцию `printNumbersInterval()`, которая последовательно выводит в консоль числа от 1 до 20, с интервалом между числами 100 мс. То есть, весь вывод должен занимать 2000 мс, в течение которых каждые 100 мс в консоли появляется очередное число.

Нажмите на кнопку, открыв консоль, для демонстрации:

```
printNumbersInterval()
```

P.S. Функция должна использовать `setInterval`.

[К решению](#)

Вывод чисел каждые 100 мс, через `setTimeout`

важность: 5

Сделайте то же самое, что в задаче [Вывод чисел каждые 100 мс](#), но с использованием рекурсивного `setTimeout` вместо `setInterval`.

[К решению](#)

Для подсветки `setInterval` или `setTimeout`?

важность: 5

Стоит задача: реализовать подсветку синтаксиса в длинном коде при помощи JavaScript, для онлайн-редактора кода. Это требует сложных вычислений, особенно загружает процессор генерация дополнительных элементов страницы, визуально осуществляющих подсветку.

Поэтому решаем обрабатывать не весь код сразу, что привело бы к зависанию скрипта, а разбить работу на части: подсвечивать по 20 строк раз в 10 мс.

Как мы знаем, есть два варианта реализации такой подсветки:

1. Через `setInterval`, с остановкой по окончании работы:

```
timer = setInterval(function() {
  if (есть еще что подсветить) highlight();
  else clearInterval(timer);
}, 10);
```

2. Через рекурсивный `setTimeout`:

```
setTimeout(function go() {
  highlight();
  if (есть еще что подсветить) setTimeout(go, 10);
}, 10);
```

Какой из них стоит использовать? Почему?

[К решению](#)

Что выведет `setTimeout`?

важность: 5

В коде ниже запланирован запуск `setTimeout`, а затем запущена тяжелая функция `hardWork`, выполнение которой занимает более долгое время, чем интервал до срабатывания таймера.

Когда сработает `setTimeout`? Выберите нужный вариант:

1. До выполнения `hardWork`.
2. Во время выполнения `hardWork`.
3. Сразу же по окончании `hardWork`.
4. Через 100 мс после окончания `hardWork`.

Что выведет `alert` в коде ниже?

```
setTimeout(function() {
  alert( i );
}, 100);

var i;

function hardWork() {
  // время выполнения этого кода >100 мс, сам код неважен
  for (i = 0; i < 1e8; i++) hardWork[i % 2] = i;
}

hardWork();
```

[К решению](#)

Что выведет после `setInterval`?

важность: 5

В коде ниже запускается `setInterval` каждые 10 мс, и через 50 мс запланирована его отмена.

После этого запущена тяжёлая функция `f`, выполнение которой (мы точно знаем) потребует более 100 мс.

Сработает ли `setInterval`, как и когда?

Варианты:

1. Да, несколько раз, *в процессе* выполнения `f`.
2. Да, несколько раз, *сразу после* выполнения `f`.
3. Да, один раз, *сразу после* выполнения `f`.
4. Нет, не сработает.
5. Может быть по-разному, как повезёт.

Что выведет `alert` в строке (*)?

```
var i;
var timer = setInterval(function() { // планируем setInterval каждые 10 мс
  i++;
}, 10);

setTimeout(function() { // через 50 мс - отмена setInterval
  clearInterval(timer);
  alert( i ); // (*)
}, 50);

// и запускаем тяжёлую функцию
function f() {
  // точное время выполнения не играет роли
  // здесь оно заведомо больше 100 мс
  for (i = 0; i < 1e8; i++) f[i % 2] = i;
}

f();
```

[К решению](#)

Кто быстрее?

важность: 5

Есть два бегуна:

```
var runner1 = new Runner();
var runner2 = new Runner();
```

У каждого есть метод `step()`, который делает шаг, увеличивая свойство `steps`.

Конкретный код метода `step()` не имеет значения, важно лишь что шаг делается не мгновенно, он требует небольшого времени.

Если запустить первого бегуна через `setInterval`, а второго – через вложенный `setTimeout` – какой сделает больше шагов за 5 секунд?

```
// первый?
setInterval(function() {
  runner1.step();
}, 15);

// или второй?
setTimeout(function go() {
  runner2.step();
  setTimeout(go, 15);
}, 15);

setTimeout(function() {
  alert( runner1.steps );
});
```

```
    alert( runner2.steps );
  }, 5000);
```

[К решению](#)

Функция-задержка

важность: 5

Напишите функцию `delay(f, ms)`, которая возвращает обёртку вокруг `f`, задерживающую вызов на `ms` миллисекунд.

Например:

```
function f(x) {
  alert( x );
}

var f1000 = delay(f, 1000);
var f1500 = delay(f, 1500);

f1000("тест"); // выведет "тест" через 1000 миллисекунд
f1500("тест2"); // выведет "тест2" через 1500 миллисекунд
```

Упрощённо можно сказать, что `delay` возвращает "задержанный на `ms`" вариант `f`.

В примере выше у функции только один аргумент, но `delay` должна быть универсальной: передавать любое количество аргументов и контекст `this`.

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

Вызов не чаще чем в N миллисекунд

важность: 5

Напишите функцию `debounce(f, ms)`, которая возвращает обёртку, которая передаёт вызов `f` не чаще, чем раз в `ms` миллисекунд.

«Лишние» вызовы игнорируются. Все аргументы и контекст – передаются.

Например:

```
function f() { ... }

var f = debounce(f, 1000);

f(1); // выполнится сразу же
f(2); // игнор

setTimeout( function() { f(3) }, 100); // игнор (прошло только 100 мс)
setTimeout( function() { f(4) }, 1100); // выполнится
setTimeout( function() { f(5) }, 1500); // игнор
```

Упрощённо можно сказать, что `debounce` возвращает вариант `f`, срабатывающий не чаще чем раз в `ms` миллисекунд.

[Открыть песочницу с тестами для задачи.](#) ↗

[К решению](#)

Тормозилка

важность: 5

Напишите функцию `throttle(f, ms)` – «тормозилку», которая возвращает обёртку, передающую вызов `f` не чаще, чем раз в `ms` миллисекунд.

У этой функции должно быть важное существенное отличие от `debounce`: если игнорируемый вызов оказался последним, т.е. после него до окончания задержки ничего нет – то он выполнится.

Чтобы лучше понять, откуда взялось это требование, и как `throttle` должна работать – разберём реальное применение, на которое и ориентирована эта задача.

Например, нужно обрабатывать передвижения мыши.

В JavaScript это делается функцией, которая будет запускаться при каждом микро-передвижении мыши и получать координаты курсора. По мере того, как мышь двигается, эта функция может запускаться очень часто, может быть 100 раз в секунду (каждые 10 мс).

Функция обработки передвижения должна обновлять некую информацию на странице.

При этом обновление – слишком «тяжёлый» процесс, чтобы делать его при каждом микро-передвижении. Имеет смысл делать его раз в 100 мс, не чаще.

Пусть функция, которая осуществляет это обновление по передвижению, называется `onmousemove`.

Вызов `throttle(onmousemove, 100)`, по сути, предназначен для того, чтобы «притормаживать» обработку `onmousemove`. Технически, он должен возвращать обёртку, которая передаёт все вызовы `onmousemove`, но не чаще чем раз в 100 мс.

При этом промежуточные движения можно игнорировать, но мышь в конце концов где-то остановится. И это последнее, итоговое положение мыши обязательно нужно обработать!

Визуально это даст следующую картину обработки перемещений мыши:

1. Первое обновление произойдёт сразу (это важно, посетитель тут же видит реакцию на своё действие).
2. Далее может быть много вызовов (микро-передвижений) с разными координатами, но пока не пройдёт 100 мс – ничего не будет.
3. По истечении 100 мс – опять обновление, с последними координатами. Промежуточные микро-передвижения игнорированы.
4. В конце концов мышь где-то остановится, обновление по окончании очередной паузы 100 мс сработает с последними координатами.

Ещё раз заметим – задача из реальной жизни, и в ней принципиально важно, что *последнее* передвижение обрабатывается. Пользователь должен увидеть, где остановил мышь.

Пример использования:

```
var f = function(a) {
  console.log(a)
};

// затормозить функцию до одного раза в 1000 мс
var f1000 = throttle(f, 1000);

f1000(1); // выведет 1
f1000(2); // (тормозим, не прошло 1000 мс)
f1000(3); // (тормозим, не прошло 1000 мс)

// когда пройдёт 1000 мс...
// выведет 3, промежуточное значение 2 игнорируется
```

[Открыть песочницу с тестами для задачи.](#)

[К решению](#)

Запуск кода из строки: eval

Функция `eval(code)` позволяет выполнить код, переданный ей в виде строки.

Этот код будет выполнен в *текущей области видимости*.

Использование eval

В простейшем случае `eval` всего лишь выполняет код, например:

```
var a = 1;

(function() {
  var a = 2;

  eval(' alert(a) '); // 2

})();
```

Но он может не только выполнить код, но и вернуть результат.

Вызов `eval` возвращает последнее вычисленное выражение:

Например:

```
alert( eval('1+1') ); // 2
```

При вызове `eval` имеет полный доступ к локальным переменным.

Это означает, что текущие переменные могут быть изменены или дополнены:

```
var x = 5;
eval(" alert( x ); x = 10"); // 5, доступ к старому значению
alert( x ); // 10, значение изменено внутри eval
```

i В строгом режиме `eval` имеет свою область видимости

В строгом режиме функционал `eval` чуть-чуть меняется.

При `use strict` код внутри `eval` по-прежнему сможет читать и менять внешние переменные, однако переменные и функции, объявленные внутри `eval`, не попадут наружу.

```
"use strict";
```

```
eval("var a = 5; function f() { }");  
alert( a ); // ошибка, переменная не определена  
// функция f тоже не видна снаружи
```

Иными словами, в новом стандарте `eval` имеет свою область видимости, а к внешним переменным обращается через замыкание, аналогично тому, как работают обычные функции.

Неграмотное использование `eval`

Начнём с того, что `eval` применяется очень редко. Действительно редко. Есть даже такое выражение «`eval is evil`» (`eval` – зло).

Причина проста: когда-то JavaScript был гораздо более слабым языком, чем сейчас, и некоторые вещи без `eval` было сделать невозможно. Но те времена давно прошли. И теперь найти тот случай, когда действительно надо выполнить код из строки – это надо постараться.

Но если вы действительно знаете, что это именно тот случай и вам необходим `eval` – есть ряд вещей, которые нужно иметь в виду.

Доступ к локальным переменным – худшее, что можно сделать при `eval`.

Дело в том, что локальные переменные могут быть легко переименованы:

```
function sayHi() {  
  var phrase = "Привет";  
  eval(str);  
}
```

Переменная `phrase` может быть переименована в `hello`, и если строка `str` обращается к ней – будет ошибка.

Современные средства сжатия JavaScript переименовывают локальные переменные автоматически. Это считается безопасным, так как локальная переменная видна лишь внутри функции и если в ней везде поменять `phrase` на `p`, то никто этого не заметит.

До сжатия:

```
function sayHi() {  
  var phrase = "Привет";  
  alert( phrase );  
}
```

После сжатия:

```
function sayHi() {  
  var a = "Привет";  
  alert( a );  
}
```

На самом деле всё ещё проще – в данном случае утилита сжатия автоматически уберёт переменную `a` и код станет таким:

```
function sayHi() {  
  alert( "Привет" );  
}
```

Итак, если где-то в функции есть `eval`, то его взаимодействие с локальными переменными будет нарушено с непредсказуемыми побочными эффектами.

Некоторые инструменты сжатия предупреждают, когда видят `eval` или стараются вообще не сжимать такой код вместе с его внешними функциями, но всё это борьба с последствиями кривого кода.

Как правило, `eval` не нужен, именно поэтому говорят, «`eval is evil`».

Запуск скрипта в глобальной области

Ок, взаимодействовать с локальными переменными нельзя.

Но допустим мы загрузили с сервера или вручную сгенерировали скрипт, который нужно выполнить. Желательно, в глобальной области, вне любых функций, чтобы он уж точно к локальным переменным отношения не имел.

Здесь `eval` может пригодиться. Есть два трюка для выполнения кода в глобальной области:

1. Везде, кроме IE8-, достаточно вызвать `eval` не напрямую, а через `window.eval`.

Вот так:

```
var a = 1;

(function() {
    var a = 2;
    window.eval(' alert(a) '); // 1, выполнено глобально везде, кроме IE8-
})();
```

2. В IE8- можно применить нестандартную функцию [execScript](#). Она, как и `eval`, выполняет код, но всегда в глобальной области видимости и не возвращает значение.

Оба способа можно объединить в единой функции `globalEval(code)`, выполняющей код без доступа к локальным переменным:

```
function globalEval(code) { // объединим два способа в одну функцию
    window.execScript ? execScript(code) : window.eval(code);
}

var a = 1;

(function() {
    var a = 2;
    globalEval(' alert(a) '); // 1, во всех браузерах
})();
```

Внешние данные через new Function

Итак, у нас есть код, который, всё же, нужно выполнить динамически, через `eval`, но не просто скрипт – а ему нужно передать какие-то значения.

Как мы говорили ранее, считать их из локальных переменных нельзя: это подвержено ошибкам при переименовании переменных и сразу ломается при сжатии JavaScript. Да и вообще, неочевидно и криво.

К счастью, существует отличная альтернатива `eval`, которая позволяет корректно взаимодействовать с внешним кодом: `new Function`.

Вызов `new Function('a,b', '..тело..')` создает функцию с указанными аргументами `a,b` и телом. Как мы помним, доступа к текущему замыканию у такой функции не будет, но можно передать параметры и получить результат.

Например:

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

// вместо обращения к a,b через eval
// будем принимать их как аргументы динамически созданной функции
var mul = new Function('a, b', ' return a * b;');

alert( mul(a, b) ); // 6
```

JSON и eval

В браузерах IE7- не было методов `JSON.stringify` и `JSON.parse`, поэтому работа с JSON происходила через `eval`.

Этот способ работы с JSON давно устарел, но его можно встретить кое-где в старом коде, так что для примера рассмотрим его.

Вызов `eval(code)` выполняет код и, если это выражение, то возвращает его значение, поэтому можно в качестве кода передать JSON.

Например:

```
var str = '{ \
  "name": "Вася", \
  "age": 25 \
}';

var user = eval('(' + str + ')');

alert( user.name ); // Вася
```

Зачем здесь нужны скобки `eval('(' + str + ')')`, почему не просто `eval(str)`?

...Всё дело в том, что в JavaScript с фигурной скобки `{` начинаются не только объекты, а в том числе и «блоки кода». Что имеется в виду в данном случае – интерпретатор определяет по контексту. Если в основном потоке кода – то блок, если в контексте выражения, то объект.

Поэтому если передать в `eval` объект напрямую, то интерпретатор подумает, что это на самом деле блок кода, а там внутри какие-то двоеточия...

Вот, для примера, `eval` без скобок, он выдаст ошибку:

```
var user = eval({ "name": "Вася", "age": 25 });
```

А если `eval` получает выражение в скобках (...), то интерпретатор точно знает, что это не блок кода, а объект:

```
var user = eval('( { "name": "Вася", "age": 25 } )');
alert( user.age ); // 25
```

⚠ Осторожно, злой JSON!

Если мы получаем JSON из недоверенного источника, например с чужого сервера, то разбор через `eval` может быть опасен.

Например, чужой сервер может быть взломан (за свой-то код мы отвечаем, а за чужой – нет) и вместо JSON вставлен злонамеренный JavaScript-код.

Поэтому рекомендуется, всё же, использовать `JSON.parse`.

При разборе через `JSON.parse` некорректный JSON просто приведёт к ошибке, а вот при разборе через `eval` этот код реально выполнится, он может вывести что-то на странице, перенаправить посетителя куда-то и т.п.

Итого

- Функция `eval(str)` выполняет код и возвращает последнее вычисленное выражение. В современном JavaScript она используется редко.
- Вызов `eval` может читать и менять локальные переменные. Это – зло, которого нужно избегать.
- Для выполнения скрипта в глобальной области используются трюк с `window.eval/execScript`. При этом локальные переменные не будут затронуты, так что такое выполнение безопасно и иногда, в редких архитектурах, может быть полезным.
- Если выполняемый код всё же должен взаимодействовать с локальными переменными – используйте `new Function`. Создавайте функцию из строки и передавайте переменные ей, это надёжно и безопасно.

Ещё примеры использования `eval` вы найдёте далее, в главе [Формат JSON, метод toJSON](#).

✔ Задачи

Eval-калькулятор

важность: 4

Напишите интерфейс, который принимает математическое выражение (`prompt`) и возвращает его результат.

Проверять выражение на корректность не требуется.

[Запустить демо](#)

[К решению](#)

Перехват ошибок, "try..catch"

Как бы мы хорошо ни программировали, в коде бывают ошибки. Или, как их иначе называют, «исключительные ситуации» (исключения).

Обычно скрипт при ошибке, как говорят, «падает», с выводом ошибки в консоль.

Но бывают случаи, когда нам хотелось бы как-то контролировать ситуацию, чтобы скрипт не просто «упал», а сделал что-то разумное.

Для этого в JavaScript есть замечательная конструкция `try..catch`.

Конструкция try...catch

Конструкция `try..catch` состоит из двух основных блоков: `try`, и затем `catch`:

```
try {
  // код ...
} catch (err) {
  // обработка ошибки
}
```

Работает она так:

1. Выполняется код внутри блока `try`.
2. Если в нём ошибок нет, то блок `catch(err)` игнорируется, то есть выполнение доходит до конца `try` и потом прыгает через `catch`.
3. Если в нём возникнет ошибка, то выполнение `try` на ней прерывается, и управление прыгает в начало блока `catch(err)`.

При этом переменная `err` (можно выбрать и другое название) будет содержать объект ошибки с подробной информацией о произошедшем.

Таким образом, при ошибке в `try` скрипт не «падает», и мы получаем возможность обработать ошибку внутри `catch`.

Посмотрим это на примерах.

- Пример без ошибок: при запуске сработают `alert` (1) и (2) :

```
try {
  alert('Начало блока try'); // (1) <--
  // .. код без ошибок
  alert('Конец блока try'); // (2) <--
} catch(e) {
  alert('Блок catch не получит управление, так как нет ошибок'); // (3)
}
alert("Потом код продолжит выполнение...");
```

- Пример с ошибкой: при запуске сработают (1) и (3) :

```
try {
  alert('Начало блока try'); // (1) <--
  lalala; // ошибка, переменная не определена!
  alert('Конец блока try'); // (2)
} catch(e) {
  alert('Ошибка ' + e.name + ":" + e.message + "\n" + e.stack); // (3) <--
}
alert("Потом код продолжит выполнение...");
```

⚠ **try..catch** подразумевает, что код синтаксически верен

Если грубо нарушена структура кода, например не закрыта фигурная скобка или где-то стоит лишняя запятая, то никакой `try..catch` здесь не поможет. Такие ошибки называются *синтаксическими*, интерпретатор не может понять такой код.

Здесь же мы рассматриваем ошибки *семантические*, то есть происходящие в корректном коде, в процессе выполнения.

⚠ **try..catch** работает только в синхронном коде

Ошибку, которая произойдет в коде, запланированном «на будущее», например, в `setTimeout`, `try..catch` не поймает:

```
try {
  setTimeout(function() {
    throw new Error(); // вылетит в консоль
  }, 1000);
} catch (e) {
  alert( "не работает" );
}
```

На момент запуска функции, назначенной через `setTimeout`, этот код уже завершится, интерпретатор выйдет из блока `try..catch`.

Чтобы поймать ошибку внутри функции из `setTimeout`, и `try..catch` должен быть в той же функции.

Объект ошибки

В примере выше мы видим объект ошибки. У него есть три основных свойства:

name

Тип ошибки. Например, при обращении к несуществующей переменной: `"ReferenceError"`.

message

Текстовое сообщение о деталях ошибки.

stack

Везде, кроме IE8-, есть также свойство `stack`, которое содержит строку с информацией о последовательности вызовов, которая привела к ошибке.

В зависимости от браузера, у него могут быть и дополнительные свойства, см. [Error в MDN](#) и [Error в MSDN](#).

Пример использования

В JavaScript есть встроенный метод `JSON.parse(str)`, который используется для чтения JavaScript-объектов (и не только) из строки.

Обычно он используется для того, чтобы обрабатывать данные, полученные по сети, с сервера или из другого источника.

Мы получаем их и вызываем метод `JSON.parse`, вот так:

```
var data = '{"name":"Вася", "age": 30}'; // строка с данными, полученная с сервера
var user = JSON.parse(data); // преобразовали строку в объект

// теперь user -- это JS-объект с данными из строки
alert( user.name ); // Вася
alert( user.age ); // 30
```

Более детально формат JSON разобран в главе [Формат JSON, метод toJSON](#).

В случае, если данные некорректны, `JSON.parse` генерирует ошибку, то есть скрипт «упадёт».

Устроит ли нас такое поведение? Конечно нет!

Получается, что если вдруг что-то не так с данными, то посетитель никогда (если, конечно, не откроет консоль) об этом не узнает.

А люди очень-очень не любят, когда что-то «просто падает», без всякого объявления об ошибке.

Бывают ситуации, когда без `try..catch` не обойтись, это – одна из таких.

Используем `try..catch`, чтобы обработать некорректный ответ:

```
var data = "Has Error"; // в данных ошибка

try {

    var user = JSON.parse(data); // <-- ошибка при выполнении
    alert( user.name ); // не сработает

} catch (e) {
    // ...выполнится catch
    alert( "Извините, в данных ошибка, мы попробуем получить их ещё раз" );
    alert( e.name );
    alert( e.message );
}
```

Здесь в `alert` только выводится сообщение, но область применения гораздо шире: можно повторять запрос, можно предлагать посетителю использовать альтернативный способ, можно отсылать информацию об ошибке на сервер... Свобода действий.

Генерация своих ошибок

Представим на минуту, что данные являются корректным JSON... Но в этом объекте нет нужного свойства `name`:

```
var data = '{ "age": 30 }'; // данные неполны

try {

    var user = JSON.parse(data); // <-- выполнится без ошибок
    alert( user.name ); // undefined

} catch (e) {
    // не выполнится
    alert( "Извините, в данных ошибка" );
}
```

Вызов `JSON.parse` выполнится без ошибок, но ошибка в данных есть. И, так как свойство `name` обязательно должно быть, то для нас это такие же некорректные данные как и "Has Error".

Для того, чтобы унифицировать и объединить обработку ошибок парсинга и ошибок в структуре, мы воспользуемся оператором `throw`.

Оператор throw

Оператор `throw` генерирует ошибку.

Синтаксис: `throw <объект ошибки>`.

Технически, в качестве объекта ошибки можно передать что угодно, это может быть даже не объект, а число или строка, но всё же лучше, чтобы это был объект, желательно – совместимый со стандартным, то есть чтобы у него были как минимум свойства `name` и `message`.

В качестве конструктора ошибок можно использовать встроенный конструктор: `new Error(message)` или любой другой.

В JavaScript встроен ряд конструкторов для стандартных ошибок: `SyntaxError`, `ReferenceError`, `RangeError` и некоторые другие. Можно использовать и их, но только чтобы не было путаницы.

В данном случае мы используем конструктор `new SyntaxError(message)`. Он создаёт ошибку того же типа, что и `JSON.parse`.

```
var data = '{ "age": 30 }'; // данные неполны

try {

    var user = JSON.parse(data); // <-- выполнится без ошибок

    if (!user.name) {
        throw new SyntaxError("Данные некорректны");
    }

    alert( user.name );
}
```

```
} catch (e) {
  alert( "Извините, в данных ошибка" );
}
```

Получилось, что блок `catch` – единое место для обработки ошибок во всех случаях: когда ошибка выявляется при `JSON.parse` или позже.

Проброс исключения

В коде выше мы предусмотрели обработку ошибок, которые возникают при некорректных данных. Но может ли быть так, что возникнет какая-то другая ошибка?

Конечно, может! Код – это вообще мешок с ошибками, бывает даже так что библиотеку выкладывают в открытый доступ, она там 10 лет лежит, её смотрят миллионы людей и на 11-й год находятся опаснейшие ошибки. Такова жизнь, таковы люди.

Блок `catch` в нашем примере предназначен для обработки ошибок, возникающих при некорректных данных. Если же в него попала какая-то другая ошибка, то вывод сообщения о «некорректных данных» будет дезинформацией посетителя.

Ошибку, о которой `catch` не знает, он не должен обрабатывать.

Такая техника называется «проброс исключения»: в `catch(e)` мы анализируем объект ошибки, и если он нам не подходит, то делаем `throw e`.

При этом ошибка «выпадает» из `try..catch` наружу. Далее она может быть поймана либо внешним блоком `try..catch` (если есть), либо «повалит» скрипт.

В примере ниже `catch` обрабатывает только ошибки `SyntaxError`, а остальные – выбрасывает дальше:

```
var data = '{ "name": "Вася", "age": 30 }'; // данные корректны

try {
  var user = JSON.parse(data);

  if (!user.name) {
    throw new SyntaxError("Ошибка в данных");
  }

  blabla(); // произошла непредусмотренная ошибка

  alert( user.name );
} catch (e) {

  if (e.name == "SyntaxError") {
    alert( "Извините, в данных ошибка" );
  } else {
    throw e;
  }

}
```

Заметим, что ошибка, которая возникла внутри блока `catch`, «выпадает» наружу, как если бы была в обычном коде.

В следующем примере такие ошибки обрабатываются ещё одним, «более внешним» `try..catch`:

```
function readData() {
  var data = '{ "name": "Вася", "age": 30 }';

  try {
    // ...
    blabla(); // ошибка!
  } catch (e) {
    // ...
    if (e.name != 'SyntaxError') {
      throw e; // пробрасываем
    }
  }
}

try {
  readData();
} catch (e) {
  alert( "Поймал во внешнем catch: " + e ); // ловим
}
```

В примере выше `try..catch` внутри `readData` умеет обрабатывать только `SyntaxError`, а внешний – все ошибки.

Без внешнего проброшенная ошибка «вывалилась» бы в консоль, с остановкой скрипта.

Оборачивание исключений

И, для полноты картины – последняя, самая продвинутая техника по работе с ошибками. Она, впрочем, является стандартной практикой во многих объектно-ориентированных языках.

Цель функции `readData` в примере выше – прочитать данные. При чтении могут возникать разные ошибки, не только `SyntaxError`, но и, возможно, к примеру, `URIError` (неправильное применение функций работы с URI), да и другие.

Код, который вызвал `readData`, хотел бы иметь либо результат, либо информацию об ошибке.

При этом очень важным является вопрос: обязан ли этот внешний код знать о всевозможных типах ошибок, которые могут возникать при чтении данных, и уметь перехватывать их?

Обычно внешний код хотел бы работать «на уровень выше», и получать либо результат, либо «ошибку чтения данных», при этом какая именно ошибка произошла – ему неважно. Ну, или, если будет важно, то хотелось бы иметь возможность это узнать, но обычно не требуется.

Это важнейший общий подход к проектированию – каждый участок функционала должен получать информацию на том уровне, который ему необходим.

Мы его видим везде в грамотно построенном коде, но не всегда отдаём себе в этом отчёт.

В данном случае, если при чтении данных происходит ошибка, то мы будем генерировать её в виде объекта `ReadError`, с соответствующим сообщением. А «исходную» ошибку – на всякий случай тоже сохраним, присвоим в свойство `cause` (англ. – причина).

Выглядит это так:

```
function ReadError(message, cause) {
  this.message = message;
  this.cause = cause;
  this.name = 'ReadError';
  this.stack = cause.stack;
}

function readData() {
  var data = '{ bad data }';

  try {
    // ...
    JSON.parse(data);
    // ...
  } catch (e) {
    // ...
    if (e.name == 'URIError') {
      throw new ReadError("Ошибка в URI", e);
    } else if (e.name == 'SyntaxError') {
      throw new ReadError("Синтаксическая ошибка в данных", e);
    } else {
      throw e; // пробрасываем
    }
  }
}

try {
  readData();
} catch (e) {
  if (e.name == 'ReadError') {
    alert( e.message );
    alert( e.cause ); // оригинальная ошибка-причина
  } else {
    throw e;
  }
}
```

Этот подход называют «оборачиванием» исключения, поскольку мы берём ошибки «более низкого уровня» и «заворачиваем» их в `ReadError`, которая соответствует текущей задаче.

Секция finally

Конструкция `try..catch` может содержать ещё один блок: `finally`.

Выглядит этот расширенный синтаксис так:

```
try {
  .. пробуем выполнить код ..
} catch(e) {
  .. перехватываем исключение ..
} finally {
  .. выполняем всегда ..
}
```

Секция `finally` не обязательна, но если она есть, то она выполняется всегда:

- после блока `try`, если ошибок не было,
- после `catch`, если они были.

Попробуйте запустить такой код?

```
try {
  alert( 'try' );
  if (confirm('Сгенерировать ошибку?')) BAD_CODE();
} catch (e) {
  alert( 'catch' );
} finally {
  alert( 'finally' );
}
```

У него два варианта работы:

1. Если вы ответите на вопрос «Сгенерировать ошибку?» утвердительно, то `try -> catch -> finally`.
2. Если ответите отрицательно, то `try -> finally`.

Секцию `finally` используют, чтобы завершить начатые операции при любом варианте развития событий.

Например, мы хотим подсчитать время на выполнение функции `sum(n)`, которая должна вернуть сумму чисел от 1 до `n` и работает рекурсивно:

```
function sum(n) {
  return n ? (n + sum(n - 1)) : 0;
}

var n = +prompt('Введите n?', 100);

var start = new Date();

try {
  var result = sum(n);
} catch (e) {
  result = 0;
} finally {
  var diff = new Date() - start;
}

alert( result ? result : 'была ошибка' );
alert( "Выполнение заняло " + diff );
```

Здесь секция `finally` гарантирует, что время будет подсчитано в любых ситуациях – при ошибке в `sum` или без неё.

Вы можете проверить это, запустив код с указанием `n=100` – будет без ошибки, `finally` выполнится после `try`, а затем с `n=100000` – будет ошибка из-за слишком глубокой рекурсии, управление прыгнет в `finally` после `catch`.

i finally и return

Блок `finally` срабатывает при *любом* выходе из `try..catch`, в том числе и `return`.

В примере ниже, из `try` происходит `return`, но `finally` получает управление до того, как контроль возвращается во внешний код.

```
function func() {

  try {
    // сразу вернуть значение
    return 1;
  } catch (e) {
    /* ... */
  } finally {
    alert( 'finally' );
  }
}

alert( func() ); // сначала finally, потом 1
```

Если внутри `try` были начаты какие-то процессы, которые нужно завершить по окончании работы, во в `finally` это обязательно будет сделано.

Кстати, для таких случаев иногда используют `try..finally` вообще без `catch`:

```
function func() {
  try {
    return 1;
  } finally {
    alert( 'Вызов завершён' );
  }
}

alert( func() ); // сначала finally, потом 1
```

В примере выше `try..finally` вообще не обрабатывает ошибки. Задача в другом – выполнить код при любом выходе из `try` – с ошибкой ли, без ошибок или через `return`.

Последняя надежда: `window.onerror`

Допустим, ошибка произошла вне блока `try..catch` или выпала из `try..catch` наружу, во внешний код. Скрипт упал.

Можно ли как-то узнать о том, что произошло? Да, конечно.

В браузере существует специальное свойство `window.onerror`, если в него записать функцию, то она выполнится и получит в аргументах сообщение ошибки, текущий URL и номер строки, откуда «выпала» ошибка.

Необходимо лишь позаботиться, чтобы функция была назначена заранее.

Например:

```
<script>
window.onerror = function(message, url, lineNumber) {
  alert("Поймана ошибка, выпавшая в глобальную область!\n" +
    "Сообщение: " + message + "\n(" + url + ":" + lineNumber + ")");
};

function readData() {
  error(); // ой, что-то не так
}

readData();
</script>
```

Как правило, роль `window.onerror` заключается в том, чтобы не оживить скрипт – скорее всего, это уже невозможно, а в том, чтобы отослать сообщение об ошибке на сервер, где разработчики о ней узнают.

Существуют даже специальные веб-сервисы, которые предоставляют скрипты для отлова и аналитики таких ошибок, например: <https://errorception.com/> или <http://www.muscula.com/>.

Итого

Обработка ошибок – большая и важная тема.

В JavaScript для этого предусмотрены:

- Конструкция `try..catch..finally` – она позволяет обработать произвольные ошибки в блоке кода.

Это удобно в тех случаях, когда проще сделать действие и потом разбираться с результатом, чем долго и нудно проверять, не упадёт ли чего.

Кроме того, иногда проверить просто невозможно, например `JSON.parse(str)` не позволяет «проверить» формат строки перед разбором. В этом случае блок `try..catch` необходим.

Полный вид конструкции:

```
try {
  .. пробуем выполнить код ..
} catch(e) {
  .. перехватываем исключение ..
} finally {
  .. выполняем всегда ..
}
```

Возможны также варианты `try..catch` или `try..finally`.

- Оператор `throw err` генерирует свою ошибку, в качестве `err` рекомендуется использовать объекты, совместимые с встроенным типом `Error`, содержащие свойства `message` и `name`.

Кроме того, мы рассмотрели некоторые важные приёмы:

- Проброс исключения – `catch(err)` должен обрабатывать только те ошибки, которые мы рассчитываем в нём увидеть, остальные – пробрасывать дальше через `throw err`.

Определить, нужна ли это ошибка, можно, например, по свойству `name`.

- Оборачивание исключений – функция, в процессе работы которой возможны различные виды ошибок, может «обернуть их» в одну общую ошибку, специфичную для её задачи, и уже её пробросить дальше. Чтобы, при необходимости, можно было подробно определить, что произошло, исходную ошибку обычно присваивают в свойство этой, общей. Обычно это нужно для логирования.
- В `window.onerror` можно присвоить функцию, которая выполнится при любой «выпавшей» из скрипта ошибке. Как правило, это используют в информационных целях, например отправляют информацию об ошибке на специальный сервис.

✓ Задачи

Finally или просто код?

важность: 5

Сравните два фрагмента кода.

1. Первый использует `finally` для выполнения кода по выходу из `try..catch`:

```
try {
  начать работу
  работать
} catch (e) {
  обработать ошибку
} finally {
  финализация: завершить работу
}
```

2. Второй фрагмент просто ставит очистку ресурсов за `try..catch`:

```
try {
  начать работу
} catch (e) {
  обработать ошибку
}

финализация: завершить работу
```

Нужно, чтобы код финализации всегда выполнялся при выходе из блока `try..catch` и, таким образом, заканчивал начатую работу. Имеет ли здесь `finally` какое-то преимущество или оба фрагмента работают одинаково?

Если имеет, то дайте пример когда код с `finally` работает верно, а без – неверно.

Eval-калькулятор с ошибками

важность: 5

Напишите интерфейс, который принимает математическое выражение (в `prompt`) и результат его вычисления через `eval`.

При ошибке нужно выводить сообщение и просить перевести выражение.

Ошибкой считается не только некорректное выражение, такое как `2+`, но и выражение, возвращающее `NaN`, например `0/0`.

[Запустить демо](#)

[К решению](#)

ООП в функциональном стиле

Инкапсуляция и наследование в функциональном стиле, а также расширенные возможности объектов JavaScript.

Введение

На протяжении долгого времени в программировании применялся [процедурный подход](#). При этом программа состоит из функций, вызывающих друг друга.

Гораздо позже появилось [объектно-ориентированное программирование](#) (ООП), которое позволяет группировать функции и данные в единой сущности – «объекте».

При объектно-ориентированной разработке мы описываем происходящее на уровне объектов, которые создаются, меняют свои свойства, взаимодействуют друг с другом и (в случае браузера) со страницей, в общем, живут.

Например, «пользователь», «меню», «компонент интерфейса»... При объектно-ориентированном подходе каждый объект должен представлять собой интуитивно понятную сущность, у которой есть методы и данные.

ООП – это не просто объекты

В JavaScript объекты часто используются просто как коллекции.

Например, встроенный объект [Math](#) содержит функции (`Math.sin`, `Math.pow`, ...) и данные (константа `Math.PI`).

При таком использовании объектов мы не можем сказать, что «применён объектно-ориентированный подход». В частности, никакую «единую сущность» `Math` из себя не представляет, это просто коллекция независимых функций с общим префиксом `Math`.

Мы уже работали в ООП-стиле, создавая объекты такого вида:

```
function User(name) {
  this.sayHi = function() {
    alert( "Привет, я " + name );
  };
}

var vasya = new User("Вася"); // создали пользователя
vasya.sayHi(); // пользователь умеет говорить "Привет"
```

Здесь мы видим ярко выраженную сущность – `User` (посетитель). Используя терминологию ООП, такие конструкторы часто называют *классами*, то есть можно сказать "класс `User`".

Класс в ООП

[Классом](#) в объектно-ориентированной разработке называют шаблон/программный код, предназначенный для создания объектов и методов.

В JavaScript классы можно организовать по-разному. Говорят, что класс `User` написан в «функциональном» стиле. Далее мы также увидим «прототипный» стиль.

ООП – это наука о том, как делать правильную архитектуру. У неё есть свои принципы, например [SOLID](#).

По приёмам объектно-ориентированной разработки пишут книги, к примеру:

- [Объектно-ориентированный анализ и проектирование с примерами приложений](#). [Э. Гради Буч и др.](#)
- [Приёмы объектно-ориентированного проектирования. Паттерны проектирования](#). [Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес.](#)

Здесь мы не имеем возможности углубиться в теорию ООП, поэтому чтение таких книг рекомендуется. Хотя основные принципы, как использовать ООП правильно, мы, всё же, затронем.

Внутренний и внешний интерфейс

Один из важнейших принципов ООП – отделение внутреннего интерфейса от внешнего.

Это – обязательная практика в разработке чего угодно сложнее hello world.

Чтобы это понять, отвлечемся от разработки и переведем взгляд на объекты реального мира.

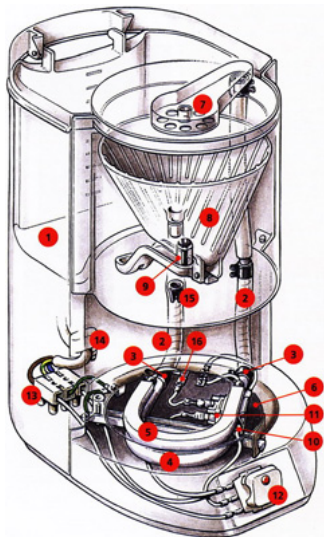
Как правило, устройства, с которыми мы имеем дело, весьма сложны. Но *разделение интерфейса на внешний и внутренний* позволяет использовать их без малейших проблем.

Пример из жизни

Например, кофеварка. Простая снаружи: кнопка, индикатор, отверстия,... И, конечно, результат – кофе :)



Но внутри... (картинка из пособия по ремонту)



Масса деталей. Но мы можем пользоваться ей, совершенно не зная об этом.

Кофеварки – довольно-таки надежны, не правда ли? Можно пользоваться годами, и только когда что-то пойдет не так – придется нести к мастеру.

Секрет надежности и простоты кофеварки – в том, что все детали отлажены и *спрятаны* внутри.

Если снять с кофеварки защитный кожух, то использование её будет более сложным (куда нажимать?) и опасным (током ударить может).

Как мы увидим, объекты очень схожи с кофеварками.

Только для того, чтобы прятать внутренние детали, используется не кожух, а специальные средства языка и соглашения.

Внутренний и внешний интерфейс

В программировании мы будем разделять методы и свойства объекта на две группы:

- *Внутренний интерфейс* – это свойства и методы, доступ к которым может быть осуществлен только из других методов объекта, их также называют «*приватными*» (есть и другие термины, встретим их далее).
- *Внешний интерфейс* – это свойства и методы, доступные снаружи объекта, их называют «*публичными*».

Если продолжить аналогию с кофеваркой – то, что спрятано внутри кофеварки: трубка кипятильника, нагревательный элемент, тепловой предохранитель и так далее – это её *внутренний интерфейс*.

Внутренний интерфейс используется для обеспечения работоспособности объекта, его детали используют друг друга. Например, трубка кипятильника подключена к нагревательному элементу.

Но снаружи кофеварка закрыта специальным кожухом, чтобы никто к ним не подобрался. Детали скрыты и недоступны. Виден лишь внешний интерфейс.

Получив объект, всё, что нужно для пользования им – это знать внешний интерфейс. О внутреннем же знать вообще не обязательно.

Это были общие слова по теории программирования.

Далее мы реализуем кофеварку на JavaScript с приватными и публичными свойствами. В кофеварке много деталей, мы конечно, не будем моделировать каждый винтик, а сосредоточимся на основных приёмах разработки.

Шаг 1: публичное и приватное свойство

Конструктор кофеварок будет называться `CoffeeMachine`.

```
function CoffeeMachine(power) {
  this.waterAmount = 0; // количество воды в кофеварке

  alert( 'Создана кофеварка мощностью: ' + power + ' ватт' );
}

// создать кофеварку
var coffeeMachine = new CoffeeMachine(100);

// залить воды
coffeeMachine.waterAmount = 200;
```

Локальные переменные, включая параметры конструктора, можно считать приватными свойствами.

В примере выше это `power` – мощность кофеварки, которая указывается при создании и далее будет использована для расчёта времени кипячения.

К локальным переменным конструктора нельзя обратиться снаружи, но они доступны внутри самого конструктора.

Свойства, записанные в `this`, можно считать публичными.

Здесь свойство `waterAmount` записано в объект, а значит – доступно для модификации снаружи. Можно доливать и выливать воду в любом количестве.

Вопрос терминологии

Далее мы будем называть `power` как «локальной переменной», так и «приватным свойством» объекта.

Это, смотря, с какой стороны посмотреть.

Термины «приватное свойство/метод», «публичное свойство/метод» относятся к общей теории ООП. А их конкретная реализация в языке программирования может быть различной.

Здесь ООП-принцип «приватного свойства» реализован через локальные переменные, поэтому и «локальная переменная» и «приватное свойство» – правильные термины, в зависимости от того, с какой точки зрения взглянуть – кода или архитектуры ООП.

Шаг 2: публичный и приватный методы

Добавим публичный метод `run`, запускающий кофеварку, а также вспомогательные внутренние методы `getBoilTime` и `onReady`:

```
function CoffeeMachine(power) {
  this.waterAmount = 0;

  // расчёт времени для кипячения
  function getBoilTime() {
    return 1000; // точная формула расчета будет позже
  }

  // что делать по окончании процесса
  function onReady() {
    alert( 'Кофе готово!' );
  }

  this.run = function() {
    // setTimeout - встроенная функция,
    // она запустит onReady через getBoilTime() миллисекунд
    setTimeout(onReady, getBoilTime());
  };
}

var coffeeMachine = new CoffeeMachine(100);
coffeeMachine.waterAmount = 200;

coffeeMachine.run();
```

Приватные методы, такие как `onReady`, `getBoilTime` могут быть объявлены как вложенные функции.

В результате естественным образом получается, что доступ к ним (через замыкание) имеют только другие функции, объявленные в том же конструкторе.

Шаг 3: константа

Для расчёта времени на кипячение воды используется формула $c \cdot m \cdot \Delta T / \text{power}$, где:

- c – коэффициент теплоёмкости воды, физическая константа равная 4200.
- m – масса воды, которую нужно нагреть.
- ΔT – температура, на которую нужно подогреть, будем считать, что изначально вода – комнатной температуры 20°C, то есть до 100° нужно греть на $\Delta T=80$.
- power – мощность.

Используем её в более реалистичном варианте `getBoilTime()`, включающем использование приватных свойств и константу:

```
"use strict"

function CoffeeMachine(power) {
  this.waterAmount = 0;

  // физическая константа - удельная теплоёмкость воды для getBoilTime
  var WATER_HEAT_CAPACITY = 4200;

  // расчёт времени для кипячения
  function getBoilTime() {
    return this.waterAmount * WATER_HEAT_CAPACITY * 80 / power; // ошибка!
  }

  // что делать по окончании процесса
  function onReady() {
    alert( 'Кофе готово!' );
  }

  this.run = function() {
    setTimeout(onReady, getBoilTime());
  };
}

var coffeeMachine = new CoffeeMachine(1000);
coffeeMachine.waterAmount = 200;

coffeeMachine.run();
```

Удельная теплоёмкость `WATER_HEAT_CAPACITY` выделена большими буквами, так как это константа.

Внимание, при запуске кода выше в методе `getBoilTime` будет ошибка. Как вы думаете, почему?

Шаг 4: доступ к объекту из внутреннего метода

Внутренний метод вызывается так: `getBoilTime()`. А чему при этом равен `this`?... Как вы наверняка помните, в современном стандарте он будет `undefined` (в старом – `window`), из-за этого при чтении `this.waterAmount` возникнет ошибка!

Её можно решить, если вызвать `getBoilTime` с явным указанием контекста: `getBoilTime.call(this)`:

```
function CoffeeMachine(power) {
  this.waterAmount = 0;
  var WATER_HEAT_CAPACITY = 4200;

  function getBoilTime() {
    return this.waterAmount * WATER_HEAT_CAPACITY * 80 / power;
  }

  function onReady() {
    alert( 'Кофе готово!' );
  }

  this.run = function() {
    setTimeout(onReady, getBoilTime.call(this));
  };
}

// создаю кофеварку, мощностью 100000W чтобы кипятила быстро
var coffeeMachine = new CoffeeMachine(100000);
coffeeMachine.waterAmount = 200;

coffeeMachine.run();
```

Такой подход будет работать, но он не очень-то удобен. Ведь получается, что теперь везде, где мы хотим вызвать `getBoilTime`, нужно явно указывать контекст, т.е. писать `getBoilTime.call(this)`.

К счастью существуют более элегантные решения.

Привязка через bind

Можно при объявлении привязать `getBoilTime` к объекту через `bind`, тогда вопрос контекста отпадёт сам собой:

```
function CoffeeMachine(power) {
  this.waterAmount = 0;

  var WATER_HEAT_CAPACITY = 4200;

  var getBoilTime = function() {
    return this.waterAmount * WATER_HEAT_CAPACITY * 80 / power;
  }.bind(this);
}
```

```

function onReady() {
  alert( 'Кофе готово!' );
}

this.run = function() {
  setTimeout(onReady, getBoilTime());
};

}

var coffeeMachine = new CoffeeMachine(100000);
coffeeMachine.waterAmount = 200;

coffeeMachine.run();

```

Это решение будет работать, теперь функцию можно просто вызывать без `call`. Но объявление функции стало менее красивым.

Сохранение `this` в замыкании

Пожалуй, самый удобный и часто применяемый путь решения состоит в том, чтобы предварительно скопировать `this` во вспомогательную переменную и обращаться из внутренних функций уже к ней.

Вот так:

```

function CoffeeMachine(power) {
  this.waterAmount = 0;

  var WATER_HEAT_CAPACITY = 4200;

  var self = this;

  function getBoilTime() {
    return self.waterAmount * WATER_HEAT_CAPACITY * 80 / power;
  }

  function onReady() {
    alert( 'Кофе готово!' );
  }

  this.run = function() {
    setTimeout(onReady, getBoilTime());
  };
}

var coffeeMachine = new CoffeeMachine(100000);
coffeeMachine.waterAmount = 200;

coffeeMachine.run();

```

Теперь `getBoilTime` получает `self` из замыкания.

Конечно, чтобы это работало, мы не должны изменять `self`, а все приватные методы, которые хотят иметь доступ к текущему объекту, должны использовать внутри себя `self` вместо `this`.

Вместо `self` можно использовать любое другое имя переменной, например `var me = this`.

Итого

Итак, мы сделали кофеварку с публичными и приватными методами и заставили их корректно работать.

В терминологии ООП отделение и защита внутреннего интерфейса называется [инкапсуляция](#).

Кратко перечислим бонусы, которые она даёт:

Защита пользователей от выстрела себе в ногу

Представьте, команда разработчиков пользуется кофеваркой. Кофеварка создана фирмой «Лучшие Кофеварки» и, в общем, работает хорошо, но с неё сняли защитный кожух и, таким образом, внутренний интерфейс стал доступен.

Все разработчики цивилизованны – и пользуются кофеваркой как обычно. Но хитрый Вася решил, что он самый умный, и подкрутил кое-что внутри кофеварки, чтобы кофе заваривался покрепче. Вася не знал, что те изменения, которые он произвёл, приведут к тому, что кофеварка испортится через два дня.

Виноват, разумеется, не только Вася, но и тот, кто снял защитный кожух с кофеварки, и тем самым позволил Васе проводить манипуляции.

В программировании – то же самое. Если пользователь объекта будет менять то, что не рассчитано на изменение снаружи – последствия могут быть непредсказуемыми.

Удобство в поддержке

Ситуация в программировании сложнее, чем с кофеваркой, т.к. кофеварку один раз купили и всё, а программа может улучшаться и дорабатываться.

При наличии чётко выделенного внешнего интерфейса, разработчик может свободно менять внутренние свойства и методы, без оглядки на коллег.

Гораздо легче разрабатывать, если знаешь, что ряд методов (все внутренние) можно переименовывать, менять их параметры, и вообще, переписать как угодно, так как внешний код к ним абсолютно точно не обращается.

Ближайшая аналогия в реальной жизни – это когда выходит «новая версия» кофеварки, которая работает гораздо лучше. Разработчик мог переделать всё внутри, но пользоваться ей по-прежнему просто, так как внешний интерфейс сохранён.

Управление сложностью

Люди обожают пользоваться вещами, которые просты с виду. А что внутри – дело десятое.

Программисты здесь не исключение.

Всегда удобно, когда детали реализации скрыты, и доступен простой, понятно документированный внешний интерфейс.

✔ Задачи

Добавить метод и свойство кофеварке

важность: 5

Улучшите готовый код кофеварки, который дан ниже: добавьте в кофеварку *публичный* метод `stop()`, который будет останавливать кипячение (через `clearTimeout`).

```
function CoffeeMachine(power) {
  this.waterAmount = 0;

  var WATER_HEAT_CAPACITY = 4200;

  var self = this;

  function getBoilTime() {
    return self.waterAmount * WATER_HEAT_CAPACITY * 80 / power;
  }

  function onReady() {
    alert( 'Кофе готово!' );
  }

  this.run = function() {
    setTimeout(onReady, getBoilTime());
  };
}
```

Вот такой код должен ничего не выводить:

```
var coffeeMachine = new CoffeeMachine(50000);
coffeeMachine.waterAmount = 200;

coffeeMachine.run();
coffeeMachine.stop(); // кофе приготовлен не будет
```

P.S. Текущую температуру воды вычислять и хранить не требуется.

P.P.S. При решении вам, скорее всего, понадобится добавить *приватное* свойство `timerId`, которое будет хранить текущий таймер.

[К решению](#)

Геттеры и сеттеры

Для *управляемого* доступа к состоянию объекта используют специальные функции, так называемые «геттеры» и «сеттеры».

Геттер и сеттер для воды

На текущий момент количество воды в кофеварке является публичным свойством `waterAmount`:

```
function CoffeeMachine(power) {
  // количество воды в кофеварке
  this.waterAmount = 0;

  ...
}
```

Это немного опасно. Ведь в это свойство можно записать произвольное количество воды, хоть весь мировой океан.

```
// не помещается в кофеварку!
coffeeMachine.waterAmount = 1000000;
```

Это ещё ничего, гораздо хуже, что можно наоборот – вылить больше, чем есть:

```
// и не волнует, было ли там столько воды вообще!
coffeeMachine.waterAmount -= 1000000;
```

Так происходит потому, что свойство полностью доступно снаружи.

Чтобы не было таких казусов, нам нужно ограничить контроль над свойством со стороны внешнего кода.

Для лучшего контроля над свойством его делают приватным, а запись значения осуществляется через специальный метод, который называют «сеттер» (setter method).

Типичное название для сеттера – setСвойство, например, в случае с кофеваркой таким сеттером будет метод setWaterAmount:

```
function CoffeeMachine(power, capacity) { // capacity - ёмкость кофеварки
  var waterAmount = 0;

  var WATER_HEAT_CAPACITY = 4200;

  function getTimeToBoil() {
    return waterAmount * WATER_HEAT_CAPACITY * 80 / power;
  }

  // "умная" установка свойства
  this.setWaterAmount = function(amount) {
    if (amount < 0) {
      throw new Error("Значение должно быть положительным");
    }
    if (amount > capacity) {
      throw new Error("Нельзя залить воды больше, чем " + capacity);
    }
    waterAmount = amount;
  };

  function onReady() {
    alert( 'Кофе готов!' );
  }

  this.run = function() {
    setTimeout(onReady, getTimeToBoil());
  };
}

var coffeeMachine = new CoffeeMachine(1000, 500);
coffeeMachine.setWaterAmount(600); // упс, ошибка!
```

Теперь waterAmount – внутреннее свойство, его можно записать (через сеттер), но, увы, нельзя прочитать.

Для того, чтобы дать возможность внешнему коду узнать его значение, создадим специальную функцию – «getter» (getter method).

Геттеры обычно имеют название вида getСвойство, в данном случае getWaterAmount:

```
function CoffeeMachine(power, capacity) {
  //...
  this.setWaterAmount = function(amount) {
    if (amount < 0) {
      throw new Error("Значение должно быть положительным");
    }
    if (amount > capacity) {
      throw new Error("Нельзя залить воды больше, чем " + capacity);
    }
    waterAmount = amount;
  };

  this.getWaterAmount = function() {
    return waterAmount;
  };
}

var coffeeMachine = new CoffeeMachine(1000, 500);
coffeeMachine.setWaterAmount(450);
alert( coffeeMachine.getWaterAmount() ); // 450
```

Единый геттер-сеттер

Для большего удобства иногда делают единый метод, который называется так же, как свойство и отвечает и за запись и за чтение.

При вызове без параметров такой метод возвращает свойство, а при передаче параметра – назначает его.

Выглядит это так:

```
function CoffeeMachine(power, capacity) {
  var waterAmount = 0;

  this.waterAmount = function(amount) {
    // вызов без параметра, значит режим геттера, возвращаем свойство
    if (!arguments.length) return waterAmount;

    // иначе режим сеттера
    if (amount < 0) {
      throw new Error("Значение должно быть положительным");
    }
    if (amount > capacity) {
      throw new Error("Нельзя залить воды больше, чем " + capacity);
    }
    waterAmount = amount;
  };
}

var coffeeMachine = new CoffeeMachine(1000, 500);

// пример использования
```

```
coffeeMachine.waterAmount(450);
alert( coffeeMachine.waterAmount() ); // 450
```

Единый геттер-сеттер используется реже, чем две отдельные функции, но в некоторых JavaScript-библиотеках, например [jQuery](#) и [D3](#) подобный подход принят на уровне концепта.

Итого

- Для большего контроля над присвоением и чтением значения, вместо свойства делают «функцию-геттер» и «функцию-сеттер», геттер возвращает значение, сеттер – устанавливает.
- Если свойство предназначено только для чтения, то может быть только геттер, только для записи – только сеттер.
- В качестве альтернативы паре геттер/сеттер применяют единую функцию, которая без аргументов ведёт себя как геттер, а с аргументом – как сеттер.

Также можно организовать геттеры/сеттеры для свойства, не меняя структуры кода, через [дескрипторы свойств](#).

✔ Задачи

Написать объект с геттерами и сеттерами

важность: 4

Напишите конструктор `User` для создания объектов:

- С приватными свойствами имя `firstName` и фамилия `surname`.
- С сеттерами для этих свойств.
- С геттером `getFullName()`, который возвращает полное имя.

Должен работать так:

```
function User() {
  /* ваш код */
}

var user = new User();
user.setFirstName("Петя");
user.setSurname("Иванов");

alert( user.getFullName() ); // Петя Иванов
```

[К решению](#)

Добавить геттер для power

важность: 5

Добавьте кофеварке геттер для приватного свойства `power`, чтобы внешний код мог узнать мощность кофеварки.

Исходный код:

```
function CoffeeMachine(power, capacity) {
  //...
  this.setWaterAmount = function(amount) {
    if (amount < 0) {
      throw new Error("Значение должно быть положительным");
    }
    if (amount > capacity) {
      throw new Error("Нельзя залить воды больше, чем " + capacity);
    }

    waterAmount = amount;
  };

  this.getWaterAmount = function() {
    return waterAmount;
  };
}
```

Обратим внимание, что ситуация, когда у свойства `power` есть геттер, но нет сеттера – вполне обычна.

Здесь это означает, что мощность `power` можно указать лишь при создании кофеварки и в дальнейшем её можно прочитать, но нельзя изменить.

[К решению](#)

Добавить публичный метод кофеварке

важность: 5

Добавьте кофеварке публичный метод `addWater(amount)`, который будет добавлять воду.

При этом, конечно же, должны происходить все необходимые проверки – на положительность и превышение ёмкости.

Исходный код:

```
function CoffeeMachine(power, capacity) {
  var waterAmount = 0;

  var WATER_HEAT_CAPACITY = 4200;

  function getTimeToBoil() {
    return waterAmount * WATER_HEAT_CAPACITY * 80 / power;
  }

  this.setWaterAmount = function(amount) {
    if (amount < 0) {
      throw new Error("Значение должно быть положительным");
    }
    if (amount > capacity) {
      throw new Error("Нельзя залить больше, чем " + capacity);
    }

    waterAmount = amount;
  };

  function onReady() {
    alert( 'Кофе готов!' );
  }

  this.run = function() {
    setTimeout(onReady, getTimeToBoil());
  };
}
```

Вот такой код должен приводить к ошибке:

```
var coffeeMachine = new CoffeeMachine(100000, 400);
coffeeMachine.addWater(200);
coffeeMachine.addWater(100);
coffeeMachine.addWater(300); // Нельзя залить больше, чем 400
coffeeMachine.run();
```

[К решению](#)

Создать сеттер для onReady

важность: 5

Обычно когда кофе готов, мы хотим что-то сделать, например выпить его.

Сейчас при готовности срабатывает функция `onReady`, но она жёстко задана в коде:

```
function CoffeeMachine(power, capacity) {
  var waterAmount = 0;

  var WATER_HEAT_CAPACITY = 4200;

  function getTimeToBoil() {
    return waterAmount * WATER_HEAT_CAPACITY * 80 / power;
  }

  this.setWaterAmount = function(amount) {
    // ... проверки пропущены для краткости
    waterAmount = amount;
  };

  this.getWaterAmount = function(amount) {
    return waterAmount;
  };

  function onReady() {
    alert( 'Кофе готов!' );
  }

  this.run = function() {
    setTimeout(onReady, getTimeToBoil());
  };
}
```

Создайте сеттер `setOnReady`, чтобы код снаружи мог назначить свой `onReady`, вот так:

```
var coffeeMachine = new CoffeeMachine(20000, 500);
coffeeMachine.setWaterAmount(150);

coffeeMachine.setOnReady(function() {
  var amount = coffeeMachine.getWaterAmount();
  alert( 'Готов кофе: ' + amount + 'мл' ); // Кофе готов: 150 мл
});

coffeeMachine.run();
```

P.S. Значение `onReady` по умолчанию должно быть таким же, как и раньше.

P.P.S. Постарайтесь сделать так, чтобы `setOnReady` можно было вызвать не только до, но и *после* запуска кофеварки, то есть чтобы функцию `onReady` можно было изменить в любой момент до её срабатывания.

[К решению](#)

Добавить метод `isRunning`

важность: 5

Из внешнего кода мы хотели бы иметь возможность понять – запущена кофеварка или нет.

Для этого добавьте кофеварке публичный метод `isRunning()`, который будет возвращать `true`, если она запущена и `false`, если нет.

Нужно, чтобы такой код работал:

```
var coffeeMachine = new CoffeeMachine(20000, 500);
coffeeMachine.setWaterAmount(100);

alert( 'До: ' + coffeeMachine.isRunning() ); // До: false

coffeeMachine.run();
alert( 'В процессе: ' + coffeeMachine.isRunning() ); // В процессе: true

coffeeMachine.setOnReady(function() {
  alert( "После: " + coffeeMachine.isRunning() ); // После: false
});
```

Исходный код возьмите из решения [предыдущей задачи](#).

[К решению](#)

Функциональное наследование

Наследование – это создание новых «классов» на основе существующих.

В JavaScript его можно реализовать несколькими путями, один из которых – с использованием наложения конструкторов, мы рассмотрим в этой главе.

Зачем наследование?

Ранее мы обсуждали различные реализации кофеварки. Продолжим эту тему далее.

Хватит ли нам только кофеварки для удобной жизни? Вряд ли... Скорее всего, ещё понадобятся как минимум холодильник, микроволновка, а возможно и другие *машины*.

В реальной жизни у этих *машин* есть базовые правила пользования. Например, большая кнопка – включение, шнур с розеткой нужно воткнуть в питание и т.п.

Можно сказать, что «у всех машин есть общие свойства, а конкретные машины могут их дополнять».

Именно поэтому, увидев новую технику, мы уже можем что-то с ней сделать, даже не читая инструкцию.

Механизм наследования позволяет определить базовый класс *Машина*, в нём описать то, что свойственно всем машинам, а затем на его основе построить другие, более конкретные: *Кофеварка*, *Холодильник* и т.п.



В веб-разработке всё так же

В веб-разработке нам могут понадобиться классы *Меню*, *Табы*, *Диалог* и другие компоненты интерфейса. В них всех обычно есть что-то общее.

Можно выделить такой общий функционал в класс *Компонент* и наследовать их от него, чтобы не дублировать код.

Наследование от `Machine`

Базовый класс «машина» `Machine` будет реализовывать общего вида методы «включить» `enable()` и «выключить» `disable()`:

```
function Machine() {
  var enabled = false;

  this.enable = function() {
    enabled = true;
  };

  this.disable = function() {
    enabled = false;
  };
}
```

Унаследуем от него кофеварку. При этом она получит эти методы автоматически:

```
function CoffeeMachine(power) {
  Machine.call(this); // отнаследовать
```

```

var waterAmount = 0;

this.setWaterAmount = function(amount) {
  waterAmount = amount;
};
}

var coffeeMachine = new CoffeeMachine(10000);

coffeeMachine.enable();
coffeeMachine.setWaterAmount(100);
coffeeMachine.disable();

```

Наследование реализовано вызовом `Machine.call(this)` в начале конструктора `CoffeeMachine`.

Он вызывает функцию `Machine`, передавая ей в качестве контекста `this` текущий объект. `Machine`, в процессе выполнения, записывает в `this` различные полезные свойства и методы, в нашем случае `this.enable` и `this.disable`.

Далее конструктор `CoffeeMachine` продолжает выполнение и может добавить свои свойства и методы.

В результате мы получаем объект `coffeeMachine`, который включает в себя методы из `Machine` и `CoffeeMachine`.

Защищённые свойства

В коде выше есть одна проблема.

Наследник не имеет доступа к приватным свойствам родителя.

Иначе говоря, если кофеварка захочет обратиться к `enabled`, то её ждёт разочарование:

```

function Machine() {
  var enabled = false;

  this.enable = function() {
    enabled = true;
  };

  this.disable = function() {
    enabled = false;
  };
}

function CoffeeMachine(power) {
  Machine.call(this);

  this.enable();
}

// ошибка, переменная не определена!
alert( enabled );
}

var coffeeMachine = new CoffeeMachine(10000);

```

Это естественно, ведь `enabled` – локальная переменная функции `Machine`. Она находится в другой области видимости.

Чтобы наследник имел доступ к свойству, оно должно быть записано в `this`.

При этом, чтобы обозначить, что свойство является внутренним, его имя начинают с подчёркивания `_`.

```

function Machine() {
  this._enabled = false; // вместо var enabled

  this.enable = function() {
    this._enabled = true;
  };

  this.disable = function() {
    this._enabled = false;
  };
}

function CoffeeMachine(power) {
  Machine.call(this);

  this.enable();
}

alert( this._enabled ); // true
}

var coffeeMachine = new CoffeeMachine(10000);

```

Подчёркивание в начале свойства – общепринятый знак, что свойство является внутренним, предназначенным лишь для доступа из самого объекта и его наследников. Такие свойства называют *защищёнными*.

Технически, залезть в него из внешнего кода, конечно, возможно, но приличный программист так делать не будет.

Перенос свойства в защищённые

У `CoffeeMachine` есть приватное свойство `power`. Сейчас мы его тоже сделаем защищённым и перенесём в `Machine`, поскольку «мощность» свойственна всем машинам, а не только кофеварке.

```

function Machine(power) {
  this._power = power; // (1)

  this._enabled = false;

  this.enable = function() {
    this._enabled = true;
  };

  this.disable = function() {
    this._enabled = false;
  };
}

function CoffeeMachine(power) {
  Machine.apply(this, arguments); // (2)

  alert( this._enabled ); // false
  alert( this._power ); // 10000
}

var coffeeMachine = new CoffeeMachine(10000);

```

Теперь все машины `Machine` имеют мощность `power`. Обратим внимание, что мы из параметра конструктора сразу скопировали её в объект в строке (1). Иначе она была бы недоступна из наследников.

В строке (2) мы теперь вызываем не просто `Machine.call(this)`, а расширенный вариант: `Machine.apply(this, arguments)`, который вызывает `Machine` в текущем контексте вместе с передачей текущих аргументов.

Можно было бы использовать и более простой вызов `Machine.call(this, power)`, но использование `apply` гарантирует передачу всех аргументов, вдруг их количество увеличится – не надо будет переписывать.

Переопределение методов

Итак, мы получили класс `CoffeeMachine`, который наследует от `Machine`.

Аналогичным образом мы можем унаследовать от `Machine` холодильник `Fridge`, микроволновку `MicroOven` и другие классы, которые разделяют общий «машинный» функционал, то есть имеют мощность и их можно включать/выключать.

Для этого достаточно вызвать `Machine` в текущем контексте, а затем добавить свои методы.

```

// Fridge может добавить и свои аргументы,
// которые в Machine не будут использованы
function Fridge(power, temperature) {
  Machine.apply(this, arguments);
}

// ...

```

Бывает так, что реализация конкретного метода машины в наследнике имеет свои особенности.

Можно, конечно, объявить в `CoffeeMachine` свой `enable`:

```

function CoffeeMachine(power, capacity) {
  Machine.apply(this, arguments);

  // переопределить this.enable
  this.enable = function() {
    /* enable для кофеварки */
  };
}

```

...Однако, как правило, мы хотим не заменить, а *расширить* метод родителя, добавить к нему что-то. Например, сделать так, чтобы при включении кофеварка тут же запускалась.

Для этого метод родителя предварительно копируют в переменную, и затем вызывают внутри нового `enable` – там, где считают нужным:

```

function CoffeeMachine(power) {
  Machine.apply(this, arguments);

  var parentEnable = this.enable; // (1)
  this.enable = function() { // (2)
    parentEnable.call(this); // (3)
    this.run(); // (4)
  };
}

// ...

```

Общая схема переопределения метода (по строкам выделенного фрагмента кода):

1. Копируем доставшийся от родителя метод `this.enable` в переменную, например `parentEnable`.
2. Заменяем `this.enable` на свою функцию...
3. ...Которая по-прежнему реализует старый функционал через вызов `parentEnable`.
4. ...И в дополнение к нему делает что-то своё, например запускает приготовление кофе.

Обратим внимание на строку (3) .

В ней родительский метод вызывается так: `parentEnable.call(this)` . Если бы вызов был таким: `parentEnable()` , то ему бы не передан текущий `this` и возникла бы ошибка.

Технически, можно сделать возможность вызывать его и как `parentEnable()` , но тогда надо гарантировать, что контекст будет правильным, например привязать его при помощи `bind` или при объявлении, в родителе, вообще не использовать `this` , а получать контекст через замыкание, вот так:

```
function Machine(power) {
  this._enabled = false;

  var self = this;

  this.enable = function() {
    // используем внешнюю переменную вместо this
    self._enabled = true;
  };

  this.disable = function() {
    self._enabled = false;
  };
}

function CoffeeMachine(power) {
  Machine.apply(this, arguments);

  var waterAmount = 0;

  this.setWaterAmount = function(amount) {
    waterAmount = amount;
  };

  var parentEnable = this.enable;
  this.enable = function() {
    parentEnable(); // теперь можно вызывать как угодно, this не важен
    this.run();
  }

  function onReady() {
    alert( 'Кофе готово!' );
  }

  this.run = function() {
    setTimeout(onReady, 1000);
  };
}

var coffeeMachine = new CoffeeMachine(10000);
coffeeMachine.setWaterAmount(50);
coffeeMachine.enable();
```

В коде выше родительский метод `parentEnable = this.enable` успешно продолжает работать даже при вызове без контекста. А всё потому, что использует `self` внутри.

Итого

Организация наследования, которая описана в этой главе, называется «функциональным паттерном наследования».

Её общая схема (кратко):

1. Объявляется конструктор родителя `Machine` . В нём могут быть приватные (`private`), публичные (`public`) и защищённые (`protected`) свойства:

```
function Machine(params) {
  // локальные переменные и функции доступны только внутри Machine
  var privateProperty;

  // публичные доступны снаружи
  this.publicProperty = ...;

  // защищённые доступны внутри Machine и для потомков
  // мы договариваемся не трогать их снаружи
  this._protectedProperty = ...
}

var machine = new Machine(...);
machine.public();
```

2. Для наследования конструктор потомка вызывает родителя в своём контексте через `apply` . После чего может добавить свои переменные и методы:

```
function CoffeeMachine(params) {
  // универсальный вызов с передачей любых аргументов
  Machine.apply(this, arguments);

  this.coffeePublicProperty = ...
}

var coffeeMachine = new CoffeeMachine(...);
coffeeMachine.publicProperty();
coffeeMachine.coffeePublicProperty();
```

3. В `CoffeeMachine` свойства, полученные от родителя, можно перезаписать своими. Но обычно требуется не заменить, а расширить метод родителя. Для этого он предварительно копируется в переменную:

```
function CoffeeMachine(params) {
  Machine.apply(this, arguments);

  var parentProtected = this._protectedProperty;
  this._protectedProperty = function(args) {
    parentProtected.apply(this, args); // (*)
    // ...
  };
}
```

Строку (*) можно упростить до `parentProtected(args)`, если метод родителя не использует `this`, а, например, привязан к `var self = this`:

```
function Machine(params) {
  var self = this;

  this._protected = function() {
    self.property = "value";
  };
}
```

Надо сказать, что способ наследования, описанный в этой главе, используется нечасто.

В следующих главах мы будем изучать прототипный подход, который обладает рядом преимуществ.

Но знать и понимать его необходимо, поскольку во многих существующих библиотеках классы написаны в функциональном стиле, и расширять/наследовать от них можно только так.

✔ Задачи

Запускать только при включённой кофеварке

важность: 5

В коде `CoffeeMachine` сделайте так, чтобы метод `run` выводил ошибку, если кофеварка выключена.

В итоге должен работать такой код:

```
var coffeeMachine = new CoffeeMachine(10000);
coffeeMachine.run(); // ошибка, кофеварка выключена!
```

А вот так – всё в порядке:

```
var coffeeMachine = new CoffeeMachine(10000);
coffeeMachine.enable();
coffeeMachine.run(); // ...Кофе готов!
```

[Открыть песочницу для задачи.](#)

[К решению](#)

Останавливать кофеварку при выключении

важность: 5

Когда кофеварку выключают – текущая варка кофе должна останавливаться.

Например, следующий код кофе не сварит:

```
var coffeeMachine = new CoffeeMachine(10000);
coffeeMachine.enable();
coffeeMachine.run();
coffeeMachine.disable(); // остановит работу, ничего не выведет
```

Реализуйте это на основе решения [предыдущей задачи](#).

[К решению](#)

Унаследуйте холодильник

важность: 4

Создайте класс для холодильника `Fridge(power)`, наследующий от `Machine`, с приватным свойством `food` и методами `addFood(...)`, `getFood()`:

- Приватное свойство `food` хранит массив еды.
- Публичный метод `addFood(item)` добавляет в массив `food` новую еду, доступен вызов с несколькими аргументами `addFood(item1, item2...)` для добавления нескольких элементов сразу.
- Если холодильник выключен, то добавить еду нельзя, будет ошибка.

- Максимальное количество еды ограничено `power/100`, где `power` – мощность холодильника, указывается в конструкторе. При попытке добавить больше – будет ошибка
- Публичный метод `getFood()` возвращает еду в виде массива, добавление или удаление элементов из которого не должно влиять на свойство `food` холодильника.

Код для проверки:

```
var fridge = new Fridge(200);
fridge.addFood("котлета"); // ошибка, холодильник выключен
```

Ещё код для проверки:

```
// создать холодильник мощностью 500 (не более 5 еды)
var fridge = new Fridge(500);
fridge.enable();
fridge.addFood("котлета");
fridge.addFood("сок", "зелень");
fridge.addFood("варенье", "пирог", "торт"); // ошибка, слишком много еды
```

Код использования холодильника без ошибок:

```
var fridge = new Fridge(500);
fridge.enable();
fridge.addFood("котлета");
fridge.addFood("сок", "варенье");

var fridgeFood = fridge.getFood();
alert( fridgeFood ); // котлета, сок, варенье

// добавление элементов не влияет на еду в холодильнике
fridgeFood.push("вилка", "ложка");

alert( fridge.getFood() ); // внутри по-прежнему: котлета, сок, варенье
```

Исходный код класса `Machine`, от которого нужно наследовать:

```
function Machine(power) {
  this._power = power;
  this._enabled = false;

  var self = this;

  this.enable = function() {
    self._enabled = true;
  };

  this.disable = function() {
    self._enabled = false;
  };
}
```

[К решению](#)

Добавьте методы в холодильник

важность: 5

Добавьте в холодильник методы:

- Публичный метод `filterFood(func)`, который возвращает всю еду, для которой `func(item) == true`
- Публичный метод `removeFood(item)`, который удаляет еду `item` из холодильника.

Код для проверки:

```
var fridge = new Fridge(500);
fridge.enable();
fridge.addFood({
  title: "котлета",
  calories: 100
});
fridge.addFood({
  title: "сок",
  calories: 30
});
fridge.addFood({
  title: "зелень",
  calories: 10
});
fridge.addFood({
  title: "варенье",
  calories: 150
});

fridge.removeFood("нет такой еды"); // без эффекта
alert( fridge.getFood().length ); // 4

var dietItems = fridge.filterFood(function(item) {
  return item.calories < 50;
});

dietItems.forEach(function(item) {
```

```
    alert( item.title ); // сок, зелень
    fridge.removeFood(item);
  });

  alert( fridge.getFood().length ); // 2
```

В качестве исходного кода используйте решение [предыдущей задачи](#).

[К решению](#)

Переопределите disable

важность: 5

Переопределите метод `disable` холодильника, чтобы при наличии в нём еды он выдавал ошибку.

Код для проверки:

```
var fridge = new Fridge(500);
fridge.enable();
fridge.addFood("кус-кус");
fridge.disable(); // ошибка, в холодильнике есть еда
```

В качестве исходного кода используйте решение [предыдущей задачи](#).

[К решению](#)

ООП в прототипном стиле

В этом разделе мы изучим прототипы и классы на них – де-факто стандарт объектно-ориентированной разработки в JavaScript.

Прототип объекта

Объекты в JavaScript можно организовать в цепочки так, чтобы свойство, не найденное в одном объекте, автоматически искалось бы в другом.

Связующим звеном выступает специальное свойство `__proto__`.

Прототип proto

Если один объект имеет специальную ссылку `__proto__` на другой объект, то при чтении свойства из него, если свойство отсутствует в самом объекте, оно ищется в объекте `__proto__`.

Свойство `__proto__` доступно во всех браузерах, кроме IE10-, а в более старых IE оно, конечно же, тоже есть, но напрямую к нему не обратиться, требуются чуть более сложные способы, которые мы рассмотрим позднее.

Пример кода (кроме IE10-):

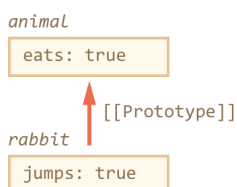
```
var animal = {
  eats: true
};
var rabbit = {
  jumps: true
};

rabbit.__proto__ = animal;

// в rabbit можно найти оба свойства
alert( rabbit.jumps ); // true
alert( rabbit.eats ); // true
```

1. Первый `alert` здесь работает очевидным образом – он выводит свойство `jumps` объекта `rabbit`.
2. Второй `alert` хочет вывести `rabbit.eats`, ищет его в самом объекте `rabbit`, не находит – и продолжает поиск в объекте `rabbit.__proto__`, то есть, в данном случае, в `animal`.

Иллюстрация происходящего при чтении `rabbit.eats` (поиск идет снизу вверх):



Объект, на который указывает ссылка `__proto__`, называется «*прототипом*». В данном случае получилось, что `animal` является прототипом для `rabbit`.

Также говорят, что объект `rabbit` «*прототипно наследует*» от `animal`.

Обратим внимание – прототип используется исключительно при чтении. Запись значения, например, `rabbit.eats = value` или удаление `delete rabbit.eats` – работает напрямую с объектом.

В примере ниже мы записываем свойство в сам `rabbit`, после чего `alert` перестаёт брать его у прототипа, а берёт уже из самого объекта:

```
var animal = {
  eats: true
};
var rabbit = {
  jumps: true,
  eats: false
};

rabbit.__proto__ = animal;

alert( rabbit.eats ); // false, свойство взято из rabbit
```

Другими словами, прототип – это «резервное хранилище свойств и методов» объекта, автоматически используемое при поиске.

У объекта, который является `__proto__`, может быть свой `__proto__`, у того – свой, и так далее. При этом свойства будут искаться по цепочке.

i Ссылка `proto` в спецификации

Если вы будете читать спецификацию ECMAScript – свойство `__proto__` обозначено в ней как `[[Prototype]]`.

Двойные квадратные скобки здесь важны, чтобы не перепутать его с совсем другим свойством, которое называется `prototype`, и которое мы рассмотрим позже.

Метод `hasOwnProperty`

Обычный цикл `for...in` не делает различия между свойствами объекта и его прототипа.

Он перебирает всё, например:

```
var animal = {
  eats: true
};

var rabbit = {
  jumps: true,
  __proto__: animal
};

for (var key in rabbit) {
  alert( key + " = " + rabbit[key] ); // выводит и "eats" и "jumps"
}
```

Иногда хочется посмотреть, что находится именно в самом объекте, а не в прототипе.

Вызов `obj.hasOwnProperty(prop)` [↗](#) возвращает `true`, если свойство `prop` принадлежит самому объекту `obj`, иначе `false`.

Например:

```
var animal = {
  eats: true
};

var rabbit = {
  jumps: true,
  __proto__: animal
};

alert( rabbit.hasOwnProperty('jumps') ); // true: jumps принадлежит rabbit
alert( rabbit.hasOwnProperty('eats') ); // false: eats не принадлежит
```

Для того, чтобы перебрать свойства самого объекта, достаточно профильтровать `key` через `hasOwnProperty`:

```
var animal = {
  eats: true
};

var rabbit = {
  jumps: true,
  __proto__: animal
};

for (var key in rabbit) {
  if (!rabbit.hasOwnProperty(key)) continue; // пропустить "не свои" свойства
  alert( key + " = " + rabbit[key] ); // выводит только "jumps"
}
```

`Object.create(null)`

Зачастую объекты используют для хранения произвольных значений по ключу, как коллекцию:

```
var data = {};  
data.text = "Привет";  
data.age = 35;  
// ...
```

При дальнейшем поиске в этой коллекции мы найдём не только `text` и `age`, но и встроенные функции:

```
var data = {};  
alert(data.toString); // функция, хотя мы её туда не записывали
```

Это может быть неприятным сюрпризом и приводить к ошибкам, если названия свойств приходят от посетителя и могут быть произвольными.

Чтобы этого избежать, мы можем исключать свойства, не принадлежащие самому объекту:

```
var data = {};  
  
// выведет toString только если оно записано в сам объект  
alert(data.hasOwnProperty('toString') ? data.toString : undefined);
```

Однако, есть путь и проще:

```
var data = Object.create(null);  
data.text = "Привет";  
  
alert(data.text); // Привет  
alert(data.toString); // undefined
```

Объект, создаваемый при помощи `Object.create(null)` не имеет прототипа, а значит в нём нет лишних свойств. Для коллекции – как раз то, что надо.

Методы для работы с proto

В современных браузерах есть два дополнительных метода для работы с `__proto__`. Зачем они нужны, если есть `__proto__`? В общем-то, не очень нужны, но по историческим причинам тоже существуют.

Чтение: [Object.getPrototypeOf\(obj\)](#) ↗

Возвращает `obj.__proto__` (кроме IE8-)

Запись: [Object.setPrototypeOf\(obj, proto\)](#) ↗

Устанавливает `obj.__proto__ = proto` (кроме IE10-).

Кроме того, есть ещё один вспомогательный метод:

Создание объекта с прототипом: [Object.create\(proto, descriptors\)](#) ↗

Создаёт пустой объект с `__proto__`, равным первому аргументу (кроме IE8-), второй необязательный аргумент может содержать [дескрипторы свойств](#).

Итого

- В JavaScript есть встроенное «наследование» между объектами при помощи специального свойства `__proto__`.
- При установке свойства `rabbit.__proto__ = animal` говорят, что объект `animal` будет «прототипом» `rabbit`.
- При чтении свойства из объекта, если его в нём нет, оно ищется в `__proto__`. Прототип задействуется только при чтении свойства. Операции присвоения `obj.prop =` или удаления `delete obj.prop` совершаются всегда над самим объектом `obj`.

Несколько прототипов одному объекту присвоить нельзя, но можно организовать объекты в цепочку, когда один объект ссылается на другой при помощи `__proto__`, тот ссылается на третий, и так далее.

В современных браузерах есть методы для работы с прототипом:

- [Object.getPrototypeOf\(obj\)](#) ↗ (кроме IE8-)
- [Object.setPrototypeOf\(obj, proto\)](#) ↗ (кроме IE10-)
- [Object.create\(proto, descriptors\)](#) ↗ (кроме IE8-)

Возможно, вас смущает недостаточная поддержка `__proto__` в старых IE. Но это не страшно. В последующих главах мы рассмотрим дополнительные методы работы с `__proto__`, включая те, которые работают везде.

Также мы рассмотрим, как свойство `__proto__` используется внутри самого языка JavaScript и как организовать классы с его помощью.

✔ Задачи

Чему равно свойство после delete?

важность: 5

Какие значения будут выводиться в коде ниже?

```
var animal = {
  jumps: null
};
var rabbit = {
  jumps: true
};

rabbit.__proto__ = animal;

alert( rabbit.jumps ); // ? (1)

delete rabbit.jumps;

alert( rabbit.jumps ); // ? (2)

delete animal.jumps;

alert( rabbit.jumps ); // ? (3)
```

Итого три вопроса.

[К решению](#)

Прототип и this

важность: 5

Сработает ли вызов `rabbit.eat()` ?

Если да, то в какой именно объект он запишет свойство `full`: в `rabbit` или `animal` ?

```
var animal = {
  eat: function() {
    this.full = true;
  }
};

var rabbit = {
  __proto__: animal
};

rabbit.eat();
```

[К решению](#)

Алгоритм для поиска

важность: 5

Есть объекты:

```
var head = {
  glasses: 1
};

var table = {
  pen: 3
};

var bed = {
  sheet: 1,
  pillow: 2
};

var pockets = {
  money: 2000
};
```

Задание состоит из двух частей:

1. Присвойте объектам ссылки `__proto__` так, чтобы любой поиск чего-либо шёл по алгоритму `pockets -> bed -> table -> head`.
То есть `pockets.pen == 3`, `bed.glasses == 1`, но `table.money == undefined`.
2. После этого ответьте на вопрос, как быстрее искать `glasses`: обращением к `pockets.glasses` или `head.glasses`? Попробуйте протестировать.

[К решению](#)

Свойство `F.prototype` и создание объектов через `new`

До этого момента мы говорили о наследовании объектов, объявленных через `{...}`.

Но в реальных проектах объекты обычно создаются функцией-конструктором через `new`. Посмотрим, как указать прототип в этом случае.

Свойство `F.prototype`

Самым очевидным решением является назначение `__proto__` в конструкторе.

Например, если я хочу, чтобы у всех объектов, которые создаются `new Rabbit`, был прототип `animal`, я могу сделать так:

```
var animal = {
  eats: true
};

function Rabbit(name) {
  this.name = name;
  this.__proto__ = animal;
}

var rabbit = new Rabbit("Кроль");
alert( rabbit.eats ); // true, из прототипа
```

Недостаток этого подхода – он не работает в IE10-.

К счастью, в JavaScript с древнейших времён существует альтернативный, встроенный в язык и полностью кросс-браузерный способ.

Чтобы новым объектам автоматически ставить прототип, конструктору ставится свойство `prototype`.

При создании объекта через `new`, в его прототип `__proto__` записывается ссылка из `prototype` функции-конструктора.

Например, код ниже полностью аналогичен предыдущему, но работает всегда и везде:

```
var animal = {
  eats: true
};

function Rabbit(name) {
  this.name = name;
}

Rabbit.prototype = animal;

var rabbit = new Rabbit("Кроль"); // rabbit.__proto__ == animal
alert( rabbit.eats ); // true
```

Установка `Rabbit.prototype = animal` буквально говорит интерпретатору следующее: "При создании объекта через `new Rabbit` запиши ему `__proto__ = animal`".

i Свойство `prototype` имеет смысл только у конструктора

Свойство с именем `prototype` можно указать на любом объекте, но особый смысл оно имеет, лишь если назначено функции-конструктору.

Само по себе, без вызова оператора `new`, оно вообще ничего не делает, его единственное назначение – указывать `__proto__` для новых объектов.

! Значением `prototype` может быть только объект

Технически, в это свойство можно записать что угодно.

Однако, при работе `new`, свойство `prototype` будет использовано лишь в том случае, если это объект. Примитивное значение, такое как число или строка, будет проигнорировано.

Свойство `constructor`

У каждой функции по умолчанию уже есть свойство `prototype`.

Оно содержит объект такого вида:

```
function Rabbit() {}

Rabbit.prototype = {
  constructor: Rabbit
};
```

В коде выше я создал `Rabbit.prototype` вручную, но ровно такой же – генерируется автоматически.

Проверим:

```
function Rabbit() {}

// в Rabbit.prototype есть одно свойство: constructor
alert( Object.getOwnPropertyNames(Rabbit.prototype) ); // constructor

// оно равно Rabbit
alert( Rabbit.prototype.constructor == Rabbit ); // true
```

Можно его использовать для создания объекта с тем же конструктором, что и данный:

```
function Rabbit(name) {
  this.name = name;
  alert( name );
}

var rabbit = new Rabbit("Кроль");
var rabbit2 = new rabbit.constructor("Крольчиха");
```

Эта возможность бывает полезна, когда, получив объект, мы не знаем в точности, какой у него был конструктор (например, сделан вне нашего кода), а нужно создать такой же.

⚠ Свойство **constructor** легко потерять

JavaScript никак не использует свойство `constructor`. То есть, оно создаётся автоматически, а что с ним происходит дальше – это уже наша забота. В стандарте прописано только его создание.

В частности, при перезаписи `Rabbit.prototype = { jumps: true }` свойства `constructor` больше не будет.

Сам интерпретатор JavaScript его в служебных целях не требует, поэтому в работе объектов ничего не «сломается». Но если мы хотим, чтобы возможность получить конструктор, всё же, была, то можно при перезаписи гарантировать наличие `constructor` вручную:

```
Rabbit.prototype = {
  jumps: true,
  constructor: Rabbit
};
```

Либо можно поступить аккуратно и добавить свойства к встроенному `prototype` без его замены:

```
// сохранится встроенный constructor
Rabbit.prototype.jumps = true
```

Эмуляция `Object.create` для IE8-

Как мы только что видели, с конструкторами всё просто, назначить прототип можно кросс-браузерно при помощи `F.prototype`.

Теперь небольшое «лирическое отступление» в область совместимости.

Прямые методы работы с прототипом отсутствуют в старых IE, но один из них – `Object.create(proto)` можно эмулировать, как раз при помощи `prototype`. И он будет работать везде, даже в самых устаревших браузерах.

Кросс-браузерный аналог – назовём его `inherit`, состоит буквально из нескольких строк:

```
function inherit(proto) {
  function F() {}
  F.prototype = proto;
  var object = new F;
  return object;
}
```

Результат вызова `inherit(animal)` идентичен `Object.create(animal)`. Она создаёт новый пустой объект с прототипом `animal`.

Например:

```
var animal = {
  eats: true
};

var rabbit = inherit(animal);
alert( rabbit.eats ); // true
```

Посмотрите внимательно на функцию `inherit` и вы, наверняка, сами поймёте, как она работает...

Если где-то неясности, то её построчное описание:

```
function inherit(proto) {
  function F() {} // (1)
  F.prototype = proto // (2)
  var object = new F; // (3)
  return object; // (4)
}
```

1. Создана новая функция `F`. Она ничего не делает с `this`, так что если вызвать `new F`, то получим пустой объект.
2. Свойство `F.prototype` устанавливается в будущий прототип `proto`
3. Результатом вызова `new F` будет пустой объект с `__proto__` равным значению `F.prototype`.
4. Мы получили пустой объект с заданным прототипом, как и хотели. Возвратим его.

Для унификации можно запустить такой код, и метод `Object.create` станет кросс-браузерным:

```
if (!Object.create) Object.create = inherit; /* определение inherit - выше */
```

В частности, аналогичным образом работает библиотека [es5-shim](#), при подключении которой `Object.create` станет доступен для всех браузеров.

Итого

Для произвольной функции – назовём её `Person`, верно следующее:

- Прототип `__proto__` новых объектов, создаваемых через `new Person`, можно задавать при помощи свойства `Person.prototype`.
- Значением `Person.prototype` по умолчанию является объект с единственным свойством `constructor`, содержащим ссылку на `Person`. Его можно использовать, чтобы из самого объекта получить функцию, которая его создала. Однако, JavaScript никак не поддерживает корректность этого свойства, поэтому программист может его изменить или удалить.
- Современный метод `Object.create(proto)` можно эмулировать при помощи `prototype`, если хочется, чтобы он работал в IE8-.

✔ Задачи

Прототип после создания

важность: 5

В примерах ниже создаётся объект `new Rabbit`, а затем проводятся различные действия с `prototype`.

Каковы будут результаты выполнения? Почему?

Начнём с этого кода. Что он выведет?

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

var rabbit = new Rabbit();

alert( rabbit.eats );
```

Добавили строку (выделена), что будет теперь?

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

var rabbit = new Rabbit();

Rabbit.prototype = {};

alert( rabbit.eats );
```

А если код будет такой? (заменена одна строка):

```
function Rabbit(name) {}
Rabbit.prototype = {
  eats: true
};

var rabbit = new Rabbit();

Rabbit.prototype.eats = false;

alert( rabbit.eats );
```

А такой? (заменена одна строка)

```
function Rabbit(name) {}
Rabbit.prototype = {
  eats: true
};

var rabbit = new Rabbit();

delete rabbit.eats; // (*)

alert( rabbit.eats );
```

И последний вариант:

```
function Rabbit(name) {}
Rabbit.prototype = {
  eats: true
};

var rabbit = new Rabbit();

delete Rabbit.prototype.eats; // (*)
```



```
alert( rabbit.eats );
```

[К решению](#)

Аргументы по умолчанию

важность: 4

Есть функция `Menu`, которая получает аргументы в виде объекта `options`:

```
/* options содержит настройки меню: width, height и т.п. */
function Menu(options) {
  ...
}
```

Ряд опций должны иметь значение по умолчанию. Мы могли бы проставить их напрямую в объекте `options`:

```
function Menu(options) {
  options.width = options.width || 300; // по умолчанию ширина 300
  ...
}
```

...Но такие изменения могут привести к непредвиденным результатам, т.к. объект `options` может быть повторно использован во внешнем коде. Он передается в `Menu` для того, чтобы параметры из него читали, а не писали.

Один из способов безопасно назначить значения по умолчанию – скопировать все свойства `options` в локальные переменные и затем уже менять. Другой способ – клонировать `options` путём копирования всех свойств из него в новый объект, который уже изменяется.

При помощи наследования и `Object.create` предложите третий способ, который позволяет избежать копирования объекта и не требует новых переменных.

[К решению](#)

Есть ли разница между вызовами?

важность: 5

Создадим новый объект, вот такой:

```
function Rabbit(name) {
  this.name = name;
}
Rabbit.prototype.sayHi = function() {
  alert( this.name );
}
var rabbit = new Rabbit("Rabbit");
```

Одинаково ли сработают эти вызовы?

```
rabbit.sayHi();
Rabbit.prototype.sayHi();
Object.getPrototypeOf(rabbit).sayHi();
rabbit.__proto__.sayHi();
```

Все ли они являются кросс-браузерными? Если нет – в каких браузерах сработает каждый?

[К решению](#)

Создать объект тем же конструктором

важность: 5

Пусть у нас есть произвольный объект `obj`, созданный каким-то конструктором, каким – мы не знаем, но хотели бы создать новый объект с его помощью.

Сможем ли мы сделать так?

```
var obj2 = new obj.constructor();
```

Приведите пример конструкторов для `obj`, при которых такой код будет работать верно – и неверно.

[К решению](#)

Встроенные "классы" в JavaScript

В JavaScript есть встроенные объекты: Date, Array, Object и другие. Они используют прототипы и демонстрируют организацию «псевдоклассов» на JavaScript, которую мы вполне можем применить и для себя.

Откуда методы у {} ?

Начнём мы с того, что создадим пустой объект и выведем его.

```
var obj = {};  
alert( obj ); // "[object Object]" ?
```

Где код, который генерирует строковое представление для alert(obj) ? Объект-то ведь пустой.

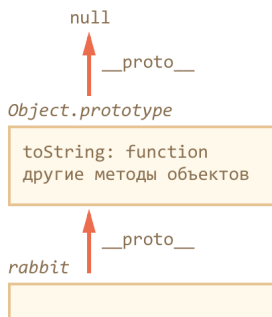
Object.prototype

...Конечно же, это сделал метод toString, который находится... Конечно, не в самом объекте (он пуст), а в его прототипе obj.__proto__, можно его даже вывести:

```
alert( {}.__proto__.toString ); // function toString
```

Откуда новый объект obj получает такой __proto__ ?

1. Запись obj = {} является краткой формой obj = new Object, где Object – встроенная функция-конструктор для объектов.
2. При выполнении new Object, создаваемому объекту ставится __proto__ по prototype конструктора, который в данном случае равен встроенному Object.prototype.
3. В дальнейшем при обращении к obj.toString() – функция будет взята из Object.prototype.

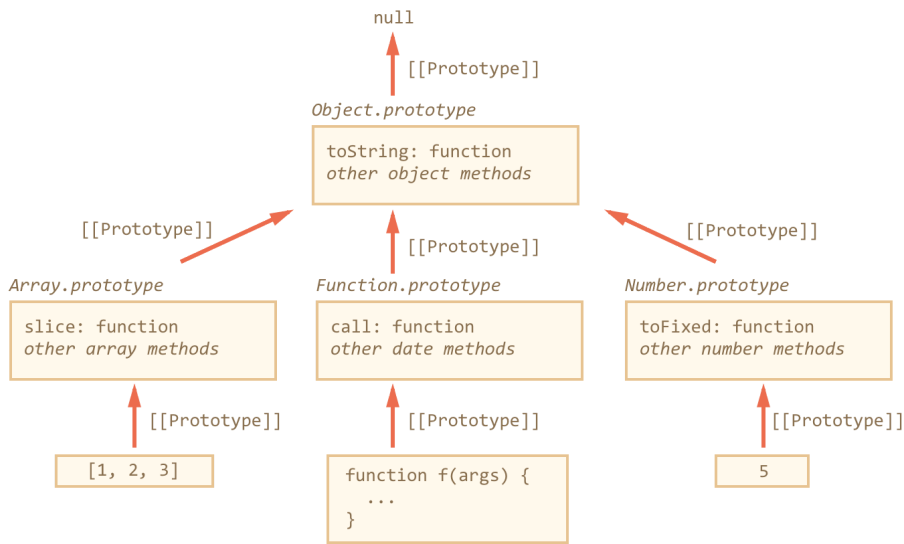


Это можно легко проверить:

```
var obj = {};  
  
// метод берётся из прототипа?  
alert( obj.toString == Object.prototype.toString ); // true, да  
  
// проверим, правда ли что __proto__ это Object.prototype?  
alert( obj.__proto__ == Object.prototype ); // true  
  
// А есть ли __proto__ у Object.prototype?  
alert( obj.__proto__.__proto__ ); // null, нет
```

Встроенные «классы» в JavaScript

Точно такой же подход используется в массивах Array, функциях Function и других объектах. Встроенные методы для них находятся в Array.prototype, Function.prototype и т.п.



Например, когда мы создаём массив, `[1, 2, 3]`, то это альтернативный вариант синтаксиса `new Array`, так что у массивов есть стандартный прототип `Array.prototype`.

Но в нём есть методы лишь для массивов, а для общих методов всех объектов есть ссылка `Array.prototype.__proto__`, равная `Object.prototype`.

Аналогично, для функций.

Лишь для чисел (как и других примитивов) всё немного иначе, но об этом чуть далее.

Объект `Object.prototype` – вершина иерархии, единственный, у которого `__proto__` равно `null`.

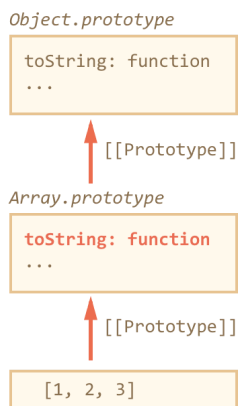
Поэтому говорят, что "все объекты наследуют от `Object`", а если более точно, то от `Object.prototype`.

«Псевдоклассом» или, более коротко, «классом», называют функцию-конструктор вместе с её `prototype`. Такой способ объявления классов называют «прототипным стилем ООП».

При наследовании часть методов переопределяется, например, у массива `Array` есть свой `toString`, который выводит элементы массива через запятую:

```
var arr = [1, 2, 3]
alert( arr ); // 1,2,3 <-- результат Array.prototype.toString
```

Как мы видели раньше, у `Object.prototype` есть свой `toString`, но так как в `Array.prototype` он ищется первым, то берётся именно вариант для массивов:



Вызов методов через `call` и `apply` из прототипа

Ранее мы говорили о применении методов массивов к «псевдомассивам», например, можно использовать `[].join` для `arguments` :

```
function showList() {
  alert( [].join.call(arguments, " - "));
}

showList("Вася", "Паша", "Маша"); // Вася - Паша - Маша
```

Так как метод `join` находится в `Array.prototype`, то можно вызвать его оттуда напрямую, вот так:

```
function showList() {
  alert( Array.prototype.join.call(arguments, " - "));
}

showList("Вася", "Паша", "Маша"); // Вася - Паша - Маша
```

Это эффективнее, потому что не создаётся лишний объект массива `[]`, хотя, с другой стороны – больше букв писать.

Примитивы

Примитивы не являются объектами, но методы берут из соответствующих прототипов: `Number.prototype`, `Boolean.prototype`, `String.prototype`.

По стандарту, если обратиться к свойству числа, строки или логического значения, то будет создан объект соответствующего типа, например `new String` для строки, `new Number` для чисел, `new Boolean` – для логических выражений.

Далее будет произведена операция со свойством или вызов метода по обычным правилам, с поиском в прототипе, а затем этот объект будет уничтожен.

Именно так работает код ниже:

```
var user = "Вася"; // создали строку (примитив)

alert( user.toUpperCase() ); // ВАСЯ
// был создан временный объект new String
// вызван метод
// new String уничтожен, результат возвращён
```

Можно даже попробовать записать в этот временный объект свойство:

```
// попытаемся записать свойство в строку:
var user = "Вася";
user.age = 30;

alert( user.age ); // undefined
```

Свойство `age` было записано во временный объект, который был тут же уничтожен, так что смысла в такой записи немного.

Конструкторы `String/Number/Boolean` – только для внутреннего использования

Технически, можно создавать объекты для примитивов и вручную, например `new Number`. Но в ряде случаев получится откровенно бредовое поведение. Например:

```
alert( typeof 1 ); // "number"
alert( typeof new Number(1) ); // "object" !?!
```

Или, ещё страннее:

```
var zero = new Number(0);

if (zero) { // объект - true, так что alert выполнится
  alert( "число ноль -- true?!?" );
}
```

Поэтому в явном виде `new String`, `new Number` и `new Boolean` никогда не вызываются.

Значения `null` и `undefined` не имеют свойств

Значения `null` и `undefined` стоят особняком. Вышесказанное к ним не относится.

Для них нет соответствующих классов, в них нельзя записать свойство (будет ошибка), в общем, на конкурсе «самое примитивное значение» они точно разделили бы первое место.

Изменение встроенных прототипов

Встроенные прототипы можно изменять. В том числе – добавлять свои методы.

Мы можем написать метод для многократного повторения строки, и он тут же станет доступным для всех строк:

```
String.prototype.repeat = function(times) {
    return new Array(times + 1).join(this);
};

alert( "ля".repeat(3) ); // ляляля
```

Аналогично мы могли бы создать метод `Object.prototype.each(func)`, который будет применять `func` к каждому свойству:

```
Object.prototype.each = function(f) {
    for (var prop in this) {
        var value = this[prop];
        f.call(value, prop, value); // вызовет f(prop, value), this=value
    }
}

// Попробуем! (внимание, пока что это работает неверно!)
var user = {
    name: 'Вася',
    age: 25
};

user.each(function(prop, val) {
    alert( prop ); // name -> age -> (!) each
});
```

Обратите внимание – пример выше работает не совсем корректно. Вместе со свойствами объекта `user` он выводит и наше свойство `each`. Технически, это правильно, так как цикл `for...in` перебирает свойства и в прототипе тоже, но не очень удобно.

Конечно, это легко поправить добавлением проверки `hasOwnProperty`:

```
Object.prototype.each = function(f) {
    for (var prop in this) {
        // пропускать свойства из прототипа
        if (!this.hasOwnProperty(prop)) continue;

        var value = this[prop];
        f.call(value, prop, value);
    }
};

// Теперь все будет в порядке
var obj = {
    name: 'Вася',
    age: 25
};

obj.each(function(prop, val) {
    alert( prop ); // name -> age
});
```

Здесь это сработало, теперь код работает верно. Но мы же не хотим добавлять `hasOwnProperty` в цикл по любому объекту! Поэтому либо не добавляйте свойства в `Object.prototype`, либо можно использовать [дескриптор свойства](#) и флаг `enumerable`.

Это, конечно, не будет работать в IE8-:

```
Object.prototype.each = function(f) {
    for (var prop in this) {
        var value = this[prop];
        f.call(value, prop, value);
    }
};

// поправить объявление свойства, установив флаг enumerable: false
Object.defineProperty(Object.prototype, 'each', {
    enumerable: false
});

// Теперь все будет в порядке
var obj = {
    name: 'Вася',
    age: 25
};

obj.each(function(prop, val) {
    alert( prop ); // name -> age
});
```

Есть несколько «за» и «против» модификации встроенных прототипов:

Достоинства

Недостатки

- Методы в прототипе автоматически доступны везде, их вызов прост и красив.

- Новые свойства, добавленные в прототип из разных мест, могут конфликтовать между собой. Представьте, что вы подключили две библиотеки, которые добавили одно и то же свойство в прототип, но определили его по-разному. Конфликт неизбежен.
- Изменения встроенных прототипов влияют глобально, на все-все скрипты, делать их не очень хорошо с архитектурной точки зрения.

Как правило, минусы весомее, но есть одно исключение, когда изменения встроенных прототипов не только разрешены, но и приветствуются.

Допустимо изменение прототипа встроенных объектов, которое добавляет поддержку метода из современных стандартов в те браузеры, где её пока нет.

Например, добавим `Object.create(proto)` в старые браузеры:

```
if (!Object.create) {
  Object.create = function(proto) {
    function F() {}
    F.prototype = proto;
    return new F;
  };
}
```

Именно так работает библиотека [es5-shim](#), которая предоставляет многие функции современного JavaScript для старых браузеров. Они добавляются во встроенные объекты и их прототипы.

Итого

- Методы встроенных объектов хранятся в их прототипах.
- Встроенные прототипы можно расширить или поменять.
- Добавление методов в `Object.prototype`, если оно не сопровождается `Object.defineProperty` с установкой `enumerable` (IE9+), «сломает» циклы `for...in`, поэтому стараются в этот прототип методы не добавлять.

Другие прототипы изменять менее опасно, но все же не рекомендуется во избежание конфликтов.

Отдельно стоит изменение с целью добавления современных методов в старые браузеры, таких как [Object.create](#), [Object.keys](#), [Function.prototype.bind](#) и т.п. Это допустимо и как раз делается [es5-shim](#).

✔ Задачи

Добавить функциям defer

важность: 5

Добавьте всем функциям в прототип метод `defer(ms)`, который откладывает вызов функции на `ms` миллисекунд.

После этого должен работать такой код:

```
function f() {
  alert( "привет" );
}

f.defer(1000); // выведет "привет" через 1 секунду
```

[К решению](#)

Добавить функциям defer с аргументами

важность: 4

Добавьте всем функциям в прототип метод `defer(ms)`, который возвращает обёртку, откладывающую вызов функции на `ms` миллисекунд.

Например, должно работать так:

```
function f(a, b) {
  alert( a + b );
}

f.defer(1000)(1, 2); // выведет 3 через 1 секунду.
```

То есть, должны корректно передаваться аргументы.

[К решению](#)

Свои классы на прототипах

Используем ту же структуру, что JavaScript использует внутри себя, для объявления своих классов.

Обычный конструктор

Вспомним, как мы объявляли классы ранее.

Например, этот код задаёт класс `Animal` в функциональном стиле, без всяких прототипов:

```
function Animal(name) {
  this.speed = 0;
  this.name = name;

  this.run = function(speed) {
    this.speed += speed;
    alert( this.name + ' бежит, скорость ' + this.speed );
  };

  this.stop = function() {
    this.speed = 0;
    alert( this.name + ' стоит' );
  };
};

var animal = new Animal('Зверь');

alert( animal.speed ); // 0, начальная скорость
animal.run(3); // Зверь бежит, скорость 3
animal.run(10); // Зверь бежит, скорость 13
animal.stop(); // Зверь стоит
```

Класс через прототип

А теперь создадим аналогичный класс, используя прототипы, наподобие того, как сделаны классы `Object`, `Date` и остальные.

Чтобы объявить свой класс, нужно:

1. Объявить функцию-конструктор.
2. Записать методы и свойства, нужные всем объектам класса, в `prototype`.

Опишем класс `Animal`:

```
// конструктор
function Animal(name) {
  this.name = name;
  this.speed = 0;
}

// методы в прототипе
Animal.prototype.run = function(speed) {
  this.speed += speed;
  alert( this.name + ' бежит, скорость ' + this.speed );
};

Animal.prototype.stop = function() {
  this.speed = 0;
  alert( this.name + ' стоит' );
};

var animal = new Animal('Зверь');

alert( animal.speed ); // 0, свойство взято из прототипа
animal.run(5); // Зверь бежит, скорость 5
animal.run(5); // Зверь бежит, скорость 10
animal.stop(); // Зверь стоит
```

В объекте `animal` будут храниться свойства конкретного экземпляра: `name` и `speed`, а общие методы – в прототипе.

Совершенно такой же подход, как и для встроенных классов в JavaScript.

Сравнение

Чем такое задание класса лучше и хуже функционального стиля?

Достоинства

- Функциональный стиль записывает в каждый объект и свойства и методы, а прототипный – только свойства. Поэтому прототипный стиль – быстрее и экономнее по памяти.

Недостатки

- При создании методов через прототип, мы теряем возможность использовать локальные переменные как приватные свойства, у них больше нет общей области видимости с конструктором.

Таким образом, прототипный стиль – быстрее и экономнее, но немного менее удобен.

К примеру, есть у нас приватное свойство `name` и метод `sayHi` в функциональном стиле ООП:

```
function Animal(name) {
  this.sayHi = function() {
    alert( name );
  };
}

var animal = new Animal("Зверь");
animal.sayHi(); // Зверь
```

При задании методов в прототипе мы не сможем её так оставить, ведь методы находятся *вне* конструктора, у них нет общей области видимости, поэтому приходится записывать `name` в сам объект, обозначив его как защищённое:

```
function Animal(name) {
  this._name = name;
}

Animal.prototype.sayHi = function() {
  alert( this._name );
}

var animal = new Animal("Зверь");
animal.sayHi(); // Зверь
```

Впрочем, недостаток этот – довольно условный. Ведь при наследовании в функциональном стиле также пришлось бы писать `this._name`, чтобы потомок получил доступ к этому значению.

✔ Задачи

Перепишите в виде класса

важность: 5

Есть класс `CoffeeMachine`, заданный в функциональном стиле.

Задача: переписать `CoffeeMachine` в виде класса с использованием прототипа.

Исходный код:

```
function CoffeeMachine(power) {
  var waterAmount = 0;

  var WATER_HEAT_CAPACITY = 4200;

  function getTimeToBoil() {
    return waterAmount * WATER_HEAT_CAPACITY * 80 / power;
  }

  this.run = function() {
    setTimeout(function() {
      alert( 'Кофе готов!' );
    }, getTimeToBoil());
  };

  this.setWaterAmount = function(amount) {
    waterAmount = amount;
  };
}

var coffeeMachine = new CoffeeMachine(10000);
coffeeMachine.setWaterAmount(50);
coffeeMachine.run();
```

P.S. При описании через прототипы локальные переменные недоступны методам, поэтому нужно будет переделать их в защищённые свойства.

[К решению](#)

Хомяки с `__proto__`

важность: 5

Вы – руководитель команды, которая разрабатывает игру, хомяковую ферму. Один из программистов получил задание создать класс «хомяк» (англ – "Hamster").

Объекты-хомяки должны иметь массив `food` для хранения еды и метод `found` для добавления.

Ниже – его решение. При создании двух хомяков, если поел один – почему-то сытым становится и второй тоже.

В чём дело? Как поправить?

```
function Hamster() {}

Hamster.prototype.food = []; // пустой "живот"
```



```

Hamster.prototype.found = function(something) {
  this.food.push(something);
};

// Создаём двух хомяков и кормим первого
var speedy = new Hamster();
var lazy = new Hamster();

speedy.found("яблоко");
speedy.found("орех");

alert( speedy.food.length ); // 2
alert( lazy.food.length ); // 2 (!??)

```

[К решению](#)

Наследование классов в JavaScript

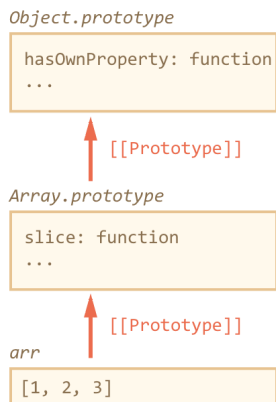
Наследование на уровне объектов в JavaScript, как мы видели, реализуется через ссылку `__proto__` .

Теперь поговорим о наследовании на уровне классов, то есть когда объекты, создаваемые, к примеру, через `new Admin` , должны иметь все методы, которые есть у объектов, создаваемых через `new User` , и ещё какие-то свои.

Наследование Array от Object

Для реализации наследования в наших классах мы будем использовать тот же подход, который принят внутри JavaScript.

Взглянем на него ещё раз на примере `Array` , который наследует от `Object` :



- Методы массивов `Array` хранятся в `Array.prototype` .
- `Array.prototype` имеет прототипом `Object.prototype` .

Поэтому когда экземпляры класса `Array` хотят получить метод массива – они берут его из своего прототипа, например `Array.prototype.slice` .

Если же нужен метод объекта, например, `hasOwnProperty` , то его в `Array.prototype` нет, и он берётся из `Object.prototype` .

Отличный способ «потрогать это руками» – запустить в консоли команду `console.dir([1,2,3])` .

Вывод в Chrome будет примерно таким:

```

> console.dir([1,2,3])
└─> Array[3]
  0: 1
  1: 2
  2: 3
  length: 3
  __proto__: Array.prototype
    ▶ concat: function concat() { [native code] }
    ▶ ...
    ▶ unshift: function unshift() { [native code] }
    __proto__: Object.prototype
      ▶ ...
      ▶ constructor: function Object() { [native code] }
      ▶ hasOwnProperty: function hasOwnProperty() { [native code] }
      ▶ isPrototypeOf: function isPrototypeOf() { [native code] }
      ▶ ...

```

Здесь отчётливо видно, что сами данные и `length` находятся в массиве, дальше в `__proto__` идут методы для массивов `concat` , то есть `Array.prototype` , а далее – `Object.prototype` .

i console.dir для доступа к свойствам

Обратите внимание, я использовал именно `console.dir` , а не `console.log` , поскольку `log` зачастую выводит объект в виде строки, без доступа к свойствам.

Наследование в наших классах

Применим тот же подход для наших классов: объявим класс `Rabbit`, который будет наследовать от `Animal`.

Вначале создадим два этих класса по отдельности, они пока что будут совершенно независимы.

`Animal`:

```
function Animal(name) {
  this.name = name;
  this.speed = 0;
}

Animal.prototype.run = function(speed) {
  this.speed += speed;
  alert( this.name + ' бежит, скорость ' + this.speed );
};

Animal.prototype.stop = function() {
  this.speed = 0;
  alert( this.name + ' стоит' );
};
```

`Rabbit`:

```
function Rabbit(name) {
  this.name = name;
  this.speed = 0;
}

Rabbit.prototype.jump = function() {
  this.speed++;
  alert( this.name + ' прыгает' );
};

var rabbit = new Rabbit('Кроль');
```

Для того, чтобы наследование работало, объект `rabbit = new Rabbit` должен использовать свойства и методы из своего прототипа `Rabbit.prototype`, а если их там нет, то – свойства и метода родителя, которые хранятся в `Animal.prototype`.

Если ещё короче – порядок поиска свойств и методов должен быть таким: `rabbit -> Rabbit.prototype -> Animal.prototype`, по аналогии с тем, как это сделано для объектов и массивов.

Для этого можно поставить ссылку `__proto__` с `Rabbit.prototype` на `Animal.prototype`.

Можно сделать это так:

```
Rabbit.prototype.__proto__ = Animal.prototype;
```

Однако, прямой доступ к `__proto__` не поддерживается в IE10-, поэтому для поддержки этих браузеров мы используем функцию `Object.create`. Она либо встроена либо легко эмулируется во всех браузерах.

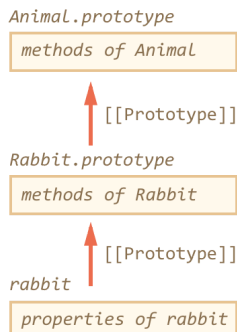
Класс `Animal` остаётся без изменений, а `Rabbit.prototype` мы будем создавать с нужным прототипом, используя `Object.create`:

```
function Rabbit(name) {
  this.name = name;
  this.speed = 0;
}

// задаём наследование
Rabbit.prototype = Object.create(Animal.prototype);

// и добавим свой метод (или методы...)
Rabbit.prototype.jump = function() { ... };
```

Теперь выглядеть иерархия будет так:



В `prototype` по умолчанию всегда находится свойство `constructor`, указывающее на функцию-конструктор. В частности, `Rabbit.prototype.constructor == Rabbit`. Если мы рассчитываем использовать это свойство, то при замене `prototype` через `Object.create` нужно его явно сохранить:

```
Rabbit.prototype = Object.create(Animal.prototype);
Rabbit.prototype.constructor = Rabbit;
```

Полный код наследования

Для наглядности – вот итоговый код с двумя классами `Animal` и `Rabbit`:

```
// 1. Конструктор Animal
function Animal(name) {
  this.name = name;
  this.speed = 0;
}

// 1.1. Методы -- в прототип
Animal.prototype.stop = function() {
  this.speed = 0;
  alert( this.name + ' стоит' );
}

Animal.prototype.run = function(speed) {
  this.speed += speed;
  alert( this.name + ' бежит, скорость ' + this.speed );
};

// 2. Конструктор Rabbit
function Rabbit(name) {
  this.name = name;
  this.speed = 0;
}

// 2.1. Наследование
Rabbit.prototype = Object.create(Animal.prototype);
Rabbit.prototype.constructor = Rabbit;

// 2.2. Методы Rabbit
Rabbit.prototype.jump = function() {
  this.speed++;
  alert( this.name + ' прыгает, скорость ' + this.speed );
}
```

Как видно, наследование задаётся всего одной строчкой, поставленной в правильном месте.

Обратим внимание: `Rabbit.prototype = Object.create(Animal.prototype)` присваивается сразу после объявления конструктора, иначе он перезапишет уже записанные в прототип методы.

Неправильный вариант: `Rabbit.prototype = new Animal`

В некоторых устаревших руководствах предлагают вместо `Object.create(Animal.prototype)` записывать в прототип `new Animal`, вот так:

```
// вместо Rabbit.prototype = Object.create(Animal.prototype)
Rabbit.prototype = new Animal();
```

Частично, он рабочий, поскольку иерархия прототипов будет такая же, ведь `new Animal` – это объект с прототипом `Animal.prototype`, как и `Object.create(Animal.prototype)`. Они в этом плане идентичны.

Но у этого подхода важный недостаток. Как правило мы не хотим создавать `Animal`, а хотим только унаследовать его методы!

Более того, на практике создание объекта может требовать обязательных аргументов, влиять на страницу в браузере, делать запросы к серверу и что-то ещё, чего мы хотели бы избежать. Поэтому рекомендуется использовать вариант с `Object.create`.

Вызов конструктора родителя

Посмотрим внимательно на конструкторы `Animal` и `Rabbit` из примеров выше:

```
function Animal(name) {
  this.name = name;
  this.speed = 0;
}

function Rabbit(name) {
  this.name = name;
  this.speed = 0;
}
```

Как видно, объект `Rabbit` не добавляет никакой особенной логики при создании, которой не было в `Animal`.

Чтобы упростить поддержку кода, имеет смысл не дублировать код конструктора `Animal`, а напрямую вызвать его:

```
function Rabbit(name) {
  Animal.apply(this, arguments);
}
```

Такой вызов запустит функцию `Animal` в контексте текущего объекта, со всеми аргументами, она выполнится и запишет в `this` всё, что нужно.

Здесь можно было бы использовать и `Animal.call(this, name)`, но `apply` надёжнее, так как работает с любым количеством аргументов.

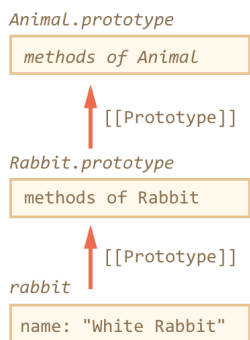
Переопределение метода

Итак, `Rabbit` наследует `Animal`. Теперь если какого-то метода нет в `Rabbit.prototype` – он будет взят из `Animal.prototype`.

В `Rabbit` может понадобиться задать какие-то методы, которые у родителя уже есть. Например, кролики бегают не так, как остальные животные, поэтому переопределим метод `run()`:

```
Rabbit.prototype.run = function(speed) {
  this.speed++;
  this.jump();
};
```

Вызов `rabbit.run()` теперь будет брать `run` из своего прототипа:



Вызов метода родителя внутри своего

Более частая ситуация – когда мы хотим не просто заменить метод на свой, а взять метод родителя и расширить его. Скажем, кролик бежит так же, как и другие звери, но время от времени подпрыгивает.

Для вызова метода родителя можно обратиться к нему напрямую, взяв из прототипа:

```
Rabbit.prototype.run = function() {
  // вызвать метод родителя, передав ему текущие аргументы
  Animal.prototype.run.apply(this, arguments);
  this.jump();
}
```

Обратите внимание на вызов через `apply` и явное указание контекста.

Если вызвать просто `Animal.prototype.run()`, то в качестве `this` функция `run` получит `Animal.prototype`, а это неверно, нужен текущий объект.

Итого

- Для наследования нужно, чтобы «склад методов потомка» (`Child.prototype`) наследовал от «склада метода родителей» (`Parent.prototype`).

Это можно сделать при помощи `Object.create`:

Код:

```
Rabbit.prototype = Object.create(Animal.prototype);
```

- Для того, чтобы наследник создавался так же, как и родитель, он вызывает конструктор родителя в своём контексте, используя `apply(this, arguments)`, вот так:

```
function Rabbit(...) {
  Animal.apply(this, arguments);
}
```

- При переопределении метода родителя в потомке, к исходному методу можно обратиться, взяв его напрямую из прототипа:

```
Rabbit.prototype.run = function() {
  var result = Animal.prototype.run.apply(this, ...);
  // result -- результат вызова метода родителя
}
```

Структура наследования полностью:

```
// ----- Класс-Родитель -----
// Конструктор родителя пишет свойства конкретного объекта
function Animal(name) {
  this.name = name;
  this.speed = 0;
}
```

```

// Методы хранятся в прототипе
Animal.prototype.run = function() {
  alert(this.name + " бежит!")
}

// ----- Класс-потомок -----
// Конструктор потомка
function Rabbit(name) {
  Animal.apply(this, arguments);
}

// Унаследовать
Rabbit.prototype = Object.create(Animal.prototype);

// Желательно и constructor сохранить
Rabbit.prototype.constructor = Rabbit;

// Методы потомка
Rabbit.prototype.run = function() {
  // Вызов метода родителя внутри своего
  Animal.prototype.run.apply(this);
  alert( this.name + " подпрыгивает!" );
};

// Готово, можно создавать объекты
var rabbit = new Rabbit('Кроль');
rabbit.run();

```

Такое наследование лучше функционального стиля, так как не дублирует методы в каждом объекте.

Кроме того, есть ещё неявное, но очень важное архитектурное отличие.

Зачастую вызов конструктора имеет какие-то побочные эффекты, например влияет на документ. Если конструктор родителя имеет какое-то поведение, которое нужно переопределить в потомке, то в функциональном стиле это невозможно.

Иначе говоря, в функциональном стиле в процессе создания `Rabbit` нужно обязательно вызывать `Animal.apply(this, arguments)`, чтобы получить методы родителя – и если этот `Animal.apply` кроме добавления методов говорит: «Му-у-у!», то это проблема:

```

function Animal() {
  this.walk = function() {
    alert('walk')
  };
  alert( 'Му-у-у!' );
}

function Rabbit() {
  Animal.apply(this, arguments); // как избавиться от мычания, но получить walk?
}

```

...Которой нет в прототипном подходе, потому что в процессе создания `new Rabbit` мы вовсе не обязаны вызывать конструктор родителя. Ведь методы находятся в прототипе.

Поэтому прототипный подход стоит предпочитать функциональному как более быстрый и универсальный. А что касается красоты синтаксиса – она сильно лучше в новом стандарте ES6, которым можно пользоваться уже сейчас, если взять транслятор [babeljs](#).

✓ Задачи

Найдите ошибку в наследовании

важность: 5

Найдите ошибку в прототипном наследовании. К чему она приведёт?

```

function Animal(name) {
  this.name = name;
}

Animal.prototype.walk = function() {
  alert( "ходит " + this.name );
};

function Rabbit(name) {
  this.name = name;
}
Rabbit.prototype = Animal.prototype;

Rabbit.prototype.walk = function() {
  alert( "прыгает! и ходит: " + this.name );
};

```

[К решению](#)

В чём ошибка в наследовании

важность: 5

Найдите ошибку в прототипном наследовании. К чему она приведёт?

```

function Animal(name) {
  this.name = name;
}

```

```
this.walk = function() {
  alert( "ходит " + this.name );
};
}

function Rabbit(name) {
  Animal.apply(this, arguments);
}
Rabbit.prototype = Object.create(Animal.prototype);

Rabbit.prototype.walk = function() {
  alert( "прыгает " + this.name );
};

var rabbit = new Rabbit("Кроль");
rabbit.walk();
```

[К решению](#)

Класс "часы"

важность: 5

Есть реализация часиков, оформленная в виде одной функции-конструктора. У неё есть приватные свойства `timer`, `template` и метод `render`.

Задача: переписать часы на прототипах. Приватные свойства и методы сделать защищёнными.

P.S. Часики тикают в браузерной консоли (надо открыть её, чтобы увидеть).

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Класс "расширенные часы"

важность: 5

Есть реализация часиков на прототипах. Создайте класс, расширяющий её, добавляющий поддержку параметра `precision`, который будет задавать частоту тика в `setInterval`. Значение по умолчанию: `1000`.

- Для этого класс `Clock` надо унаследовать. Пишите ваш новый код в файле `extended-clock.js`.
- Исходный класс `Clock` менять нельзя.
- Пусть конструктор потомка вызывает конструктор родителя. Это позволит избежать проблем при расширении `Clock` новыми опциями.

P.S. Часики тикают в браузерной консоли (надо открыть её, чтобы увидеть).

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Меню с таймером для анимации

важность: 5

Есть класс `Menu`. У него может быть два состояния: открыто `STATE_OPEN` и закрыто `STATE_CLOSED`.

Создайте наследника `AnimatingMenu`, который добавляет третье состояние `STATE_ANIMATING`.

- При вызове `open()` состояние меняется на `STATE_ANIMATING`, а через 1 секунду, по таймеру, открытие завершается вызовом `open()` родителя.
- Вызов `close()` при необходимости отменяет таймер анимации (назначаемый в `open`) и передаёт вызов родительскому `close`.
- Метод `showState` для нового состояния выводит "анимация", для остальных – полагается на родителя.

[Исходный документ, вместе с тестом](#) ↗

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Что содержит constructor?

важность: 5

В коде ниже создаётся простейшая иерархия классов: `Animal` -> `Rabbit`.

Что содержит свойство `rabbit.constructor`? Распознает ли проверка в `alert` объект как `Rabbit`?

```
function Animal() {}

function Rabbit() {}
```

```
Rabbit.prototype = Object.create(Animal.prototype);
var rabbit = new Rabbit();
alert( rabbit.constructor == Rabbit ); // что выведет?
```

[К решению](#)

Проверка класса: "instanceof"

Оператор `instanceof` позволяет проверить, какому классу принадлежит объект, с учетом прототипного наследования.

Алгоритм работы instanceof

Вызов `obj instanceof Constructor` возвращает `true`, если объект принадлежит классу `Constructor` или классу, наследующему от него.

Пример использования:

```
function Rabbit() {}
// создаём объект
var rabbit = new Rabbit();
// проверяем -- этот объект создан Rabbit?
alert( rabbit instanceof Rabbit ); // true, верно
```

Массив `arr` принадлежит классу `Array`, но также и является объектом `Object`. Это верно, так как массивы наследуют от объектов:

```
var arr = [];
alert( arr instanceof Array ); // true
alert( arr instanceof Object ); // true
```

Как это часто бывает в JavaScript, здесь есть ряд тонкостей. Проверка происходит через сравнение прототипов, поэтому в некоторых ситуациях может даже ошибаться!

Алгоритм проверки `obj instanceof Constructor`:

1. Получить `obj.__proto__`
2. Сравнить `obj.__proto__` с `Constructor.prototype`
3. Если не совпадает, тогда заменить `obj` на `obj.__proto__` и повторить проверку на шаге 2 до тех пор, пока либо не найдется совпадение (результат `true`), либо цепочка прототипов не закончится (результат `false`).

В проверке `rabbit instanceof Rabbit` совпадение происходит на первом же шаге этого алгоритма, так как: `rabbit.__proto__ == Rabbit.prototype`.

А если рассмотреть `arr instanceof Object`, то совпадение будет найдено на следующем шаге, так как `arr.__proto__.__proto__ == Object.prototype`.

Забавно, что сама функция-конструктор не участвует в процессе проверки! Важна только цепочка прототипов для проверяемого объекта.

Это может приводить к забавному результату и даже ошибкам в проверке при изменении `prototype`, например:

```
// Создаём объект rabbit, как обычно
function Rabbit() {}
var rabbit = new Rabbit();

// изменили prototype...
Rabbit.prototype = {};

// ...instanceof перестал работать!
alert( rabbit instanceof Rabbit ); // false
```

Стоит ли говорить, что это один из доводов для того, чтобы никогда не менять `prototype`? Так сказать, во избежание.

Не друзья: `instanceof` и фреймы

Оператор `instanceof` не срабатывает, когда значение приходит из другого окна или фрейма.

Например, массив, который создан в ифрейме и передан родительскому окну – будет массивом *в том ифрейме*, но не в родительском окне. Проверка `instanceof Array` в родительском окне вернёт `false`.

Вообще, у каждого окна и фрейма – своя иерархия объектов и свой `window`.

Как правило, эта проблема возникает со встроенными объектами, в этом случае используется проверка внутреннего свойства `[[Class]]`, которое подробнее описано в главе [Типы данных: \[\[Class\]\], instanceof и утки](#).

Итого

- Оператор `obj instanceof Func` проверяет тот факт, что `obj` является результатом вызова `new Func`. Он учитывает цепочку `__proto__`, поэтому наследование поддерживается.
- Оператор `instanceof` не сможет проверить тип значения, если объект создан в одном окне/фрейме, а проверяется в другом. Это потому, что в каждом окне – своя иерархия объектов. Для точной проверки типов встроенных объектов можно использовать свойство `[[Class]]`.

Оператор `instanceof` особенно востребован в случаях, когда мы работаем с иерархиями классов. Это наилучший способ проверить принадлежность тому или иному классу с учётом наследования.

✔ Задачи

Странное поведение instanceof

важность: 5

Почему `instanceof` в коде ниже возвращает `true`, ведь объект `a` явно создан не `B()`?

```
function A() {}
function B() {}
A.prototype = B.prototype = {};
var a = new A();
alert( a instanceof B ); // true
```

[К решению](#)

Что выведет instanceof?

важность: 5

В коде ниже создаётся простейшая иерархия классов: `Animal` -> `Rabbit`.

Что выведет `instanceof`?

Распознает ли он `rabbit` как `Animal`, `Rabbit` и к тому же `Object`?

```
function Animal() {}
function Rabbit() {}
Rabbit.prototype = Object.create(Animal.prototype);
var rabbit = new Rabbit();
alert( rabbit instanceof Rabbit );
alert( rabbit instanceof Animal );
alert( rabbit instanceof Object );
```

[К решению](#)

Свои ошибки, наследование от Error

Когда мы работаем с внешними данными, возможны самые разные ошибки.

Если приложение сложное, то ошибки естественным образом укладываются в иерархию, разобраться в которой помогает `instanceof`.

Свой объект ошибки

Для примера создадим функцию `readUser(json)`, которая будет разбирать JSON с данными посетителя. Мы его получаем с сервера – может, нашего, а может – чужого, в общем – желательно проверить на ошибки. А может, это даже и не JSON, а какие-то другие данные – не важно, для наглядности поработаем с JSON.

Пример `json` на входе в функцию: `{ "name": "Вася", "age": 30 }`.

В процессе работы `readUser` возможны различные ошибки. Одна – очевидно, `SyntaxError` – если передан некорректный JSON.

Но могут быть и другие, например `PropertyError` – эта ошибка будет возникать, если в прочитанном объекте нет свойства `name` или `age`.

Реализуем класс `PropertyError`:

```
function PropertyError(property) {
  Error.call(this, property);
  this.name = "PropertyError";

  this.property = property;
  this.message = "Ошибка в свойстве " + property;

  if (Error.captureStackTrace) {
    Error.captureStackTrace(this, PropertyError);
  }
}
```



```
    } else {
      this.stack = (new Error()).stack;
    }
  }
}

PropertyError.prototype = Object.create(Error.prototype);
```

В этом коде вы можете видеть ряд важных деталей, важных именно для ошибок:

name – имя ошибки.

Должно совпадать с именем функции.

message – сообщение об ошибке.

Несмотря на то, что `PropertyError` наследует от `Error` (последняя строка), конструктор у неё немного другой. Он принимает не сообщение об ошибке, а название свойства `property`, ну а сообщение генерируется из него.

В результате в объекте ошибки есть как стандартное свойство `message`, так и более точное `property`.

Это частая практика – добавлять в объект ошибки свойства, которых нет в базовых объектах `Error`, более подробно описывающие ситуацию для данного класса ошибок.

stack – стек вызовов, которые в итоге привели к ошибке.

У встроенных объектов `Error` это свойство есть автоматически, вот к примеру:

```
function f() {
  alert( new Error().stack );
}

f(); // выведет список вложенных вызовов, с номерами строк, где они были сделаны
```

Если же объект ошибки делаем мы, то «по умолчанию» такого свойства у него не будет. Нам нужно как-то самим узнавать последовательность вложенных вызовов на текущий момент. Однако удобного способа сделать это в JavaScript нет, поэтому мы поступаем хитро и копируем его из нового объекта `new Error`, который генерируем тут же.

В V8 (Chrome, Opera, Node.JS) есть нестандартное расширение [Error.captureStackTrace](#), которое позволяет получить стек.

Это делает строка из кода выше:

```
Error.captureStackTrace(this, PropertyError);
```

Такой вызов записывает в объект `this` (текущий объект ошибки) стек вызовов, ну а второй аргумент – вообще не обязателен, но если есть, то говорит, что при генерации стека нужно на этой функции остановиться. В результате в стеке будет информация о цепочке вложенных вызовов вплоть до вызова `PropertyError`.

То есть, будет последовательность вызовов до генерации ошибки, но не включая код самого конструктора ошибки, который, как правило, не интересен. Такое поведение максимально соответствует встроенным ошибкам JavaScript.

i Конструктор родителя здесь не обязателен

Обычно, когда мы наследуем, то вызываем конструктор родителя. В данном случае вызов выглядит как `Error.call(this, message)`.

Строго говоря, этот вызов здесь не обязателен. Встроенный конструктор `Error` ничего полезного не делает, даже свойство `this.message` (не говоря уже об `name` и `stack`) не назначает. Единственный возможный смысл его вызова – он ставит специальное внутреннее свойство `[[ErrorData]]`, которое выводится в `toString` и позволяет увидеть, что это ошибка. Поэтому по стандарту вызывать конструктор `Error` при наследовании в таких случаях рекомендовано.

`instanceof + try...catch` = ♥

Давайте теперь используем наш новый класс для `readUser`:

```
// Объявление
function PropertyError(property) {
  this.name = "PropertyError";

  this.property = property;
  this.message = "Ошибка в свойстве " + property;

  if (Error.captureStackTrace) {
    Error.captureStackTrace(this, PropertyError);
  } else {
    this.stack = (new Error()).stack;
  }
}

PropertyError.prototype = Object.create(Error.prototype);

// Генерация ошибки
function readUser(data) {

  var user = JSON.parse(data);
```

```

if (!user.age) {
  throw new PropertyError("age");
}

if (!user.name) {
  throw new PropertyError("name");
}

return user;
}

```

// Запуск и try..catch

```

try {
  var user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof PropertyError) {
    if (err.property == 'name') {
      // если в данном месте кода возможны анонимы, то всё нормально
      alert( "Здравствуйте, Аноним!" );
    } else {
      alert( err.message ); // Ошибка в свойстве ...
    }
  } else if (err instanceof SyntaxError) {
    alert( "Ошибка в синтаксисе данных: " + err.message );
  } else {
    throw err; // неизвестная ошибка, не знаю что с ней делать
  }
}

```

Всё работает – и наша ошибка `PropertyError` и встроенная `SyntaxError` корректно генерируются, перехватываются, обрабатываются.

Обратим внимание на проверку типа ошибки в `try..catch`. Оператор `instanceof` проверяет класс с учётом наследования. Это значит, что если мы в дальнейшем решим создать новый тип ошибки, наследующий от `PropertyError`, то проверка `err instanceof PropertyError` для класса-наследника тоже будет работать. Код получился расширяемым, это очень важно.

Дальнейшее наследование

`PropertyError` – это просто общего вида ошибка в свойстве. Создадим ошибку `PropertyRequiredError`, которая означает, что свойства нет.

Это подвид `PropertyError`, так что унаследуем от неё. Общий вид конструктора-наследника – стандартный:

```

function PropertyRequiredError(property) {
  // вызываем конструктор родителя и передаём текущие аргументы
  PropertyError.apply(this, arguments);
  ...
}

```

Достаточно ли в наследнике просто вызвать конструктор родителя? Увы, нет.

Если так поступить, то свойство `this.name` будет некорректным, да и `Error.captureStackTrace` тоже получит неправильную функцию вторым параметром.

Можно ли как-то поправить конструктор родителя, чтобы от него было проще наследовать?

Для этого нужно убрать из него упоминания о конкретном классе `PropertyError`, чтобы сделать код универсальным. Частично – это возможно. Как мы помним, существует свойство `constructor`, которое есть в `prototype` по умолчанию, и которое мы можем намеренно сохранить при наследовании.

Исправим родителя `PropertyError` для более удобного наследования от него:

```

function PropertyError(property) {
  this.name = "PropertyError";

  this.property = property;
  this.message = "Ошибка в свойстве " + property;

  if (Error.captureStackTrace) {
    Error.captureStackTrace(this, this.constructor); // (*)
  } else {
    this.stack = (new Error()).stack;
  }
}

```

```

PropertyError.prototype = Object.create(Error.prototype);
PropertyError.prototype.constructor = PropertyError;

```

В строке (*) вместо ссылки на `PropertyError` используем `constructor` чтобы получить именно конструктор для текущего объекта. В наследнике там будет `PropertyRequiredError`, как и задумано.

Мы убрали одну жёсткую привязку к `PropertyError`, но со второй (`this.name`), увы, сложности. Оно должно содержать имя ошибки, то есть, имя её функции-конструктора. Его можно получить через `this.name = this.constructor.name`, но в IE11- это работать не будет.

Если поддерживать IE11-, то тут уж придётся в наследнике его записывать вручную.

Полный код для наследника:

```

function PropertyRequiredError(property) {
  PropertyError.apply(this, arguments);
  this.name = 'PropertyRequiredError';
  this.message = 'Отсутствует свойство ' + property;
}

```

```
PropertyRequiredError.prototype = Object.create(PropertyError.prototype);
PropertyRequiredError.prototype.constructor = PropertyRequiredError;
```

```
var err = new PropertyRequiredError("age");
// пройдёт проверку
alert( err instanceof PropertyError ); // true
```

Здесь заодно и `message` в наследнике было перезаписано на более точное. Если хочется избежать записи и перезаписи, то можно оформить его в виде геттера через `Object.defineProperty`.

Итого

- Чтобы наследовать от ошибок `Error`, нужно самостоятельно позаботиться о `name`, `message` и `stack`.
- Благодаря тому, что `instanceof` поддерживает наследование, удобно организуются проверки на нужный тип. В иерархию ошибок можно в любой момент добавить новые классы, с понятным кодом и предсказуемым поведением.

Чтобы создавать наследники от `Error` было проще, можно создать класс `CustomError`, записать в него универсальный код, наподобие `PropertyError` и далее наследовать уже от него:

```
// общего вида "наша" ошибка
function CustomError(message) {
  this.name = "CustomError";
  this.message = message;

  if (Error.captureStackTrace) {
    Error.captureStackTrace(this, this.constructor);
  } else {
    this.stack = (new Error()).stack;
  }
}
```

```
CustomError.prototype = Object.create(Error.prototype);
CustomError.prototype.constructor = CustomError;
```

```
// наследник
function PropertyError(property) {
  CustomError.call(this, "Отсутствует свойство " + property);
  this.name = "PropertyError";

  this.property = property;
}
```

```
PropertyError.prototype = Object.create(CustomError.prototype);
PropertyError.prototype.constructor = PropertyError;
```

```
// и ещё уровень
function PropertyRequiredError(property) {
  PropertyError.call(this, property);
  this.name = 'PropertyRequiredError';
  this.message = 'Отсутствует свойство ' + property;
}
```

```
PropertyRequiredError.prototype = Object.create(PropertyError.prototype);
PropertyRequiredError.prototype.constructor = PropertyRequiredError;
```

```
// использование
var err = new PropertyRequiredError("age");
// пройдёт проверку
alert( err instanceof PropertyRequiredError ); // true
alert( err instanceof PropertyError ); // true
alert( err instanceof CustomError ); // true
alert( err instanceof Error ); // true
```

✔ Задачи

Унаследуйте от `SyntaxError`

важность: 5

Создайте ошибку `FormatError`, которая будет наследовать от встроенного класса `SyntaxError`.

Синтаксис для её создания – такой же, как обычно:

```
var err = new FormatError("ошибка форматирования");

alert( err.message ); // ошибка форматирования
alert( err.name ); // FormatError
alert( err.stack ); // стек на момент генерации ошибки

alert( err instanceof SyntaxError ); // true
```

[К решению](#)

Примеси

В JavaScript невозможно унаследовать от двух и более объектов. Ссылка `__proto__` – только одна.

Но потребность такая существует – к примеру, мы написали код, реализующий методы работы с шаблонизатором или методы по обмену событиями, и хочется легко и непринуждённо добавлять эти возможности к любому классу.

Обычно это делают через примеси.

Примесь (англ. *mixin*) – класс или объект, реализующий какое-либо чётко выделенное поведение. Используется для уточнения поведения других классов, не предназначен для самостоятельного использования.

Пример примеси

Самый простой вариант примеси – это объект с полезными методами, которые мы просто копируем в нужный прототип.

Например:

```
// примесь
var sayHiMixin = {
  sayHi: function() {
    alert("Привет " + this.name);
  },
  sayBye: function() {
    alert("Пока " + this.name);
  }
};

// использование:
function User(name) {
  this.name = name;
}

// передать методы примеси
for(var key in sayHiMixin) User.prototype[key] = sayHiMixin[key];

// User "умеет" sayHi
new User("Вася").sayHi(); // Привет Вася
```

Как видно из примера, методы примеси активно используют `this` и предназначены именно для запуска в контексте «объекта-носителя примеси».

Если какие-то из методов примеси не нужны – их можно перезаписать своими после копирования.

Примесь для событий

Теперь пример из реальной жизни.

Важный аспект, который может понадобиться объектам – это умение работать с событиями.

То есть, чтобы объект мог специальным вызовом генерировать «уведомление о событии», а на эти уведомления другие объекты могли «подписываться», чтобы их получать.

Например, объект «Пользователь» при входе на сайт может генерировать событие "login", а другие объекты, например «Календарь» может такие уведомления получать и подгружать информацию о пользователе.

Или объект «Меню» может при выборе пункта меню генерировать событие "select" с информацией о выбранном пункте меню, а другие объекты – подписавшись на это событие, будут узнавать об этом.

События – это средство «поделиться информацией» с неопределённым кругом заинтересованных лиц. А там уже кому надо – тот среагирует.

Примесь `eventMixin`, реализующая события:

```
var eventMixin = {

  /**
   * Подписка на событие
   * Использование:
   * menu.on('select', function(item) { ... })
   */
  on: function(eventName, handler) {
    if (!this._eventHandlers) this._eventHandlers = {};
    if (!this._eventHandlers[eventName]) {
      this._eventHandlers[eventName] = [];
    }
    this._eventHandlers[eventName].push(handler);
  },

  /**
   * Прекращение подписки
   * menu.off('select', handler)
   */
  off: function(eventName, handler) {
    var handlers = this._eventHandlers && this._eventHandlers[eventName];
    if (!handlers) return;
    for(var i=0; i<handlers.length; i++) {
      if (handlers[i] == handler) {
        handlers.splice(i--, 1);
      }
    }
  },

  /**
   * Генерация события с передачей данных
   * this.trigger('select', item);
   */
  trigger: function(eventName /*, ... */) {
    if (!this._eventHandlers || !this._eventHandlers[eventName]) {
      return; // обработчиков для события нет
    }
  }
};
```

```

// вызвать обработчики
var handlers = this._eventHandlers[eventName];
for (var i = 0; i < handlers.length; i++) {
  handlers[i].apply(this, [].slice.call(arguments, 1));
}
}
};

```

Здесь есть три метода:

1. `.on(имя события, функция)` – назначает функцию к выполнению при наступлении события с данным именем. Такие функции хранятся в защищённом свойстве объекта `_eventHandlers`.
2. `.off(имя события, функция)` – удаляет функцию из списка предназначенных к выполнению.
3. `.trigger(имя события, аргументы)` – генерирует событие, при этом вызываются все назначенные на него функции, и им передаются аргументы.

Использование:

```

// Класс Menu с примесью eventMixin
function Menu() {
  // ...
}

for(var key in eventMixin) {
  Menu.prototype[key] = eventMixin[key];
}

// Генерирует событие select при выборе значения
Menu.prototype.choose = function(value) {
  this.trigger("select", value);
}

// Создадим меню
var menu = new Menu();

// При наступлении события select вызвать эту функцию
menu.on("select", function(value) {
  alert("Выбрано значение " + value);
});

// Запускаем выбор (событие select вызовет обработчики)
menu.choose("123");

```

...То есть, смысл событий – обычно в том, что объект, в процессе своей деятельности, внутри себя (`this.trigger`) генерирует уведомления, на которые внешний код через `menu.on(...)` может быть подписан. И узнавать из них ценную информацию о происходящем, например – что выбран некий пункт меню.

Один раз написав методы `on/off/trigger` в примеси, мы затем можем использовать их во множестве прототипов.

Итого

- Примесь – объект, содержащий методы и свойства для реализации конкретного функционала. Возможны вариации этого приёма проектирования. Например, примесь может предусматривать конструктор, который должен запускаться в конструкторе объекта. Но как правило просто набора методов хватает.
- Для добавления примеси в класс – её просто «подмешивают» в прототип.
- «Подмешать» можно сколько угодно примесей, но если имена методов в разных примесях совпадают, то возможны конфликты. Их уже разрешать – разработчику. Например, можно заменить конфликтующий метод на свой, который будет решать несколько задач сразу. Конфликты при грамотно оформленных примесях возникают редко.

Современные возможности ES-2015

Современный стандарт ES-2015 и его расширения для JavaScript.

ES-2015 сейчас

Стандарт ES-2015 [↗](https://kangax.github.io/compat-table/es6/) был принят в июне 2015. Пока что большинство браузеров реализуют его частично, текущее состояние реализации различных возможностей можно посмотреть здесь: <https://kangax.github.io/compat-table/es6/> [↗](#).

Когда стандарт будет более-менее поддерживаться во всех браузерах, то весь учебник будет обновлён в соответствии с ним. Пока же, как центральное место для «сбора» современных фиш JavaScript, создан этот раздел.

Чтобы писать код на ES-2015 прямо сейчас, есть следующие варианты.

Конкретный движок JS

Самое простое – это когда нужен один конкретный движок JS, например V8 (Chrome).

Тогда можно использовать только то, что поддерживается именно в нём. Заметим, что в V8 большинство возможностей ES-2015 поддерживаются только при включённом `use strict`.

При разработке на Node.JS обычно так и делают. Если же нужна кросс-браузерная поддержка, то этот вариант не подойдёт.

Babel.JS

[Babel.JS](#) – это [транспайлер](#), переписывающий код на ES-2015 в код на предыдущем стандарте ES5.

Он состоит из двух частей:

1. Собственно транспайлер, который переписывает код.
2. [Полифилл](#), который добавляет методы `Array.from`, `String.prototype.repeat` и другие.

На странице <https://babeljs.io/repl/> можно поэкспериментировать с транспайлером: слева вводится код в ES-2015, а справа появляется результат его преобразования в ES5.

Обычно Babel.JS работает на сервере в составе системы сборки JS-кода (например [webpack](#) или [brunch](#)) и автоматически переписывает весь код в ES5.

Настройка такой конвертации тривиальна, единственно – нужно поднять саму систему сборки, а добавить к ней Babel легко, плагины есть к любой из них.

Если же хочется «поиграться», то можно использовать и браузерный вариант Babel.

Это выглядит так:

```
<!-- browser.js лежит на моём сервере, не надо брать с него -->
<script src="https://js.cx/babel-core/browser.min.js"></script>

<script type="text/babel">
  let arr = ["hello", 2]; // let

  let [str, times] = arr; // деструктуризация

  alert( str.repeat(times) ); // hellohello, метод repeat
</script>
```

Сверху подключается браузерный скрипт `browser.min.js` из пакета Babel. Он включает в себя полифилл и транспайлер. Далее он автоматически транслирует и выполняет скрипты с `type="text/babel"`.

Размер `browser.min.js` превышает 1 мегабайт, поэтому такое использование в production строго не рекомендуется.

Примеры на этом сайте

Только при поддержке браузера

Запускаемые примеры с ES-2015 будут работать только если ваш браузер поддерживает соответствующую возможность стандарта.

Это означает, что при запуске примеров в браузере, который их не поддерживает, будет ошибка. Это не означает, что пример неправильный! Просто пока нет поддержки...

Рекомендуется [Chrome Canary](#), большинство примеров в нём работает. [Firefox Developer Edition](#) тоже неплох в поддержке современного стандарта, но на момент написания этого текста переменные `let` работают только при указании `version=1.7` в типе скрипта: `<script type="application/javascript;version=1.7">`. Надеюсь, скоро это требование (`version=1.7`) отменят.

Впрочем, если пример в браузере не работает (обычно проявляется как ошибка синтаксиса) – почти все примеры вы можете запустить при помощи Babel, на странице [Babel: try it out](#). Там же увидите и преобразованный код.

На практике для кросс-браузерности всё равно используют Babel.

Ещё раз заметим, что самая актуальная ситуация по поддержке современного стандарта браузерами и транспайлерами отражена на странице <https://kangax.github.io/compat-table/es6/>.

Итак, поехали!

Переменные: `let` и `const`

В ES-2015 предусмотрены новые способы объявления переменных: через `let` и `const` вместо `var`.

Например:

```
let a = 5;
```

`let`

У объявлений переменной через `let` есть три основных отличия от `var`:

1. Область видимости переменной `let` – блок `{...}`.

Как мы помним, переменная, объявленная через `var`, видна везде в функции.

Переменная, объявленная через `let`, видна только в рамках блока `{...}`, в котором объявлена.

Это, в частности, влияет на объявления внутри `if`, `while` или `for`.

Например, переменная через `var` :

```
var apples = 5;
if (true) {
  var apples = 10;
  alert(apples); // 10 (внутри блока)
}
alert(apples); // 10 (снаружи блока то же самое)
```

В примере выше `apples` – одна переменная на весь код, которая модифицируется в `if` .

То же самое с `let` будет работать по-другому:

```
let apples = 5; // (*)
if (true) {
  let apples = 10;
  alert(apples); // 10 (внутри блока)
}
alert(apples); // 5 (снаружи блока значение не изменилось)
```

Здесь, фактически, две независимые переменные `apples` , одна – глобальная, вторая – в блоке `if` .

Заметим, что если объявление `let apples` в первой строке (*) удалить, то в последнем `alert` будет ошибка: переменная неопределена:

```
if (true) {
  let apples = 10;
  alert(apples); // 10 (внутри блока)
}
alert(apples); // ошибка!
```

Это потому что переменная `let` всегда видна именно в том блоке, где объявлена, и не более.

2. Переменная `let` видна только после объявления.

Как мы помним, переменные `var` существуют и до объявления. Они равны `undefined` :

```
alert(a); // undefined
var a = 5;
```

С переменными `let` всё проще. До объявления их вообще нет.

Такой доступ приведёт к ошибке:

```
alert(a); // ошибка, нет такой переменной
let a = 5;
```

Заметим также, что переменные `let` нельзя повторно объявлять. То есть, такой код выведет ошибку:

```
let x;
let x; // ошибка: переменная x уже объявлена
```

Это – хоть и выглядит ограничением по сравнению с `var` , но на самом деле проблем не создаёт. Например, два таких цикла совсем не конфликтуют:

```
// каждый цикл имеет свою переменную i
for(let i = 0; i<10; i++) { /* ... */ }
for(let i = 0; i<10; i++) { /* ... */ }
alert(i); // ошибка: глобальной i нет
```

При объявлении внутри цикла переменная `i` будет видна только в блоке цикла. Она не видна снаружи, поэтому будет ошибка в последнем `alert` .

3. При использовании в цикле, для каждой итерации создаётся своя переменная.

Переменная `var` – одна на все итерации цикла и видна даже после цикла:

```
for(var i=0; i<10; i++) { /* ... */ }
alert(i); // 10
```

С переменной `let` – всё по-другому.

Каждому повторению цикла соответствует своя независимая переменная `let`. Если внутри цикла есть вложенные объявления функций, то в замыкании каждой будет та переменная, которая была при соответствующей итерации.

Это позволяет легко решить классическую проблему с замыканиями, описанную в задаче [Армия функций](#).

```
function makeArmy() {
  let shooters = [];

  for (let i = 0; i < 10; i++) {
    shooters.push(function() {
      alert( i ); // выводит свой номер
    });
  }

  return shooters;
}

var army = makeArmy();

army[0](); // 0
army[5](); // 5
```

Если бы объявление было `var i`, то была бы одна переменная `i` на всю функцию, и вызовы в последних строках выводили бы 10 (подробнее – см. задачу [Армия функций](#)).

А выше объявление `let i` создаёт для каждого повторения блока в цикле свою переменную, которую функция и получает из замыкания в последних строках.

const

Объявление `const` задаёт константу, то есть переменную, которую нельзя менять:

```
const apple = 5;
apple = 10; // ошибка
```

В остальном объявление `const` полностью аналогично `let`.

Заметим, что если в константу присвоен объект, то от изменения защищена сама константа, но не свойства внутри неё:

```
const user = {
  name: "Вася"
};

user.name = "Петя"; // допустимо
user = 5; // нельзя, будет ошибка
```

То же самое верно, если константе присвоен массив или другое объектное значение.

константы и КОНСТАНТЫ

Константы, которые жёстко заданы всегда, во время всей программы, обычно пишутся в верхнем регистре. Например: `const ORANGE = "#ffa500"`.

Большинство переменных – константы в другом смысле: они не меняются после присвоения. Но при разных запусках функции это значение может быть разным. Для таких переменных можно использовать `const` и обычные строчные буквы в имени.

Итого

Переменные `let`:

- Видны только после объявления и только в текущем блоке.
- Нельзя переобъявлять (в том же блоке).
- При объявлении переменной в цикле `for(let ...)` – она видна только в этом цикле. Причём каждой итерации соответствует своя переменная `let`.

Переменная `const` – это константа, в остальном – как `let`.

Деструктуризация

Деструктуризация (destructuring assignment) – это особый синтаксис присваивания, при котором можно присвоить массив или объект сразу нескольким переменным, разбив его на части.

Массив

Пример деструктуризации массива:

```
'use strict';

let [firstName, lastName] = ["Илья", "Кантор"];
```



```
alert(firstName); // Илья
alert(lastName); // Кантор
```

При таком присвоении первое значение массива пойдёт в переменную `firstName`, второе – в `lastName`, а последующие (если есть) – будут отброшены.

Ненужные элементы массива также можно отбросить, поставив лишнюю запятую:

```
'use strict';

// первый и второй элементы не нужны
let [, , title] = "Юлий Цезарь Император Рима".split(" ");

alert(title); // Император
```

В коде выше первый и второй элементы массива никуда не записались, они были отброшены. Как, впрочем, и все элементы после третьего.

Оператор «spread»

Если мы хотим получить и последующие значения массива, но не уверены в их числе – можно добавить ещё один параметр, который получит «всё остальное», при помощи оператора `"..."` («spread», троеточие):

```
'use strict';

let [firstName, lastName, ...rest] = "Юлий Цезарь Император Рима".split(" ");

alert(firstName); // Юлий
alert(lastName); // Цезарь
alert(rest); // Император,Рима (массив из 2х элементов)
```

Значением `rest` будет массив из оставшихся элементов массива. Вместо `rest` можно использовать и другое имя переменной, оператор здесь – троеточие. Оно должно стоять только последним элементом в списке слева.

Значения по умолчанию

Если значений в массиве меньше, чем переменных – ошибки не будет, просто присвоится `undefined`:

```
'use strict';

let [firstName, lastName] = [];

alert(firstName); // undefined
```

Впрочем, как правило, в таких случаях задают значение по умолчанию. Для этого нужно после переменной использовать символ `=` со значением, например:

```
'use strict';

// значения по умолчанию
let [firstName="Гость", lastName="Анонимный"] = [];

alert(firstName); // Гость
alert(lastName); // Анонимный
```

В качестве значений по умолчанию можно использовать не только примитивы, но и выражения, даже включающие в себя вызовы функций:

```
'use strict';

function defaultLastName() {
  return Date.now() + '-visitor';
}

// lastName получит значение, соответствующее текущей дате:
let [firstName, lastName=defaultLastName()] = ["Вася"];

alert(firstName); // Вася
alert(lastName); // 1436...-visitor
```

Заметим, что вызов функции `defaultLastName()` для генерации значения по умолчанию будет осуществлён только при необходимости, то есть если значения нет в массиве.

Деструктуризация объекта

Деструктуризацию можно использовать и с объектами. При этом мы указываем, какие свойства в какие переменные должны «идти».

Базовый синтаксис:

```
let {var1, var2} = {var1:..., var2:...}
```

Объект справа – уже существующий, который мы хотим разбить на переменные. А слева – список переменных, в которые нужно соответствующие свойства записать.

Например:

```
'use strict';  
  
let options = {  
  title: "Меню",  
  width: 100,  
  height: 200  
};
```

```
let {title, width, height} = options;
```

```
alert(title); // Меню  
alert(width); // 100  
alert(height); // 200
```

Как видно, свойства `options.title`, `options.width` и `options.height` автоматически присвоились соответствующим переменным.

Если хочется присвоить свойство объекта в переменную с другим именем, например, чтобы свойство `options.width` пошло в переменную `w`, то можно указать соответствие через двоеточие, вот так:

```
'use strict';  
  
let options = {  
  title: "Меню",  
  width: 100,  
  height: 200  
};
```

```
let {width: w, height: h, title} = options;
```

```
alert(title); // Меню  
alert(w); // 100  
alert(h); // 200
```

В примере выше свойство `width` отправилось в переменную `w`, свойство `height` – в переменную `h`, а `title` – в переменную с тем же названием.

Если каких-то свойств в объекте нет, можно указать значение по умолчанию через знак равенства `=`, вот так:

```
'use strict';  
  
let options = {  
  title: "Меню"  
};
```

```
let {width=100, height=200, title} = options;
```

```
alert(title); // Меню  
alert(width); // 100  
alert(height); // 200
```

Можно и сочетать одновременно двоеточие и равенство:

```
'use strict';  
  
let options = {  
  title: "Меню"  
};
```

```
let {width:w=100, height:h=200, title} = options;
```

```
alert(title); // Меню  
alert(w); // 100  
alert(h); // 200
```

А что, если в объекте больше значений, чем переменных? Можно ли куда-то присвоить «остаток», аналогично массивам?

Такой возможности в текущем стандарте нет. Она планируется в будущем стандарте, и выглядеть она будет примерно так:

```
'use strict';  
  
let options = {  
  title: "Меню",  
  width: 100,  
  height: 200  
};
```

```
let {title, ...size} = options;
```

```
// title = "Меню"  
// size = { width: 100, height: 200 } (остаток)
```

Этот код будет работать, например, при использовании Babel со включёнными экспериментальными возможностями, но ещё раз заметим, что в текущий стандарт такая возможность не вошла.

i Деструктуризация без объявления

В примерах выше переменные объявлялись прямо перед присваиванием: `let {...} = {...}`. Конечно, можно и без `let`, использовать уже существующие переменные.

Однако, здесь есть небольшой «подвох». В JavaScript, если в основном потоке кода (не внутри другого выражения) встречается `{...}`, то это воспринимается как блок.

Например, можно использовать такой блок для ограничения видимости переменных:

```
'use strict';
{
  // вспомогательные переменные, локальные для блока
  let a = 5;
  // поработали с ними
  alert(a); // 5
  // больше эти переменные не нужны
}
alert(a); // ошибка нет такой переменной
```

Конечно, это бывает удобно, но в данном случае это создаст проблему при деструктуризации:

```
let a, b;
{a, b} = {a:5, b:6}; // будет ошибка, оно посчитает, что {a,b} - блок
```

Чтобы избежать интерпретации `{a, b}` как блока, нужно обернуть всё присваивание в скобки:

```
let a, b;
({a, b} = {a:5, b:6}); // внутри выражения это уже не блок
```

Вложенные деструктуризации

Если объект или массив содержат другие объекты или массивы, и их тоже хочется разбить на переменные – не проблема.

Деструктуризации можно как угодно сочетать и вкладывать друг в друга.

В коде ниже `options` содержит подобъект и подмассив. В деструктуризации ниже сохраняется та же структура:

```
'use strict';

let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Пончик", "Пирожное"]
}

let { title="Меню", size: {width, height}, items: [item1, item2] } = options;

// Меню 100 200 Пончик Пирожное
alert(title); // Меню
alert(width); // 100
alert(height); // 200
alert(item1); // Пончик
alert(item2); // Пирожное
```

Как видно, весь объект `options` корректно разбит на переменные.

Итого

- Деструктуризация позволяет разбивать объект или массив на переменные при присвоении.
- Синтаксис:

```
let {prop : varName = default, ...} = object
```

Здесь двоеточие `:` задаёт отображение свойства `prop` в переменную `varName`, а равенство `=default` задаёт выражение, которое будет использовано, если значение отсутствует (не указано или `undefined`).

Для массивов имеет значение порядок, поэтому нельзя использовать `:`, но значение по умолчанию – можно:

```
let [var1 = default, var2, ...rest] = array
```

Объявление переменной в начале конструкции не обязательно. Можно использовать и существующие переменные. Однако при деструктуризации объекта может потребоваться обернуть выражение в скобки.

- Вложенные объекты и массивы тоже работают, при деструктуризации нужно лишь сохранить ту же структуру, что и исходный объект/массив.

Как мы увидим далее, деструктуризации особенно удобны при чтении объектных параметров функций.

Функции

В функциях основные изменения касаются передачи параметров, плюс введена дополнительная короткая запись через стрелочку => .

Параметры по умолчанию

Можно указывать параметры по умолчанию через равенство = , например:

```
function showMenu(title = "Без заголовка", width = 100, height = 200) {
  alert(title + ' ' + width + ' ' + height);
}

showMenu("Меню"); // Меню 100 200
```

Параметр по умолчанию используется при отсутствующем аргументе или равном undefined , например:

```
function showMenu(title = "Заголовок", width = 100, height = 200) {
  alert('title=' + title + ' width=' + width + ' height=' + height);
}

// По умолчанию будут взяты 1 и 3 параметра
// title=Заголовок width=null height=200
showMenu(undefined, null);
```

При передаче любого значения, кроме undefined , включая пустую строку, ноль или null , параметр считается переданным, и значение по умолчанию не используется.

Параметры по умолчанию могут быть не только значениями, но и выражениями.

Например:

```
function sayHi(who = getCurrentUser().toUpperCase()) {
  alert('Привет, ' + who);
}

function getCurrentUser() {
  return 'Вася';
}

sayHi(); // Привет, ВАСЯ
```

Заметим, что значение выражения getCurrentUser().toUpperCase() будет вычислено, и соответствующие функции вызваны – лишь в том случае, если это необходимо, то есть когда функция вызвана без параметра.

В частности, выражение по умолчанию не вычисляется при объявлении функции. В примере выше функция getCurrentUser() будет вызвана именно в последней строке, так как не передан параметр.

Оператор spread вместо arguments

Чтобы получить массив аргументов, можно использовать оператор ... , например:

```
function showName(firstName, lastName, ...rest) {
  alert(firstName + ' ' + lastName + ' - ' + rest);
}

// выведет: Юлий Цезарь - Император,Рима
showName("Юлий", "Цезарь", "Император", "Рима");
```

В rest попадёт массив всех аргументов, начиная со второго.

Заметим, что rest – настоящий массив, с методами map , forEach и другими, в отличие от arguments .

Оператор ... должен быть в конце

Оператор ... собирает «все оставшиеся» аргументы, поэтому такое объявление не имеет смысла:

```
function f(arg1, ...rest, arg2) { // arg2 после ...rest ?!
  // будет ошибка
}
```

Параметр ...rest должен быть в конце функции.

Выше мы увидели использование ... для чтения параметров в объявлении функции. Но этот же оператор можно использовать и при вызове функции, для передачи массива параметров как списка, например:

```
'use strict';

let numbers = [2, 3, 15];

// Оператор ... в вызове передаст массив как список аргументов
// Этот вызов аналогичен Math.max(2, 3, 15)
let max = Math.max(...numbers);

alert( max ); // 15
```

Формально говоря, эти два вызова делают одно и то же:

```
Math.max(...numbers);
Math.max.apply(Math, numbers);
```

Похоже, что первый – короче и красивее.

Деструктуризация в параметрах

Если функция получает объект, то она может его тут же разбить в переменные:

```
'use strict';

let options = {
  title: "Меню",
  width: 100,
  height: 200
};

function showMenu({title, width, height}) {
  alert(title + ' ' + width + ' ' + height); // Меню 100 200
}

showMenu(options);
```

Можно использовать и более сложную деструктуризацию, с соответствиями и значениями по умолчанию:

```
'use strict';

let options = {
  title: "Меню"
};

function showMenu({title="Заголовок", width:w=100, height:h=200}) {
  alert(title + ' ' + w + ' ' + h);
}

// объект options будет разбит на переменные
showMenu(options); // Меню 100 200
```

Заметим, что в примере выше какой-то аргумент у `showMenu()` обязательно должен быть, чтобы разбить его на переменные.

Если хочется, чтобы функция могла быть вызвана вообще без аргументов – нужно добавить ей параметр по умолчанию – уже не внутрь деструктуризации, а в самом списке аргументов:

```
'use strict';

function showMenu({title="Заголовок", width:w=100, height:h=200} = {}) {
  alert(title + ' ' + w + ' ' + h);
}

showMenu(); // Заголовок 100 200
```

В коде выше весь объект аргументов по умолчанию равен пустому объекту `{}`, поэтому всегда есть что деструктуризовать.

Имя «name»

В свойстве `name` у функции находится её имя.

Например:

```
'use strict';

function f() {} // f.name == "f"

let g = function g() {}; // g.name == "g"

alert(f.name + ' ' + g.name) // f g
```

В примере выше показаны Function Declaration и Named Function Expression. В синтаксисе выше довольно очевидно, что у этих функций есть имя `name`. В конце концов, оно указано в объявлении.

Но современный JavaScript идёт дальше, он старается даже анонимным функциям дать разумные имена.

Например, при создании анонимной функции с одновременной записью в переменную или свойство – её имя равно названию переменной (или свойства).

Например:

```
'use strict';
// свойство g.name = "g"
let g = function() {};

let user = {
  // свойство user.sayHi.name == "sayHi"
  sayHi: function() {}
};
```

Функции в блоке

Объявление функции Function Declaration, сделанное в блоке, видно только в этом блоке.

Например:

```
'use strict';
if (true) {
  sayHi(); // работает

  function sayHi() {
    alert("Привет!");
  }
}
sayHi(); // ошибка, функции не существует
```

То есть, иными словами, такое объявление – ведёт себя в точности как если бы `let sayHi = function() {...}` было сделано в начале блока.

Функции через =>

Появился новый синтаксис для задания функций через «стрелку» => .

Его простейший вариант выглядит так:

```
'use strict';
let inc = x => x+1;
alert( inc(1) ); // 2
```

Эти две записи – примерно аналогичны:

```
let inc = x => x+1;
let inc = function(x) { return x + 1; };
```

Как видно, "x => x+1" – это уже готовая функция. Слева от => находится аргумент, а справа – выражение, которое нужно вернуть.

Если аргументов несколько, то нужно обернуть их в скобки, вот так:

```
'use strict';
let sum = (a,b) => a + b;
// аналог с function
// let inc = function(a, b) { return a + b; };
alert( sum(1, 2) ); // 3
```

Если нужно задать функцию без аргументов, то также используются скобки, в этом случае – пустые:

```
'use strict';
// вызов getTime() будет возвращать текущее время
let getTime = () => new Date().getHours() + ':' + new Date().getMinutes();
alert( getTime() ); // текущее время
```

Когда тело функции достаточно большое, то можно его обернуть в фигурные скобки {...} :

```
'use strict';
let getTime = () => {
  let date = new Date();
  let hours = date.getHours();
  let minutes = date.getMinutes();
};
```

```
return hourse + ':' + minutes;
};

alert( getTime() ); // текущее время
```

Заметим, что как только тело функции оборачивается в `{...}`, то её результат уже не возвращается автоматически. Такая функция должна делать явный `return`, как в примере выше, если конечно хочет что-либо вернуть.

Функции-стрелки очень удобны в качестве коллбеков, например:

```
'use strict';

let arr = [5, 8, 3];

let sorted = arr.sort( (a,b) => a - b );

alert(sorted); // 3, 5, 8
```

Такая запись – коротка и понятна. Далее мы познакомимся с дополнительными преимуществами использования функций-стрелок для этой цели.

Функции-стрелки не имеют своего `this`

Внутри функций-стрелок – тот же `this`, что и снаружи.

Это очень удобно в обработчиках событий и коллбэках, например:

```
'use strict';

let group = {
  title: "Наш курс",
  students: ["Вася", "Петя", "Даша"],

  showList: function() {
    this.students.forEach(
      student => alert(this.title + ': ' + student)
    )
  }
}

group.showList();
// Наш курс: Вася
// Наш курс: Петя
// Наш курс: Даша
```

Здесь в `forEach` была использована функция-стрелка, поэтому `this.title` в коллбэке – тот же, что и во внешней функции `showList`. То есть, в данном случае – `group.title`.

Если бы в `forEach` вместо функции-стрелки была обычная функция, то была бы ошибка:

```
'use strict';

let group = {
  title: "Наш курс",
  students: ["Вася", "Петя", "Даша"],

  showList: function() {
    this.students.forEach(function(student) {
      alert(this.title + ': ' + student); // будет ошибка
    })
  }
}

group.showList();
```

При запуске будет "попытка прочитать свойство `title` у `undefined`", так как `.forEach(f)` при запуске `f` не ставит `this`. То есть, `this` внутри `forEach` будет `undefined`.

Функции стрелки нельзя запускать с `new`

Отсутствие у функции-стрелки "своего `this`" влечёт за собой естественное ограничение: такие функции нельзя использовать в качестве конструктора, то есть нельзя вызывать через `new`.

`=>` это не то же самое, что `.bind(this)`

Есть тонкое различие между функцией стрелкой `=>` и обычной функцией, у которой вызван `.bind(this)`:

- Вызовом `.bind(this)` мы передаём текущий `this`, привязывая его к функции.
- При `=>` привязки не происходит, так как функция стрелка вообще не имеет контекста `this`. Поиск `this` в ней осуществляется так же, как и поиск обычной переменной, то есть, выше в замыкании. До появления стандарта ES-2015 такое было невозможно.

Функции-стрелки не имеют своего `arguments`

В качестве `arguments` используются аргументы внешней «обычной» функции.

Например:

```
'use strict';

function f() {
  let showArg = () => alert(arguments[0]);
  showArg();
}

f(1); // 1
```

Вызов `showArg()` выведет `1`, получив его из аргументов функции `f`. Функция-стрелка здесь вызвана без параметров, но это не важно: `arguments` всегда берутся из внешней «обычной» функции.

Сохранение внешнего `this` и `arguments` удобно использовать для форвардинга вызовов и создания декораторов.

Например, декоратор `defer(f, ms)` ниже получает функцию `f` и возвращает обёртку вокруг неё, откладывающую вызов на `ms` миллисекунд:

```
'use strict';

function defer(f, ms) {
  return function() {
    setTimeout(() => f.apply(this, arguments), ms)
  }
}

function sayHi(who) {
  alert('Привет, ' + who);
}

let sayHiDeferred = defer(sayHi, 2000);
sayHiDeferred("Вася"); // Привет, Вася через 2 секунды
```

Аналогичная реализация без функции-стрелки выглядела бы так:

```
function defer(f, ms) {
  return function() {
    let args = arguments;
    let ctx = this;
    setTimeout(function() {
      return f.apply(ctx, args);
    }, ms);
  }
}
```

В этом коде пришлось создавать дополнительные переменные `args` и `ctx` для передачи внешних аргументов и контекста через замыкание.

Итого

Основные улучшения в функциях:

- Можно задавать параметры по умолчанию, а также использовать деструктуризацию для чтения подходящего объекта.
- Оператор `spread` (троеточие) в объявлении позволяет функции получать оставшиеся аргументы в массив: `function f(arg1, arg2, ...rest)`.
- Тот же оператор `spread` в вызове функции позволяет передать её массив как список аргументов (вместо `apply`).
- У функции есть свойство `name`, оно содержит имя, указанное при объявлении функции, либо, если его нет, то имя свойства или переменную, в которую она записана. Есть и некоторые другие ситуации, в которых интерпретатор подставляет «самое подходящее» имя.
- Объявление `Function Declaration` в блоке `{...}` видно только в этом блоке.
- Появились функции-стрелки:
 - Без фигурных скобок возвращают выражение `expr`: `(args) => expr`.
 - С фигурными скобками требуют явного `return`.
 - Не имеют своих `this` и `arguments`, при обращении получают их из окружающего контекста.
 - Не могут быть использованы как конструкторы, с `new`.

Строки

Есть ряд улучшений и новых методов для строк.

Начнём с, пожалуй, самого важного.

Строки-шаблоны

Добавлен новый вид кавычек для строк:

```
let str = `обратные кавычки`;
```

Основные отличия от двойных `"..."` и одинарных `'...'` кавычек:

- В них разрешён перевод строки.

Например:

```
alert(`моя
многострочная
строка`);
```

Заметим, что пробелы и, собственно, перевод строки также входят в строку, и будут выведены.

- Можно вставлять выражения при помощи `${...}`.

Например:

```
'use strict';
let apples = 2;
let oranges = 3;

alert(`${apples} + ${oranges} = ${apples + oranges}`); // 2 + 3 = 5
```

Как видно, при помощи `${...}` можно вставлять как и значение переменной `${apples}`, так и более сложные выражения, которые могут включать в себя операторы, вызовы функций и т.п. Такую вставку называют «интерполяцией».

Функции шаблонизации

Можно использовать свою функцию шаблонизации для строк.

Название этой функции ставится перед первой обратной кавычкой:

```
let str = func`моя строка`;
```

Эта функция будет автоматически вызвана и получит в качестве аргументов строку, разбитую по вхождением параметров `${...}` и сами эти параметры.

Например:

```
'use strict';

function f(strings, ...values) {
  alert(JSON.stringify(strings)); // ["Sum of ","","","\n","!"]
  alert(JSON.stringify(strings.raw)); // ["Sum of ","","","\n","!"]
  alert(JSON.stringify(values)); // [3,5,8]
}

let apples = 3;
let oranges = 5;

// | s[0] | v[0] | s[1] | v[1] | s[2] | | v[2] | | s[3]
let str = f`Sum of ${apples} + ${oranges} =\n ${apples + oranges}`;
```

В примере выше видно, что строка разбивается по очереди на части: «кусоч строки» – «параметр» – «кусоч строки» – «параметр».

- Участки строки идут в первый аргумент-массив `strings`.
- У этого массива есть дополнительное свойство `strings.raw`. В нём находятся строки в точности как в оригинале. Это влияет на спец-символы, например в `strings` символ `\n` – это перевод строки, а в `strings.raw` – это именно два символа `\n`.
- Дальнейший список аргументов функции шаблонизации – это значения выражений в `${...}`, в данном случае их три.

Зачем `strings.raw`?

В отличие от `strings`, в `strings.raw` содержатся участки строки в «изначально введённом» виде.

То есть, если в строке находится `\n` или `\u1234` или другое особое сочетание символов, то оно таким и останется.

Это нужно в тех случаях, когда функция шаблонизации хочет произвести обработку полностью самостоятельно (свои спец. символы?). Или же когда обработка спец. символов не нужна – например, строка содержит «обычный текст», набранный непрограммистом без учёта спец. символов.

Как видно, функция имеет доступ ко всему: к выражениям, к участкам текста и даже, через `strings.raw` – к оригинально введённому тексту без учёта стандартных спец. символов.

Функция шаблонизации может как-то преобразовать строку и вернуть новый результат.

В простейшем случае можно просто «склеить» полученные фрагменты в строку:

```
'use strict';

// str восстанавливает строку
function str(strings, ...values) {
  let str = "";
  for(let i=0; i<values.length; i++) {
    str += strings[i];
    str += values[i];
  }
}
```

```

// последний кусок строки
str += strings[strings.length-1];
return str;
}

let apples = 3;
let oranges = 5;

// Sum of 3 + 5 = 8!
alert( `Str`Sum of ${apples} + ${oranges} = ${apples + oranges}!` );

```

Функция `str` в примере выше делает то же самое, что обычные обратные кавычки. Но, конечно, можно пойти намного дальше. Например, генерировать из HTML-строки DOM-узлы (функции шаблонизации не обязательно возвращать именно строку).

Или можно реализовать интернационализацию. В примере ниже функция `i18n` осуществляет перевод строки.

Она подбирает по строке вида "Hello, `{name}!`" шаблон перевода "Привет, `{0}!`" (где `{0}` – место для вставки параметра) и возвращает переведённый результат со вставленным именем `name` :

```

'use strict';

let messages = {
  "Hello, {0}!": "Привет, {0}!"
};

function i18n(strings, ...values) {
  // По форме строки получим шаблон для поиска в messages
  // На месте каждого из значений будет его номер: {0}, {1}, ...
  let pattern = "";
  for(let i=0; i<values.length; i++) {
    pattern += strings[i] + '{' + i + '}';
  }
  pattern += strings[strings.length-1];
  // Теперь pattern = "Hello, {0}!"

  let translated = messages[pattern]; // "Привет, {0}!"

  // Заменит в "Привет, {0}" цифры вида {num} на values[num]
  return translated.replace(/\{(\d)\}/g, (s, num) => values[num]);
}

// Пример использования
let name = "Вася";

// Перевести строку
alert( i18n`Hello, ${name}!` ); // Привет, Вася!

```

Итоговое использование выглядит довольно красиво, не правда ли?

Разумеется, эту функцию можно улучшить и расширить. Функция шаблонизации – это своего рода «стандартный синтаксический сахар» для упрощения форматирования и парсинга строк.

Улучшена поддержка юникода

Внутренняя кодировка строк в JavaScript – это UTF-16, то есть под каждый символ отводится ровно два байта.

Но под всевозможные символы всех языков мира 2 байт не хватает. Поэтому бывает так, что одному символу языка соответствует два юникодных символа (итого 4 байта). Такое сочетание называют «суррогатной парой».

Самый частый пример суррогатной пары, который можно встретить в литературе – это китайские иероглифы.

Заметим, однако, что не всякий китайский иероглиф – суррогатная пара. Существенная часть «основного» юникод-диапазона как раз отдана под китайский язык, поэтому некоторые иероглифы – которые в неё «влезли» – представляются одним юникод-символом, а те, которые не поместились (реже используемые) – двумя.

Например:

```

alert( '我'.length ); // 1
alert( '饜'.length ); // 2

```

В тексте выше для первого иероглифа есть отдельный юникод-символ, и поэтому длина строки `1`, а для второго используется суррогатная пара. Соответственно, длина – `2`.

Китайскими иероглифами суррогатные пары, естественно, не ограничиваются.

Ими представлены редкие математические символы, а также некоторые символы для эмоций, к примеру:

```

alert( 'X'.length ); // 2, MATHEMATICAL SCRIPT CAPITAL X
alert( '😂'.length ); // 2, FACE WITH TEARS OF JOY

```

В современный JavaScript добавлены методы `String.fromCodePoint` и `str.codePointAt` – аналоги `String.fromCharCode` и `str.charCodeAt`, корректно работающие с суррогатными парами.

Например, `charCodeAt` считает суррогатную пару двумя разными символами и возвращает код каждой:

```

// как будто в строке два разных символа (на самом деле один)
alert( 'X'.charCodeAt(0) + ' ' + 'X'.charCodeAt(1) ); // 55349 56499

```

...В то время как `codePointAt` возвращает его Unicode-код суррогатной пары правильно:

```
// один символ с "длинным" (более 2 байт) unicode-кодом
alert( 'X'.codePointAt(0) ); // 119987
```

Метод `String.fromCharCode(code)` корректно создаёт строку из «длинного кода», в отличие от старого `String.fromCharCode(code)`.

Например:

```
// Правильно
alert( String.fromCharCode(119987) ); // X
// Неверно!
alert( String.fromCharCode(119987) ); // 裂
```

Более старый метод `fromCharCode` в последней строке дал неверный результат, так как он берёт только первые два байта от числа 119987 и создаёт символ из них, а остальные отбрасывает.

`\u{длинный код}`

Есть и ещё синтаксическое улучшение для больших Unicode-кодов.

В JavaScript-строках давно можно вставлять символы по Unicode-коду, вот так:

```
alert( "\u2033" ); // ", символ двойного штриха
```

Синтаксис: `\uNNNN`, где `NNNN` – четырёхзначный шестнадцатичный код, причём он должен быть ровно четырёхзначным.

«Лишние» цифры уже не войдут в код, например:

```
alert( "\u20331" ); // Два символа: символ двойного штриха ", а затем 1
```

Чтобы вводить более длинные коды символов, добавили запись `\u{NNNNNNNN}`, где `NNNNNNNN` – максимально восьмизначный (но можно и меньше цифр) код.

Например:

```
alert( "\u{20331}" ); // 𠄎, китайский иероглиф с этим кодом
```

Unicode-нормализация

Во многих языках есть символы, которые получаются как сочетание основного символа и какого-то значка над ним или под ним.

Например, на основе обычного символа `а` существуют символы: `áâãäåä`. Самые часто встречающиеся подобные сочетания имеют отдельный юникодный код. Но отнюдь не все.

Для генерации произвольных сочетаний используются несколько юникодных символов: основа и один или несколько значков.

Например, если после символа `S` идёт символ «точка сверху» (код `\u0307`), то показано это будет как «S с точкой сверху» `Š`.

Если нужен ещё значок над той же буквой (или под ней) – без проблем. Просто добавляем соответствующий символ.

К примеру, если добавить символ «точка снизу» (код `\u0323`), то будет «S с двумя точками сверху и снизу» `Š̌`.

Пример этого символа в JavaScript-строке:

```
alert("S\u0307\u0323"); // Š̌
```

Такая возможность добавить произвольной букве нужные значки, с одной стороны, необходима, а с другой стороны – возникает проблемка: можно представить одинаковый с точки зрения визуального отображения и интерпретации символ – разными сочетаниями Unicode-кодов.

Вот пример:

```
alert("S\u0307\u0323"); // Š̌
alert("S\u0323\u0307"); // Š̌
alert( "S\u0307\u0323" == "S\u0323\u0307" ); // false
```

В первой строке после основы `S` идёт сначала значок «верхняя точка», а потом – нижняя, во второй – наоборот. По кодам строки не равны друг другу. Но символ задают один и тот же.

С целью разрешить эту ситуацию, существует *юникодная нормализация*, при которой строки приводятся к единому, «нормальному», виду.

В современном JavaScript это делает метод `str.normalize()` [↗](#).

```
alert( "S\u0307\u0323".normalize() == "S\u0323\u0307".normalize() ); // true
```

Забавно, что в данной конкретной ситуации `normalize()` приведёт последовательность из трёх символов к одному: `\u1e68` (S с двумя точками) [↗](#).

```
alert( "S\u0307\u0323".normalize().length ); // 1, нормализовало в один символ
alert( "S\u0307\u0323".normalize() == "\u1e68" ); // true
```

Это, конечно, не всегда так, просто в данном случае оказалось, что именно такой символ в юникоде уже есть. Если добавить значков, то нормализация уже даст несколько символов.

Для большинства практических задач информации, данной выше, должно быть вполне достаточно, но если хочется более подробно ознакомиться с вариантами и правилами нормализации – они описаны в приложении к стандарту юникод [Unicode Normalization Forms](#).

Полезные методы

Добавлены ряд полезных методов общего назначения:

- [str.includes\(s\)](#) – проверяет, включает ли одна строка в себя другую, возвращает true/false .
- [str.endsWith\(s\)](#) – возвращает true , если строка str заканчивается подстрокой s .
- [str.startsWith\(s\)](#) – возвращает true , если строка str начинается со строки s .
- [str.repeat\(times\)](#) – повторяет строку str times раз.

Конечно, всё это можно было сделать при помощи других встроенных методов, но новые методы более удобны.

Итого

Улучшения:

- Строки-шаблоны – для удобного задания строк (многострочных, с переменными), плюс возможность использовать функцию шаблонизации для самостоятельного форматирования.
- Юникод – улучшена работа с суррогатными парами.
- Полезные методы для проверок вхождения одной строки в другую.

Объекты и прототипы

В этом разделе мы рассмотрим нововведения, которые касаются именно объектов.

По классам – чуть позже, в отдельном разделе, оно того заслуживает.

Короткое свойство

Зачастую у нас есть переменные, например, name и isAdmin , и мы хотим использовать их в объекте.

При объявлении объекта в этом случае достаточно указать только имя свойства, а значение будет взято из переменной с аналогичным именем.

Например:

```
'use strict';
let name = "Вася";
let isAdmin = true;

let user = {
  name,
  isAdmin
};
alert( JSON.stringify(user) ); // {"name": "Вася", "isAdmin": true}
```

Вычисляемые свойства

В качестве имени свойства можно использовать выражение, например:

```
'use strict';
let propName = "firstName";

let user = {
  [propName]: "Вася"
};

alert( user.firstName ); // Вася
```

Или даже так:

```
'use strict';
let a = "Мой ";
let b = "Зелёный ";
let c = "Крокодил";

let user = {
  [(a + b + c).toLowerCase()]: "Вася"
};
```

```
alert( user["мой зелёный крокодил"] ); // Вася
```

Геттер-сеттер для прототипа

В ES5 для прототипа был метод-геттер:

- `Object.getPrototypeOf(obj)`

В ES-2015 также добавился сеттер:

- `Object.setPrototypeOf(obj, newProto)`

...А также «узаконено» свойство `__proto__`, которое даёт прямой доступ к прототипу. Его, в качестве «нестандартного», но удобного способа работы с прототипом, реализовали почти все браузеры (кроме IE10-), так что было принято решение добавить его в стандарт.

Object.assign

Функция `Object.assign` получает список объектов и копирует в первый `target` свойства из остальных.

Синтаксис:

```
Object.assign(target, src1, src2...)
```

При этом последующие свойства перезаписывают предыдущие.

Например:

```
'use strict';  
  
let user = { name: "Вася" };  
let visitor = { isAdmin: false, visits: true };  
let admin = { isAdmin: true };  
  
Object.assign(user, visitor, admin);  
  
// user <- visitor <- admin  
alert( JSON.stringify(user) ); // name: Вася, visits: true, isAdmin: true
```

Его также можно использовать для 1-уровневого клонирования объекта:

```
'use strict';  
  
let user = { name: "Вася", isAdmin: false };  
  
// clone = пустой объект + все свойства user  
let clone = Object.assign({}, user);
```

Object.is(value1, value2)

Новая функция для проверки равенства значений.

Возвращает `true`, если значения `value1` и `value2` равны, иначе `false`.

Она похожа на обычное строгое равенство `===`, но есть отличия:

```
// Сравнение +0 и -0  
alert( Object.is(+0, -0) ); // false  
alert( +0 === -0 ); // true  
  
// Сравнение с NaN  
alert( Object.is(NaN, NaN) ); // true  
alert( NaN === NaN ); // false
```

Отличия эти в большинстве ситуаций не критичны, так что непохоже, чтобы эта функция вытеснила обычную проверку `===`. Что интересно – этот алгоритм сравнения, который называется [SameValue](#), применяется во внутренних реализациях различных методов современного стандарта.

Методы объекта

Долгое время в JavaScript термин «метод объекта» был просто альтернативным названием для свойства-функции.

Теперь это уже не так. Добавлены именно «методы объекта», которые, по сути, являются свойствами-функциями, привязанными к объекту.

Их особенности:

1. Более короткий синтаксис объявления.
2. Наличие в методах специального внутреннего свойства `[[HomeObject]]` («домашний объект»), ссылающегося на объект, которому метод принадлежит. Мы посмотрим его использование чуть дальше.

Для объявления метода вместо записи "prop: function() {...}" нужно написать просто "prop() { ... }".

Например:

```
'use strict';

let name = "Вася";
let user = {
  name,
  // вместо "sayHi: function() { мы пишем "sayHi() {"
  sayHi() {
    alert(this.name);
  }
};

user.sayHi(); // Вася
```

Как видно, для создания метода нужно писать меньше букв. Что же касается вызова – он ничем не отличается от обычной функции. На данном этапе можно считать, что «метод» – это просто сокращённый синтаксис для свойства-функции. Дополнительные возможности, которые даёт такое объявление, мы рассмотрим позже.

Также методами станут объявления геттеров `get prop()` и сеттеров `set prop()`:

```
'use strict';

let name = "Вася", surname="Петров";
let user = {
  name,
  surname,
  get fullName() {
    return `${name} ${surname}`;
  }
};

alert( user.fullName ); // Вася Петров
```

Можно задать и метод с вычисляемым названием:

```
'use strict';

let methodName = "getFirstName";

let user = {
  // в квадратных скобках может быть любое выражение,
  // которое должно вернуть название метода
  [methodName]() { // вместо [methodName]: function() {
    return "Вася";
  }
};

alert( user.getFirstName() ); // Вася
```

Итак, мы рассмотрели синтаксические улучшения. Если коротко, то не надо писать слово «function». Теперь перейдём к другим отличиям.

super

В ES-2015 появилось новое ключевое слово `super`. Оно предназначено только для использования в методах объекта.

Вызов `super.parentProperty` позволяет из метода объекта получить свойство его прототипа.

Например, в коде ниже `rabbit` наследует от `animal`.

Вызов `super.walk()` из метода объекта `rabbit` обращается к `animal.walk()`:

```
'use strict';

let animal = {
  walk() {
    alert("I'm walking");
  }
};

let rabbit = {
  __proto__: animal,
  walk() {
    alert(super.walk); // walk() { ... }
    super.walk(); // I'm walking
  }
};

rabbit.walk();
```

Как правило, это используется в **классах**, которые мы рассмотрим в следующем разделе, но важно понимать, что «классы» здесь на самом деле ни при чём. Свойство `super` работает через прототип, на уровне методов объекта.

При обращении через `super` используется `[[HomeObject]]` текущего метода, и от него берётся `__proto__`. Поэтому `super` работает только внутри методов.

В частности, если переписать этот код, оформив `rabbit.walk` как обычное свойство-функцию, то будет ошибка:

```
'use strict';

let animal = {
  walk() {
    alert("I'm walking");
  }
};

let rabbit = {
  __proto__: animal,
  walk: function() { // Надо: walk() {
    super.walk(); // Будет ошибка!
  }
};

rabbit.walk();
```

Ошибка возникнет, так как `rabbit.walk` теперь обычная функция и не имеет `[[HomeObject]]`. Поэтому в ней не работает `super`.

Исключением из этого правила являются функции-стрелки. В них используется `super` внешней функции. Например, здесь функция-стрелка в `setTimeout` берёт внешний `super`:

```
'use strict';

let animal = {
  walk() {
    alert("I'm walking");
  }
};

let rabbit = {
  __proto__: animal,
  walk() {
    setTimeout(() => super.walk()); // I'm walking
  }
};

rabbit.walk();
```

Ранее мы говорили о том, что у функций-стрелок нет своего `this`, `arguments`: они используют те, которые во внешней функции. Теперь к этому списку добавился ещё и `super`.

i Свойство `[[HomeObject]]` – не изменяемое

При создании метода – он привязан к своему объекту навсегда. Технически можно даже скопировать его и запустить отдельно, и `super` продолжит работать:

```
'use strict';

let animal = {
  walk() { alert("I'm walking"); }
};

let rabbit = {
  __proto__: animal,
  walk() {
    super.walk();
  }
};

let walk = rabbit.walk; // скопируем метод в переменную
walk(); // вызовет animal.walk()
// I'm walking
```

В примере выше метод `walk()` запускается отдельно от объекта, но всё равно, благодаря `[[HomeObject]]`, сохраняется доступ к его прототипу через `super`.

Это – скорее технический момент, так как методы объекта, всё же, предназначены для вызова в контексте этого объекта. В частности, правила для `this` в методах – те же, что и для обычных функций. В примере выше при вызове `walk()` без объекта `this` будет `undefined`.

Итого

Улучшения в описании свойств:

- Запись `name`: `name` можно заменить на просто `name`
- Если имя свойства находится в переменной или задано выражением `expr`, то его можно указать в квадратных скобках `[expr]`.
- Свойства-функции можно оформить как методы: `"prop: function() {}" → "prop() {}"`.

В методах работает обращение к свойствам прототипа через `super.parentProperty`.

Для работы с прототипом:

- `Object.setPrototypeOf(obj, proto)` – метод для установки прототипа.
- `obj.__proto__` – ссылка на прототип.

Дополнительно:

- Метод `Object.assign(target, src1, src2...)` – копирует свойства из всех аргументов в первый объект.
- Метод `Object.is(value1, value2)` проверяет два значения на равенство. В отличие от `===` считает `+0` и `-0` разными числами. А также считает, что `NaN` равно самому себе.

Классы

В современном JavaScript появился новый, «более красивый» синтаксис для классов.

Новая конструкция `class` – удобный «синтаксический сахар» для задания конструктора вместе с прототипом.

Class

Синтаксис для классов выглядит так:

```
class Название [extends Родитель] {
  constructor
  методы
}
```

Например:

```
'use strict';
class User {
  constructor(name) {
    this.name = name;
  }
  sayHi() {
    alert(this.name);
  }
}
let user = new User("Вася");
user.sayHi(); // Вася
```

Функция `constructor` запускается при создании `new User`, остальные методы записываются в `User.prototype`.

Это объявление примерно аналогично такому:

```
function User(name) {
  this.name = name;
}
User.prototype.sayHi = function() {
  alert(this.name);
};
```

В обоих случаях `new User` будет создавать объекты. Метод `sayHi` также в обоих случаях находится в прототипе.

Но при объявлении через `class` есть и ряд отличий:

- `User` нельзя вызывать без `new`, будет ошибка.
- Объявление класса с точки зрения области видимости ведёт себя как `let`. В частности, оно видно только в текущем блоке и только в коде, который находится ниже объявления (Function Declaration видно и до объявления).

Методы, объявленные внутри `class`, также имеют ряд особенностей:

- Метод `sayHi` является именно методом, то есть имеет доступ к `super`.
- Все методы класса работают в строгом режиме `use strict`, даже если он не указан.
- Все методы класса не перечислимы. То есть в цикле `for...in` по объекту их не будет.

Class Expression

Также, как и Function Expression, классы можно задавать «инлайн», в любом выражении и внутри вызова функции.

Это называется Class Expression:

```
'use strict';
let User = class {
  sayHi() { alert('Привет!'); }
};
new User().sayHi();
```

В примере выше у класса нет имени, что один-в-один соответствует синтаксису функций. Но имя можно дать. Тогда оно, как и в Named Function Expression, будет доступно только внутри класса:


```
'use strict';

let SiteGuest = class User {
  sayHi() { alert('Привет!'); }
};

new SiteGuest().sayHi(); // Привет
new User(); // ошибка
```

В примере выше имя `User` будет доступно только внутри класса и может быть использовано, например, для создания новых объектов данного типа.

Наиболее очевидная область применения этой возможности – создание вспомогательного класса прямо при вызове функции.

Например, функция `createModel` в примере ниже создаёт объект по классу и данным, добавляет ему `_id` и пишет в «реестр» `allModels`:

```
'use strict';

let allModels = {};

function createModel(Model, ...args) {
  let model = new Model(...args);

  model._id = Math.random().toString(36).slice(2);
  allModels[model._id] = model;

  return model;
}

let user = createModel(class User {
  constructor(name) {
    this.name = name;
  }
  sayHi() {
    alert(this.name);
  }
}, "Вася");

user.sayHi(); // Вася

alert( allModels[user._id].name ); // Вася
```

Геттеры, сеттеры и вычисляемые свойства

В классах, как и в обычных объектах, можно объявлять геттеры и сеттеры через `get/set`, а также использовать `[...]` для свойств с вычисляемыми именами:

```
'use strict';

class User {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  // геттер
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  }

  // сеттер
  set fullName(newValue) {
    [this.firstName, this.lastName] = newValue.split(' ');
  }

  // вычисляемое название метода
  ["test".toUpperCase]() {
    alert("PASSED!");
  }
};

let user = new User("Вася", "Пупков");
alert( user.fullName ); // Вася Пупков
user.fullName = "Иван Петров";
alert( user.fullName ); // Иван Петров
user.TEST(); // PASSED!
```

При чтении `fullName` будет вызван метод `get fullName()`, при присвоении – метод `set fullName` с новым значением.

class не позволяет задавать свойства-значения

В синтаксисе классов, как мы видели выше, можно создавать методы. Они будут записаны в прототип, как например `User.prototype.sayHi`.

Однако, нет возможности задать в прототипе обычное значение (не функцию), такое как `User.prototype.key = "value"`.

Конечно, никто не мешает после объявления класса в прототип дописать подобные свойства, однако предполагается, что в прототипе должны быть только методы.

Если свойство-значение, всё же, необходимо, то можно создать геттер, который будет нужное значение возвращать.

Статические свойства

Класс, как и функция, является объектом. Статические свойства класса `User` – это свойства непосредственно `User`, то есть доступные из него «через точку».

Для их объявления используется ключевое слово `static`.

Например:

```
'use strict';

class User {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  static createGuest() {
    return new User("Гость", "Сайта");
  }
};

let user = User.createGuest();

alert( user.firstName ); // Гость

alert( User.createGuest ); // createGuest ... (функция)
```

Как правило, они используются для операций, не требующих наличия объекта, например – для фабричных, как в примере выше, то есть как альтернативные варианты конструктора. Или же, можно добавить метод `User.compare`, который будет сравнивать двух пользователей для целей сортировки.

Также статическими удобно делать константы:

```
'use strict';

class Menu {
  static get elemClass() {
    return "menu"
  }
}

alert( Menu.elemClass ); // menu
```

Наследование

Синтаксис:

```
class Child extends Parent {
  ...
}
```

Посмотрим как это выглядит на практике. В примере ниже объявлено два класса: `Animal` и наследующий от него `Rabbit`:

```
'use strict';

class Animal {
  constructor(name) {
    this.name = name;
  }

  walk() {
    alert("I walk: " + this.name);
  }
}

class Rabbit extends Animal {
  walk() {
    super.walk();
    alert("...and jump!");
  }
}

new Rabbit("Вася").walk();
// I walk: Вася
// and jump!
```

Как видим, в `new Rabbit` доступны как свои методы, так и (через `super`) методы родителя.

Это потому, что при наследовании через `extends` формируется стандартная цепочка прототипов: методы `Rabbit` находятся в `Rabbit.prototype`, методы `Animal` – в `Animal.prototype`, и они связаны через `__proto__`:

```
'use strict';

class Animal { }
class Rabbit extends Animal { }

alert( Rabbit.prototype.__proto__ == Animal.prototype ); // true
```

Как видно из примера выше, методы родителя (`walk`) можно переопределить в наследнике. При этом для обращения к родительскому методу используют `super.walk()`.

С конструктором – немного особая история.

Конструктор `constructor` родителя наследуется автоматически. То есть, если в потомке не указан свой `constructor`, то используется родительский. В примере выше `Rabbit`, таким образом, использует `constructor` от `Animal`.

Если же у потомка свой `constructor`, то, чтобы в нём вызвать конструктор родителя – используется синтаксис `super()` с аргументами для родителя.

Например, вызовем конструктор `Animal` в `Rabbit`:

```
'use strict';

class Animal {
  constructor(name) {
    this.name = name;
  }

  walk() {
    alert("I walk: " + this.name);
  }
}

class Rabbit extends Animal {
  constructor() {
    // вызвать конструктор Animal с аргументом "Кроль"
    super("Кроль"); // то же, что и Animal.call(this, "Кроль")
  }
}

new Rabbit().walk(); // I walk: Кроль
```

Для такого вызова есть небольшие ограничения:

- Вызвать конструктор родителя можно только изнутри конструктора потомка. В частности, `super()` нельзя вызвать из произвольного метода.
- В конструкторе потомка мы обязаны вызвать `super()` до обращения к `this`. До вызова `super` не существует `this`, так как по спецификации в этом случае именно `super` инициализует `this`.

Второе ограничение выглядит несколько странно, поэтому проиллюстрируем его примером:

```
'use strict';

class Animal {
  constructor(name) {
    this.name = name;
  }
}

class Rabbit extends Animal {
  constructor() {
    alert(this); // ошибка, this не определён!
    // обязаны вызвать super() до обращения к this
    super();
    // а вот здесь уже можно использовать this
  }
}

new Rabbit();
```

Итого

- Классы можно объявлять как в основном потоке кода, так и «инлайн», по аналогии с `Function Declaration` и `Expression`.
- В объявлении классов можно использовать методы, геттеры/сеттеры и вычисляемые названия методов.
- При наследовании вызов конструктора родителя осуществляется через `super(...args)`, вызов родительских методов – через `super.method(...args)`.

Концепция классов, которая после долгих обсуждений получилась в стандарте ECMAScript, носит название «максимально минимальной». То есть, в неё вошли только те возможности, которые уж точно необходимы.

В частности, не вошли «приватные» и «защищённые» свойства. То есть, все свойства и методы класса технически доступны снаружи. Возможно, они появятся в будущих редакциях стандарта.

Тип данных `Symbol`

Новый примитивный тип данных `Symbol` служит для создания уникальных идентификаторов.

Мы вначале рассмотрим объявление и особенности символов, а затем – их использование.

Объявление

Синтаксис:

```
let sym = Symbol();
```

Обратим внимание, не `new Symbol`, а просто `Symbol`, так как это – примитив.

У символов есть и соответствующий `typeof` :

```
'use strict';
let sym = Symbol();
alert( typeof sym ); // symbol
```

Каждый символ – уникален. У функции `Symbol` есть необязательный аргумент «имя символа». Его можно использовать для описания символа, в целях отладки:

```
'use strict';
let sym = Symbol("name");
alert( sym.toString() ); // Symbol(name)
```

...Но при этом, если у двух символов одинаковое имя, то это не значит, что они равны:

```
alert( Symbol("name") == Symbol("name") ); // false
```

Если хочется из разных частей программы использовать именно одинаковый символ, то можно передавать между ними объект символа или же – использовать «глобальные символы» и «реестр глобальных символов», которые мы рассмотрим далее.

Глобальные символы

Существует «глобальный реестр» символов, который позволяет, при необходимости, иметь общие «глобальные» символы, которые можно получить из реестра по имени.

Для чтения (или создания, при отсутствии) «глобального» символа служит вызов `Symbol.for(имя)` .

Например:

```
'use strict';
// создание символа в реестре
let name = Symbol.for("name");
// символ уже есть, чтение из реестра
alert( Symbol.for("name") == name ); // true
```

Таким образом, можно из разных частей программы, обратившись к реестру, получить единый глобальный символ с именем `"name"` .

На заметку:

В некоторых языках программирования, например Ruby, имя однозначно идентифицирует символ. В JavaScript, как мы видим, это верно для глобальных символов.

У вызова `Symbol.for` , который возвращает символ по имени, есть обратный вызов – `Symbol.keyFor(sym)` . Он позволяет получить по глобальному символу его имя:

```
'use strict';
// создание символа в реестре
let name = Symbol.for("name");
// получение имени символа
alert( Symbol.keyFor(name) ); // name
```

`Symbol.keyFor` возвращает `undefined` , если символ не глобальный

Заметим, что `Symbol.keyFor` работает *только для глобальных символов*, для остальных будет возвращено `undefined` :

```
'use strict';
alert( Symbol.keyFor(Symbol.for("name")) ); // name, глобальный
alert( Symbol.keyFor(Symbol("name2")) ); // undefined, обычный символ
```

Таким образом, имя символа, если этот символ не глобальный, не имеет особого применения, оно полезно лишь в целях вывода и отладки.

Использование символов

Символы можно использовать в качестве имён для свойств объекта следующим образом:

```
'use strict';
let isAdmin = Symbol("isAdmin");
```

```
let user = {
  name: "Вася",
  [isAdmin]: true
};

alert(user[isAdmin]); // true
```

Особенность символов в том, что если в объект записать свойство-символ, то оно не участвует в итерации:

```
'use strict';

let user = {
  name: "Вася",
  age: 30,
  [Symbol.for("isAdmin")]: true
};

// в цикле for..in также не будет символа
alert( Object.keys(user) ); // name, age

// доступ к свойству через глобальный символ – работает
alert( user[Symbol.for("isAdmin")] );
```

Кроме того, свойство-символ недоступно, если обратиться к его названию: `user.isAdmin` не существует.

Зачем всё это, почему просто не использовать строки?

Резонный вопрос. На ум могут прийти соображения производительности, так как символы – это по сути специальные идентификаторы, они компактнее, чем строка. Но при современных оптимизациях объектов это редко имеет значение.

Самое широкое применение символов предусмотрено внутри самого стандарта JavaScript. В современном стандарте есть много системных символов. Их список есть в спецификации, в таблице [Well-known Symbols](#). В спецификации для краткости символы принято обозначать как „@имя“, например `@iterator`, но доступны они как свойства `Symbol`.

Например:

- `Symbol.toPrimitive` – идентификатор для свойства, задающего функцию преобразования объекта в примитив.
- `Symbol.iterator` – идентификатор для свойства, задающего функцию итерации по объекту.
- ...и т.п.

Мы легко поймём смысл введения нового типа «символ», если поставим себя на место создателей языка JavaScript.

Допустим, в новом стандарте нам надо добавить к объекту «особый» функционал, например, функцию, которая задаёт преобразование объекта к примитиву. Как `obj.toString`, но для преобразования в примитивы.

Мы ведь не можем просто сказать, что «свойство `obj.toPrimitive` теперь будет задавать преобразование к примитиву и автоматически вызываться в таких-то ситуациях». Это опасно. Мы не можем так просто взять и придать особый смысл свойству. Мало ли, вполне возможно, что свойство с таким именем уже используется в существующем коде, и если сделать его особым, то он сломается.

Нельзя просто взять и зарезервировать какие-то свойства существующих объектов для нового функционала.

Поэтому ввели целый тип «символы». Их можно использовать для задания таких свойств, так как они:

- а) уникальны,
- б) не участвуют в циклах,
- в) заведомо не ломают старый код, который о них слыхом не слыхивал.

Продемонстрируем отсутствие конфликта для нового системного свойства `Symbol.iterator`:

```
'use strict';

let obj = {
  iterator: 1,
  [Symbol.iterator]() {}
}

alert(obj.iterator); // 1
alert(obj[Symbol.iterator]) // function, символ не конфликтует
```

Выше мы использовали системный символ `Symbol.iterator`, поскольку он один из самых широко поддерживаемых. Мы подробно разберём его смысл в главе про [итераторы](#), пока же – это просто пример символа.

Чтобы получить все символы объекта, есть особый вызов [Object.getOwnPropertySymbols](#).

Эта функция возвращает все символы в объекте (и только их). Заметим, что старая функция `getOwnPropertyNames` символы не возвращает, что опять же гарантирует отсутствие конфликтов со старым кодом.

```
'use strict';

let obj = {
  iterator: 1,
  [Symbol.iterator]: function() {}
}

// один символ в объекте
alert( Object.getOwnPropertySymbols(obj) ); // Symbol(Symbol.iterator)
```

```
// и одно обычное свойство
alert( Object.getOwnPropertyNames(obj) ); // iterator
```

Итого

- Символы – новый примитивный тип, предназначенный для уникальных идентификаторов.
- Все символы уникальны. Символы с одинаковым именем не равны друг другу.
- Существует глобальный реестр символов, доступных через метод `Symbol.for(name)`. Для глобального символа можно получить имя вызовом и `Symbol.keyFor(sym)`.

Основная область использования символов – это системные свойства объектов, которые задают разные аспекты их поведения. Поддержка у них пока небольшая, но она растёт. Системные символы позволяют разработчикам стандарта добавлять новые «особые» свойства объектов, при этом не резервируя соответствующие строковые значения.

Системные символы доступны как свойства функции `Symbol`, например `Symbol.iterator`.

Мы можем создавать и свои символы, использовать их в объектах. Записывать их как свойства `Symbol`, разумеется, нельзя. Если нужен глобально доступный символ, то используется `Symbol.for(имя)`.

Итераторы

В современный JavaScript добавлена новая концепция «итерируемых» (iterable) объектов.

Итерируемые или, иными словами, «перебираемые» объекты – это те, содержимое которых можно перебрать в цикле.

Например, перебираемым объектом является массив. Но не только он. В браузере существует множество объектов, которые не являются массивами, но содержимое которых можно перебрать (к примеру, список DOM-узлов).

Для перебора таких объектов добавлен новый синтаксис цикла: `for...of`.

Например:

```
'use strict';
let arr = [1, 2, 3]; // массив – пример итерируемого объекта
for (let value of arr) {
  alert(value); // 1, затем 2, затем 3
}
```

Также итерируемой является строка:

```
'use strict';
for (let char of "Привет") {
  alert(char); // Выведет по одной букве: П, р, и, в, е, т
}
```

Итераторы – расширяющая понятие «массив» концепция, которая пронизывает современный стандарт JavaScript сверху донизу.

Практически везде, где нужен перебор, он осуществляется через итераторы. Это включает в себя не только строки, массивы, но и вызов функции с оператором `spread f(...args)`, и многое другое.

В отличие от массивов, «перебираемые» объекты могут не иметь «длины» `length`. Как мы увидим далее, итераторы дают возможность сделать «перебираемыми» любые объекты.

Свой итератор

Допустим, у нас есть некий объект, который надо «умным способом» перебрать.

Например, `range` – диапазон чисел от `from` до `to`, и мы хотим, чтобы `for (let num of range)` «перебирал» этот объект. При этом под перебором мы подразумеваем перечисление чисел от `from` до `to`.

Объект `range` без итератора:

```
let range = {
  from: 1,
  to: 5
};
// хотим сделать перебор
// for (let num of range) ...
```

Для возможности использовать объект в `for...of` нужно создать в нём свойство с названием `Symbol.iterator` (системный символ).

При вызове метода `Symbol.iterator` перебираемый объект должен возвращать другой объект («итератор»), который умеет осуществлять перебор.

По стандарту у такого объекта должен быть метод `next()`, который при каждом вызове возвращает очередное значение и окончен ли перебор.

В коде это выглядит следующим образом:

```

'use strict';

let range = {
  from: 1,
  to: 5
}

// сделаем объект range итерируемым
range[Symbol.iterator] = function() {

  let current = this.from;
  let last = this.to;

  // метод должен вернуть объект с методом next()
  return {
    next() {
      if (current <= last) {
        return {
          done: false,
          value: current++
        };
      } else {
        return {
          done: true
        };
      }
    }
  };
};

for (let num of range) {
  alert(num); // 1, затем 2, 3, 4, 5
}

```

Как видно из кода выше, здесь имеет место разделение сущностей:

- Перебираемый объект `range` сам не реализует методы для своего перебора.
- Для этого создаётся другой объект, который хранит текущее состояние перебора и возвращает значение. Этот объект называется итератором и возвращается при вызове метода `range[Symbol.iterator]`.
- У итератора должен быть метод `next()`, который при каждом вызове возвращает объект со свойствами:
 - `value` – очередное значение,
 - `done` – равно `false` если есть ещё значения, и `true` – в конце.

Конструкция `for..of` в начале своего выполнения автоматически вызывает `Symbol.iterator()`, получает итератор и далее вызывает метод `next()` до получения `done: true`. Такова внутренняя механика. Внешний код при переборе через `for..of` видит только значения.

Такое отделение функционала перебора от самого объекта даёт дополнительную гибкость. Например, объект может возвращать разные итераторы в зависимости от своего настроения и времени суток. Однако, бывают ситуации когда оно не нужно.

Если функционал по перебору (метод `next`) предоставляется самим объектом, то можно вернуть `this` в качестве итератора:

```

'use strict';

let range = {
  from: 1,
  to: 5,

  [Symbol.iterator]() {
    return this;
  },

  next() {
    if (this.current === undefined) {
      // инициализация состояния итерации
      this.current = this.from;
    }

    if (this.current <= this.to) {
      return {
        done: false,
        value: this.current++
      };
    } else {
      // очистка текущей итерации
      delete this.current;
      return {
        done: true
      };
    }
  }
};

for (let num of range) {
  alert(num); // 1, затем 2, 3, 4, 5
}

// Произойдёт вызов Math.max(1,2,3,4,5);
alert( Math.max(...range) ); // 5 (*)

```

При таком подходе сам объект и хранит состояние итерации (текущий перебираемый элемент).

В данном случае это работает, но для большей гибкости и понятности кода рекомендуется, всё же, выделять итератор в отдельный объект со своим состоянием и кодом.

i Оператор `spread ...` и итераторы

В последней строке (*) примера выше можно видеть, что итерируемый объект передаётся через `spread` для `Math.max`.

При этом `...range` автоматически превращает итерируемый объект в массив. То есть произойдёт цикл `for..of` по `range`, и его результаты будут использованы в качестве списка аргументов.

i Бесконечные итераторы

Возможны и бесконечные итераторы. Например, пример выше при `range.to = Infinity` будет таковым. Или можно сделать итератор, генерирующий бесконечную последовательность псевдослучайных чисел. Тоже полезно.

Нет никаких ограничений на `next`, он может возвращать всё новые и новые значения, и это нормально.

Разумеется, цикл `for..of` по такому итератору тоже будет бесконечным, нужно его прерывать, например, через `break`.

Встроенные итераторы

Встроенные в JavaScript итераторы можно получить и явным образом, без `for..of`, прямым вызовом `Symbol.iterator`.

Например, этот код получает итератор для строки и вызывает его полностью «вручную»:

```
'use strict';

let str = "Hello";

// Делает то же, что и
// for (var letter of str) alert(letter);

let iterator = str[Symbol.iterator]();

while(true) {
  let result = iterator.next();
  if (result.done) break;
  alert(result.value); // Выведет все буквы по очереди
}
```

То же самое будет работать и для массивов.

Итого

- *Итератор* – объект, предназначенный для перебора другого объекта.
- У итератора должен быть метод `next()`, возвращающий объект `{done: Boolean, value: any}`, где `value` – очередное значение, а `done: true` в конце.
- Метод `Symbol.iterator` предназначен для получения итератора из объекта. Цикл `for..of` делает это автоматически, но можно и вызвать его напрямую.
- В современном стандарте есть много мест, где вместо массива используются более абстрактные «итерируемые» (со свойством `Symbol.iterator`) объекты, например оператор `spread ...`.
- Встроенные объекты, такие как массивы и строки, являются итерируемыми, в соответствии с описанным выше.

Set, Map, WeakSet и WeakMap

В ES-2015 появились новые типы коллекций в JavaScript: `Set`, `Map`, `WeakSet` и `WeakMap`.

Map

`Map` – коллекция для хранения записей вида `ключ:значение`.

В отличие от объектов, в которых ключами могут быть только строки, в `Map` ключом может быть произвольное значение, например:

```
'use strict';

let map = new Map();

map.set('1', 'str1'); // ключ-строка
map.set(1, 'num1'); // число
map.set(true, 'bool1'); // булево значение

// в обычном объекте это было бы одно и то же,
// map сохраняет тип ключа
alert( map.get(1) ); // 'num1'
alert( map.get('1') ); // 'str1'

alert( map.size ); // 3
```

Как видно из примера выше, для сохранения и чтения значений используются методы `get` и `set`. И ключи и значения сохраняются «как есть», без преобразований типов.

Свойство `map.size` хранит общее количество записей в `map`.

Метод `set` можно чейнить:

```
map
  .set('1', 'str1')
  .set(1, 'num1')
  .set(true, 'bool1');
```

При создании `Map` можно сразу инициализировать списком значений.

Объект `map` с тремя ключами, как и в примере выше:

```
let map = new Map([
  ['1', 'str1'],
  [1, 'num1'],
  [true, 'bool1']
]);
```

Аргументом `new Map` должен быть итерируемый объект (не обязательно именно массив). Везде утиная типизация, максимальная гибкость.

В качестве ключей `map` можно использовать и объекты:

```
'use strict';

let user = { name: "Вася" };

// для каждого пользователя будем хранить количество посещений
let visitsCountMap = new Map();

// объект user является ключом в visitsCountMap
visitsCountMap.set(user, 123);

alert( visitsCountMap.get(user) ); // 123
```

Использование объектов в качестве ключей – как раз тот случай, когда `Map` сложно заменить обычными объектами `Object`. Ведь для обычных объектов ключ может быть только строкой.

Как `map` сравнивает ключи

Для проверки значений на эквивалентность используется алгоритм [SameValueZero](#). Он аналогичен строгому равенству `===`, отличие – в том, что `NaN` считается равным `NaN`. Поэтому значение `NaN` также может быть использовано в качестве ключа.

Этот алгоритм нельзя изменять или задавать свою функцию сравнения.

Методы для удаления записей:

- `map.delete(key)` удаляет запись с ключом `key`, возвращает `true`, если такая запись была, иначе `false`.
- `map.clear()` – удаляет все записи, очищает `map`.

Для проверки существования ключа:

- `map.has(key)` – возвращает `true`, если ключ есть, иначе `false`.

Итерация

Для итерации по `map` используется один из трёх методов:

- `map.keys()` – возвращает итерируемый объект для ключей,
- `map.values()` – возвращает итерируемый объект для значений,
- `map.entries()` – возвращает итерируемый объект для записей `[ключ, значение]`, он используется по умолчанию в `for...of`.

Например:

```
'use strict';

let recipeMap = new Map([
  ['огурцов', '500 гр'],
  ['помидоров', '350 гр'],
  ['сметаны', '50 гр']
]);

// цикл по ключам
for(let fruit of recipeMap.keys()) {
  alert(fruit); // огурцов, помидоров, сметаны
}

// цикл по значениям [ключ, значение]
for(let amount of recipeMap.values()) {
  alert(amount); // 500 гр, 350 гр, 50 гр
}

// цикл по записям
for(let entry of recipeMap) { // то же что и recipeMap.entries()
  alert(entry); // огурцов,500 гр , и т.д., массивы по 2 значения
}
```

i Перебор идёт в том же порядке, что и вставка

Перебор осуществляется в порядке вставки. Объекты `Map` гарантируют это, в отличие от обычных объектов `Object`.

Кроме того, у `Map` есть стандартный метод `forEach`, аналогичный массиву:

```
'use strict';

let recipeMap = new Map([
  ['огурцов', '500 гр'],
  ['помидоров', '350 гр'],
  ['сметаны', '50 гр']
]);

recipeMap.forEach( (value, key, map) => {
  alert(` ${key}: ${value} `); // огурцов: 500 гр, и т.д.
});
```

Set

`Set` – коллекция для хранения множества значений, причём каждое значение может встречаться лишь один раз.

Например, к нам приходят посетители, и мы хотели бы сохранять всех, кто пришёл. При этом повторные визиты не должны приводить к дубликатам, то есть каждого посетителя нужно «посчитать» ровно один раз.

`Set` для этого отлично подходит:

```
'use strict';

let set = new Set();

let vasya = {name: "Вася"};
let petya = {name: "Петя"};
let dasha = {name: "Даша"};

// посещения, некоторые пользователи заходят много раз
set.add(vasya);
set.add(petya);
set.add(dasha);
set.add(vasya);
set.add(petya);

// set сохраняет только уникальные значения
alert( set.size ); // 3

set.forEach( user => alert(user.name ) ); // Вася, Петя, Даша
```

В примере выше многократные добавления одного и того же объекта в `set` не создают лишних копий.

Альтернатива `Set` – это массивы с поиском дубликата при каждом добавлении, но они гораздо хуже по производительности. Или можно использовать обычные объекты, где в качестве ключа выступает какой-нибудь уникальный идентификатор посетителя. Но это менее удобно, чем простой и наглядный `Set`.

Основные методы:

- `set.add(item)` – добавляет в коллекцию `item`, возвращает `set` (чейнится).
- `set.delete(item)` – удаляет `item` из коллекции, возвращает `true`, если он там был, иначе `false`.
- `set.has(item)` – возвращает `true`, если `item` есть в коллекции, иначе `false`.
- `set.clear()` – очищает `set`.

Перебор `Set` осуществляется через `forEach` или `for..of` аналогично `Map`:

```
'use strict';

let set = new Set(["апельсины", "яблоки", "бананы"]);

// то же, что: for(let value of set)
set.forEach((value, valueAgain, set) => {
  alert(value); // апельсины, затем яблоки, затем бананы
});
```

Заметим, что в `Set` у функции `forEach` три аргумента: значение, ещё раз значение, и затем сам перебираемый объект `set`. При этом значение повторяется в аргументах два раза.

Так сделано для совместимости с `Map`, где у `forEach`-функции также три аргумента. Но в `Set` первые два всегда совпадают и содержат очередное значение множества.

WeakMap и WeakSet

`WeakSet` – особый вид `Set` не препятствующий сборщику мусора удалять свои элементы. То же самое – `WeakMap` для `Map`.

То есть, если некий объект присутствует только в `WeakSet/WeakMap` – он удаляется из памяти.

Это нужно для тех ситуаций, когда основное место для хранения и использования объектов находится где-то в другом месте кода, а здесь мы хотим хранить для них «вспомогательные» данные, существующие лишь пока жив объект.

Например, у нас есть элементы на странице или, к примеру, пользователи, и мы хотим хранить для них вспомогательную информацию, например обработчики событий или просто данные, но действительные лишь пока объект, к которому они относятся, существует.

Если поместить такие данные в `WeakMap`, а объект сделать ключом, то они будут автоматически удалены из памяти, когда удалится элемент.

Например:

```
// текущие активные пользователи
let activeUsers = [
  {name: "Вася"},
  {name: "Петя"},
  {name: "Маша"}
];

// вспомогательная информация о них,
// которая напрямую не входит в объект юзера,
// и потому хранится отдельно
let weakMap = new WeakMap();

weakMap[activeUsers[0]] = 1;
weakMap[activeUsers[1]] = 2;
weakMap[activeUsers[2]] = 3;

alert( weakMap[activeUsers[0]] ); // 1

activeUsers.splice(0, 1); // Вася более не активный пользователь

// weakMap теперь содержит только 2 элемента

activeUsers.splice(0, 1); // Петя более не активный пользователь

// weakMap теперь содержит только 1 элемент
```

Таким образом, `WeakMap` избавляет нас от необходимости вручную удалять вспомогательные данные, когда удалён основной объект.

У `WeakMap` есть ряд ограничений:

- Нет свойства `size`.
- Нельзя перебрать элементы итератором или `forEach`.
- Нет метода `clear()`.

Иными словами, `WeakMap` работает только на запись (`set`, `delete`) и чтение (`get`, `has`) элементов по конкретному ключу, а не как полноценная коллекция. Нельзя вывести всё содержимое `WeakMap`, нет соответствующих методов.

Это связано с тем, что содержимое `WeakMap` может быть модифицировано сборщиком мусора в любой момент, независимо от программиста. Сборщик мусора работает сам по себе. Он не гарантирует, что очистит объект сразу же, когда это стало возможным. В равной степени он не гарантирует и обратное. Нет какого-то конкретного момента, когда такая очистка точно произойдёт – это определяется внутренними алгоритмами сборщика и его сведениями о системе.

Поэтому содержимое `WeakMap` в произвольный момент, строго говоря, не определено. Может быть, сборщик мусора уже удалил какие-то записи, а может и нет. С этим, а также с требованиями к эффективной реализации `WeakMap`, и связано отсутствие методов, осуществляющих доступ ко всем записям.

То же самое относится и к `WeakSet`: можно добавлять элементы, проверять их наличие, но нельзя получить их список и даже узнать количество.

Эти ограничения могут показаться неудобными, но по сути они не мешают `WeakMap/WeakSet` выполнять свою основную задачу – быть «вторичным» хранилищем данных для объектов, актуальный список которых (и сами они) хранятся в каком-то другом месте.

Итого

- `Map` – коллекция записей вида `ключ: значение`, лучше `Object` тем, что перебирает всегда в порядке вставки и допускает любые ключи.
- `Set` – коллекция уникальных элементов, также допускает любые ключи.

Основная область применения `Map` – ситуации, когда строковых ключей не хватает (нужно хранить соответствия для ключей-объектов), либо когда строковый ключ может быть совершенно произвольным.

К примеру, в обычном объекте `Object` нельзя использовать «совершенно любые» ключи. Есть встроенные методы, и уж точно есть свойство с названием `__proto__`, которое зарезервировано системой. Если название ключа даётся посетителем сайта, то он может попытаться использовать такое свойство, заменить прототип, а это, при запуске JavaScript на сервере, уже может привести к серьёзным ошибкам.

- `WeakMap` и `WeakSet` – «урезанные» по функционалу варианты `Map/Set`, которые позволяют только «точно» обращаться элементам (по конкретному ключу или значению). Они не препятствуют сборке мусора, то есть если ссылка на объект осталась только в `WeakSet/WeakMap` – он будет удалён.

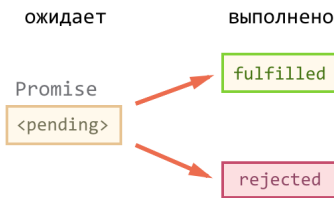
Promise

`Promise` (обычно их так и называют «промисы») – предоставляют удобный способ организации асинхронного кода.

В современном JavaScript промисы часто используются в том числе и неявно, при помощи генераторов, но об этом чуть позже.

Что такое Promise?

Promise – это специальный объект, который содержит своё состояние. Вначале `pending` («ожидание»), затем – одно из: `fulfilled` («выполнено успешно») или `rejected` («выполнено с ошибкой»).



На `promise` можно навешивать коллбэки двух типов:

- `onFulfilled` – срабатывают, когда `promise` в состоянии «выполнен успешно».
- `onRejected` – срабатывают, когда `promise` в состоянии «выполнен с ошибкой».

Способ использования, в общих чертах, такой:

1. Код, которому надо сделать что-то асинхронно, создаёт объект `promise` и возвращает его.
2. Внешний код, получив `promise`, навешивает на него обработчики.
3. По завершении процесса асинхронный код переводит `promise` в состояние `fulfilled` (с результатом) или `rejected` (с ошибкой). При этом автоматически вызываются соответствующие обработчики во внешнем коде.

Синтаксис создания `Promise`:

```
var promise = new Promise(function(resolve, reject) {
  // Эта функция будет вызвана автоматически

  // В ней можно делать любые асинхронные операции,
  // А когда они завершатся – нужно вызвать одно из:
  // resolve(результат) при успешном выполнении
  // reject(ошибка) при ошибке
})
```

Универсальный метод для навешивания обработчиков:

```
promise.then(onFulfilled, onRejected)
```

- `onFulfilled` – функция, которая будет вызвана с результатом при `resolve`.
- `onRejected` – функция, которая будет вызвана с ошибкой при `reject`.

С его помощью можно назначить как оба обработчика сразу, так и только один:

```
// onFulfilled сработает при успешном выполнении
promise.then(onFulfilled)
// onRejected сработает при ошибке
promise.then(null, onRejected)
```

i `.catch`

Для того, чтобы поставить обработчик только на ошибку, вместо `.then(null, onRejected)` можно написать `.catch(onRejected)` – это то же самое.

i Синхронный `throw` – то же самое, что `reject`

Если в функции промиса происходит синхронный `throw` (или иная ошибка), то вызывается `reject`:

```
'use strict';

let p = new Promise((resolve, reject) => {
  // то же что reject(new Error("o_0"))
  throw new Error("o_0");
})

p.catch(alert); // Error: o_0
```

Посмотрим, как это выглядит вместе, на простом примере.

Пример с `setTimeout`

Возьмём `setTimeout` в качестве асинхронной операции, которая должна через некоторое время успешно завершиться с результатом «result»:

```
'use strict';

// Создаётся объект promise
let promise = new Promise((resolve, reject) => {

  setTimeout(() => {
    // переведёт промис в состояние fulfilled с результатом "result"
    resolve("result");
  }, 1000);

});

// promise.then навешивает обработчики на успешный результат или ошибку
promise
  .then(
    result => {
      // первая функция-обработчик - запустится при вызове resolve
      alert("Fulfilled: " + result); // result - аргумент resolve
    },
    error => {
      // вторая функция - запустится при вызове reject
      // вторая функция - запустится при вызове reject
      alert("Rejected: " + error); // error - аргумент reject
    }
  );
```

В результате запуска кода выше – через 1 секунду выведется «Fulfilled: result».

А если бы вместо `resolve("result")` был вызов `reject("error")`, то вывелось бы «Rejected: error». Впрочем, как правило, если при выполнении возникла проблема, то `reject` вызывают не со строкой, а с объектом ошибки типа `new Error`:

```
// Этот promise завершится с ошибкой через 1 секунду
var promise = new Promise((resolve, reject) => {

  setTimeout(() => {
    reject(new Error("время вышло!"));
  }, 1000);

});

promise
  .then(
    result => alert("Fulfilled: " + result),
    error => alert("Rejected: " + error.message) // Rejected: время вышло!
  );
```

Конечно, вместо `setTimeout` внутри функции промиса может быть и запрос к серверу и ожидание ввода пользователя, или другой асинхронный процесс. Главное, чтобы по своему завершению он вызвал `resolve` или `reject`, которые передадут результат обработчикам.

i Только один аргумент

Функции `resolve/reject` принимают ровно один аргумент – результат/ошибку.

Именно он передаётся обработчикам в `.then`, как можно видеть в примерах выше.

Promise после `reject/resolve` – неизменны

Заметим, что после вызова `resolve/reject` промис уже не может «передумать».

Когда промис переходит в состояние «выполнен» – с результатом (`resolve`) или ошибкой (`reject`) – это навсегда.

Например:

```
'use strict';

let promise = new Promise((resolve, reject) => {

  // через 1 секунду готов результат: result
  setTimeout(() => resolve("result"), 1000);

  // через 2 секунды – reject с ошибкой, он будет проигнорирован
  setTimeout(() => reject(new Error("ignored")), 2000);

});

promise
  .then(
    result => alert("Fulfilled: " + result), // сработает
    error => alert("Rejected: " + error) // не сработает
  );
```

В результате вызова этого кода сработает только первый обработчик `then`, так как после вызова `resolve` промис уже получил состояние (с результатом), и в дальнейшем его уже ничто не изменит.

Последующие вызовы `resolve/reject` будут просто проигнорированы.

А так – наоборот, ошибка будет раньше:

```
'use strict';
```

```

let promise = new Promise((resolve, reject) => {
  // reject вызван раньше, resolve будет проигнорирован
  setTimeout(() => reject(new Error("error")), 1000);

  setTimeout(() => resolve("ignored"), 2000);
});

promise
  .then(
    result => alert("Fulfilled: " + result), // не сработает
    error => alert("Rejected: " + error) // сработает
  );

```

Промисификация

Промисификация – это когда берут асинхронный функционал и делают для него обёртку, возвращающую промис.

После промисификации использование функционала зачастую становится гораздо удобнее.

В качестве примера сделаем такую обёртку для запросов при помощи XMLHttpRequest.

Функция `httpGet(url)` будет возвращать промис, который при успешной загрузке данных с `url` будет переходить в `fulfilled` с этими данными, а при ошибке – в `rejected` с информацией об ошибке:

```

function httpGet(url) {
  return new Promise(function(resolve, reject) {
    var xhr = new XMLHttpRequest();
    xhr.open('GET', url, true);

    xhr.onload = function() {
      if (this.status == 200) {
        resolve(this.response);
      } else {
        var error = new Error(this.statusText);
        error.code = this.status;
        reject(error);
      }
    };

    xhr.onerror = function() {
      reject(new Error("Network Error"));
    };

    xhr.send();
  });
}

```

Как видно, внутри функции объект `XMLHttpRequest` создаётся и отсылается как обычно, при `onload/onerror` вызываются, соответственно, `resolve` (при статусе 200) или `reject`.

Использование:

```

httpGet("/article/promise/user.json")
  .then(
    response => alert(`Fulfilled: ${response}`),
    error => alert(`Rejected: ${error}`)
  );

```

Метод `fetch`

Заметим, что ряд современных браузеров уже поддерживает `fetch` – новый встроенный метод для AJAX-запросов, призванный заменить XMLHttpRequest. Он гораздо мощнее, чем `httpGet`. И – да, этот метод использует промисы. Полифилл для него доступен на <https://github.com/github/fetch>.

Цепочки промисов

«Чейнинг» (chaining), то есть возможность строить асинхронные цепочки из промисов – пожалуй, основная причина, из-за которой существуют и активно используются промисы.

Например, мы хотим по очереди:

1. Загрузить данные посетителя с сервера (асинхронно).
2. Затем отправить запрос о нём на github (асинхронно).
3. Когда это будет готово, вывести его github-аватар на экран (асинхронно).
4. ...И сделать код расширяемым, чтобы цепочку можно было легко продолжить.

Вот код для этого, использующий функцию `httpGet`, описанную выше:

```

'use strict';

// сделать запрос
httpGet('/article/promise/user.json')

```

```

// 1. Получить данные о пользователе в JSON и передать дальше
.then(response => {
  console.log(response);
  let user = JSON.parse(response);
  return user;
})
// 2. Получить информацию с github
.then(user => {
  console.log(user);
  return httpGet(`https://api.github.com/users/${user.name}`);
})
// 3. Вывести аватар на 3 секунды (можно с анимацией)
.then(githubUser => {
  console.log(githubUser);
  githubUser = JSON.parse(githubUser);

  let img = new Image();
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.appendChild(img);

  setTimeout(() => img.remove(), 3000); // (*)
});

```

Самое главное в этом коде – последовательность вызовов:

```

httpGet(...)
  .then(...)
  .then(...)
  .then(...)

```

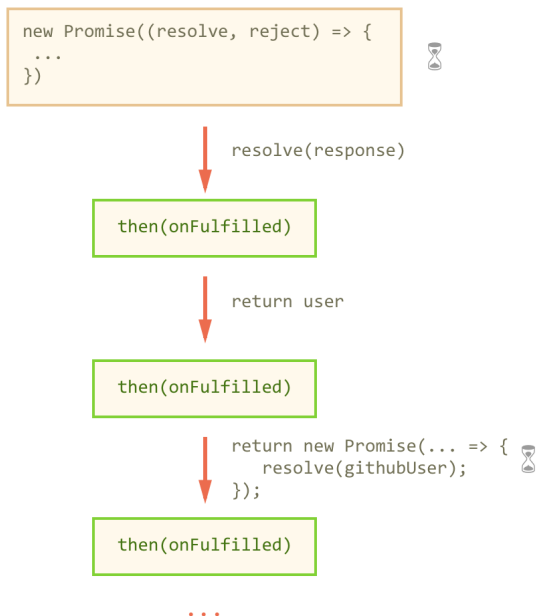
При чейнинге, то есть последовательных вызовах `.then...then...then`, в каждый следующий `then` переходит результат от предыдущего. Вызовы `console.log` оставлены, чтобы при запуске можно было посмотреть конкретные значения, хотя они здесь и не очень важны.

Если очередной `then` вернул промис, то далее по цепочке будет передан не сам этот промис, а его результат.

В коде выше:

1. Функция в первом `then` возвращает «обычное» значение `user`. Это значит, что `then` возвратит промис в состоянии «выполнен» с `user` в качестве результата. Он станет аргументом в следующем `then`.
2. Функция во втором `then` возвращает промис (результат нового вызова `httpGet`). Когда он будет завершён (может пройти какое-то время), то будет вызван следующий `then` с его результатом.
3. Третий `then` ничего не возвращает.

Схематично его работу можно изобразить так:



Значком «песочные часы» помечены периоды ожидания, которых всего два: в исходном `httpGet` и в подвызове далее по цепочке.

Если `then` возвращает промис, то до его выполнения может пройти некоторое время, оставшаяся часть цепочки будет ждать.

То есть, логика довольно проста:

- В каждом `then` мы получаем текущий результат работы.
- Можно его обработать синхронно и вернуть результат (например, применить `JSON.parse`). Или же, если нужна асинхронная обработка – инициировать её и вернуть промис.

Обратим внимание, что последний `then` в нашем примере ничего не возвращает. Если мы хотим, чтобы после `setTimeout (*)` асинхронная цепочка могла быть продолжена, то последний `then` тоже должен вернуть промис. Это общее правило: если внутри `then` стартует новый асинхронный процесс, то для того, чтобы оставшаяся часть цепочки выполнялась после его окончания, мы должны вернуть промис.

В данном случае промис должен перейти в состояние «выполнен» после срабатывания `setTimeout`.

Строку (*) для этого нужно переписать так:

```
.then(githubUser => {
  ...

  // вместо setTimeout(() => img.remove(), 3000); (*)
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      img.remove();
      // после таймаута – вызов resolve,
      // можно без результата, чтобы управление перешло в следующий then
      // (или можно передать данные пользователя дальше по цепочке)
      resolve();
    }, 3000);
  });
});
```

Теперь, если к цепочке добавить ещё `then`, то он будет вызван после окончания `setTimeout`.

Перехват ошибок

Выше мы рассмотрели «идеальный случай» выполнения, когда ошибок нет.

А что, если `github` не отвечает? Или `JSON.parse` бросил синтаксическую ошибку при обработке данных?

Да мало ли, где ошибка...

Правило здесь очень простое.

При возникновении ошибки – она отправляется в ближайший обработчик `onRejected`.

Такой обработчик нужно поставить через второй аргумент `.then(..., onRejected)` или, что то же самое, через `.catch(onRejected)`.

Чтобы поймать всевозможные ошибки, которые возникнут при загрузке и обработке данных, добавим `catch` в конец нашей цепочки:

```
'use strict';

// в httpGet обратимся к несуществующей странице
httpGet('/page-not-exists')
  .then(response => JSON.parse(response))
  .then(user => httpGet(`https://api.github.com/users/${user.name}`))
  .then(githubUser => {
    githubUser = JSON.parse(githubUser);

    let img = new Image();
    img.src = githubUser.avatar_url;
    img.className = "promise-avātar-example";
    document.body.appendChild(img);

    return new Promise((resolve, reject) => {
      setTimeout(() => {
        img.remove();
        resolve();
      }, 3000);
    });
  })
  .catch(error => {
    alert(error); // Error: Not Found
  });
```

В примере выше ошибка возникает в первом же `httpGet`, но `catch` с тем же успехом поймал бы ошибку во втором `httpGet` или в `JSON.parse`.

Принцип очень похож на обычный `try..catch`: мы делаем асинхронную цепочку из `.then`, а затем, когда нужно перехватить ошибки, вызываем `.catch(onRejected)`.

i А что после `catch`?

Обработчик `.catch(onRejected)` получает ошибку и должен обработать её.

Есть два варианта развития событий:

1. Если ошибка не критичная, то `onRejected` возвращает значение через `return`, и управление переходит в ближайший `.then(onFulfilled)`.
2. Если продолжить выполнение с такой ошибкой нельзя, то он делает `throw`, и тогда ошибка переходит в следующий ближайший `.catch(onRejected)`.

Это также похоже на обычный `try..catch` – в блоке `catch` ошибка либо обрабатывается, и тогда выполнение кода продолжается как обычно, либо он делает `throw`. Существенное отличие – в том, что промисы асинхронные, поэтому при отсутствии внешнего `.catch` ошибка не «вываливается» в консоль и не «убивает» скрипт.

Ведь возможно, что новый обработчик `.catch` будет добавлен в цепочку позже.

Промисы в деталях

Самым основным источником информации по промисам является, разумеется, [стандарт](#).

Чтобы наше понимание промисов было полным, и мы могли с лёгкостью разрешать сложные ситуации, посмотрим внимательнее, что такое промис и как он работает, но уже не в общих словах, а детально, в соответствии со стандартом ECMAScript.

Согласно стандарту, у объекта `new Promise(executor)` при создании есть четыре внутренних свойства:

- `PromiseState` – состояние, вначале «pending».
- `PromiseResult` – результат, при создании значения нет.
- `PromiseFulfillReactions` – список функций-обработчиков успешного выполнения.
- `PromiseRejectReactions` – список функций-обработчиков ошибки.

```
new Promise(executor)
```

```
PromiseState:    "pending"
PromiseResult:   undefined
PromiseFulfillReactions: []
PromiseRejectReactions: []
```

Когда функция-executor вызывает `reject` или `resolve`, то `PromiseState` становится "resolved" или "rejected", а все функции-обработчики из соответствующего списка перемещаются в специальную системную очередь "PromiseJobs".

Эта очередь автоматически выполняется, когда интерпретатору «нечего делать». Иначе говоря, все функции-обработчики выполняются асинхронно, одна за другой, по завершении текущего кода, примерно как `setTimeout(..., 0)`.

Исключение из этого правила – если `resolve` возвращает другой `Promise`. Тогда дальнейшее выполнение ожидает его результата (в очередь помещается специальная задача), и функции-обработчики выполняются уже с ним.

Добавляет обработчики в списки один метод: `.then(onResolved, onRejected)`. Метод `.catch(onRejected)` – всего лишь сокращённая запись `.then(null, onRejected)`.

Он делает следующее:

- Если `PromiseState == "pending"`, то есть промис ещё не выполнен, то обработчики добавляются в соответствующие списки.
- Иначе обработчики сразу помещаются в очередь на выполнение.

Здесь важно, что обработчики можно добавлять в любой момент. Можно до выполнения промиса (они подождут), а можно – после (выполнятся в ближайшее время, через асинхронную очередь).

Например:

```
// Промис выполнится сразу же
var promise = new Promise((resolve, reject) => resolve(1));

// PromiseState = "resolved"
// PromiseResult = 1

// Добавили обработчик к выполненному промису
promise.then(alert); // ...он сработает тут же
```

Разумеется, можно добавлять и много обработчиков на один и тот же промис:

```
// Промис выполнится сразу же
var promise = new Promise((resolve, reject) => resolve(1));

promise.then( function f1(result) {
  alert(result); // 1
  return 'f1';
})

promise.then( function f2(result) {
  alert(result); // 1
  return 'f2';
})
```

Вид объекта `promise` после этого:

```
promise
```

```
PromiseState:    "resolved"
PromiseResult:   1
PromiseFulfillReactions: [f1, f2]
PromiseRejectReactions: [Thrower, Thrower]
```

На этой иллюстрации можно увидеть добавленные нами обработчики `f1`, `f2`, а также – автоматические добавленные обработчики ошибок "Thrower".

Дело в том, что `.then`, если один из обработчиков не указан, добавляет его «от себя», следующим образом:

- Для успешного выполнения – функция `Identity`, которая выглядит как `arg => arg`, то есть возвращает аргумент без изменений.
- Для ошибки – функция `Thrower`, которая выглядит как `arg => throw arg`, то есть генерирует ошибку.

Это, по сути дела, формальность, но без неё некоторые особенности поведения промисов могут «не сойтись» в общую логику, поэтому мы упоминаем о ней здесь.

Обратим внимание, в этом примере намеренно *не используется чейнинг*. То есть, обработчики добавляются именно на один и тот же промис.

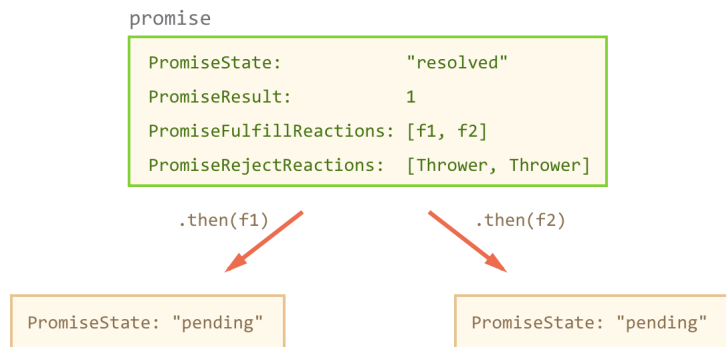
Поэтому оба `alert` выдадут одно значение `1`.

Все функции из списка обработчиков вызываются с результатом промиса, одна за другой. Никакой передачи результатов между обработчиками в рамках одного промиса нет, а сам результат промиса (`PromiseResult`) после установки не меняется.

Поэтому, чтобы продолжить работу с результатом, используется чейнинг.

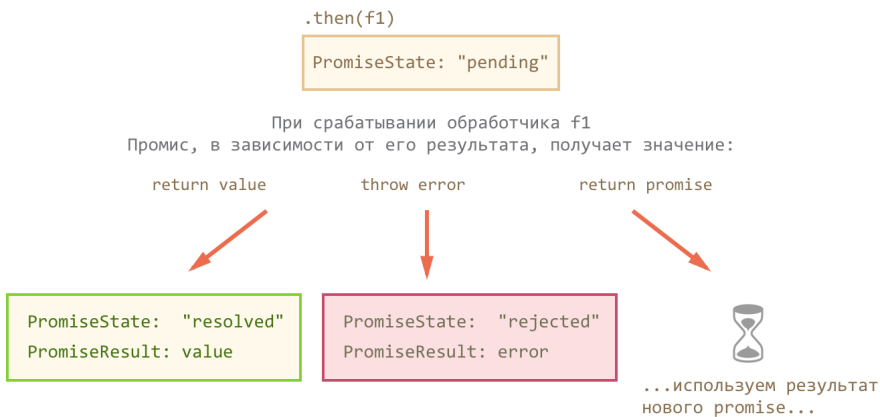
Для того, чтобы результат обработчика передать следующей функции, `.then` создаёт новый промис и возвращает его.

В примере выше создаётся два таких промиса (т.к. два вызова `.then`), каждый из которых даёт свою ветку выполнения:



Изначально эти новые промисы – «пустые», они ждут. Когда в будущем выполнятся обработчики `f1`, `f2`, то их результат будет передан в новые промисы по стандартному принципу:

- Если вернётся обычное значение (не промис), новый промис перейдёт в `"resolved"` с ним.
- Если был `throw`, то новый промис перейдёт в состояние `"rejected"` с ошибкой.
- Если вернётся промис, то используем его результат (он может быть как `resolved`, так и `rejected`).

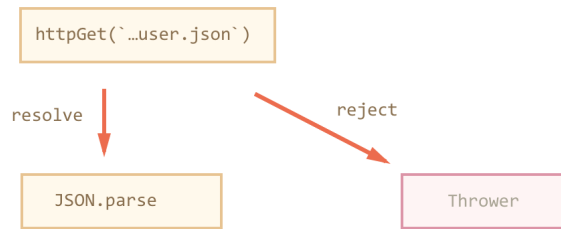


Дальше выполнятся уже обработчики на новом промисе, и так далее.

Чтобы лучше понять происходящее, посмотрим на цепочку, которая получается в процессе написания кода для показа github-аватара.

Первый промис и обработка его результата:

```
httpGet('/article/promise/user.json')
  .then(JSON.parse)
```



Если промис завершился через `resolve`, то результат – в `JSON.parse`, если `reject` – то в `Thrower`.

Как было сказано выше, `Thrower` – это стандартная внутренняя функция, которая автоматически используется, если второй обработчик не указан.

Можно считать, что второй обработчик выглядит так:

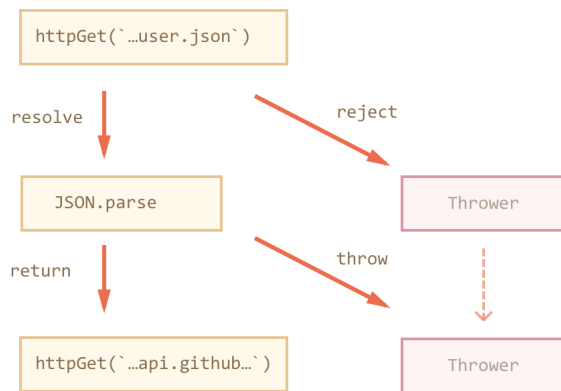
```
httpGet('/article/promise/user.json')
  .then(JSON.parse, err => throw err)
```

Заметим, что когда обработчик в промисах делает `throw` – в данном случае, при ошибке запроса, то такая ошибка не «валит» скрипт и не выводится в консоли. Она просто будет передана в ближайший следующий обработчик `onRejected`.

Добавим в код ещё строку:

```
httpGet('/article/promise/user.json')
  .then(JSON.parse)
  .then(user => httpGet(`https://api.github.com/users/${user.name}`))
```

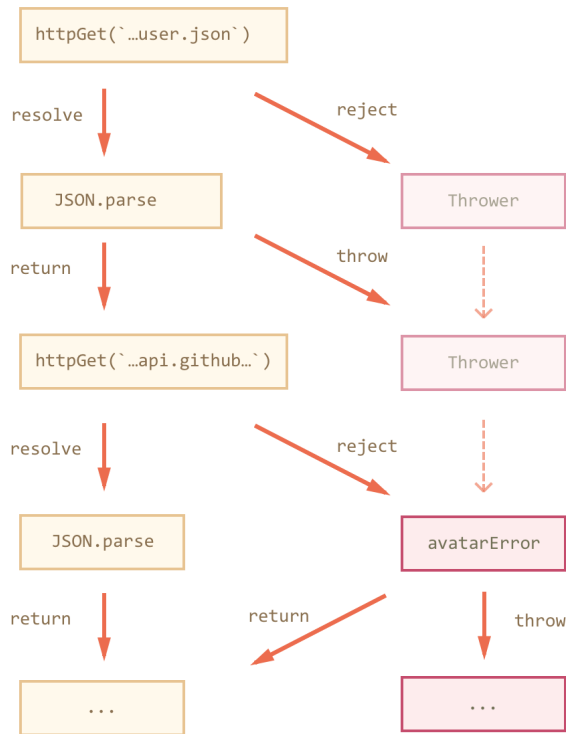
Цепочка «выросла вниз»:



Функция `JSON.parse` либо возвращает объект с данными, либо генерирует ошибку (что расценивается как `reject`).

Если всё хорошо, то `then(user => httpGet(...))` вернёт новый промис, на который стоят уже два обработчика:

```
httpGet('/article/promise/user.json')
  .then(JSON.parse)
  .then(user => httpGet(`https://api.github.com/users/${user.name}`))
  .then(
    JSON.parse,
    function avatarError(error) {
      if (error.code == 404) {
        return {name: "NoGithub", avatar_url: '/article/promise/anon.png'};
      } else {
        throw error;
      }
    }
  )
})
```



Наконец-то хоть какая-то обработка ошибок!

Обработчик `avatarError` перехватит ошибки, которые были ранее. Функция `httpGet` при генерации ошибки записывает её HTTP-код в свойство `error.code`, так что мы легко можем понять – что это:

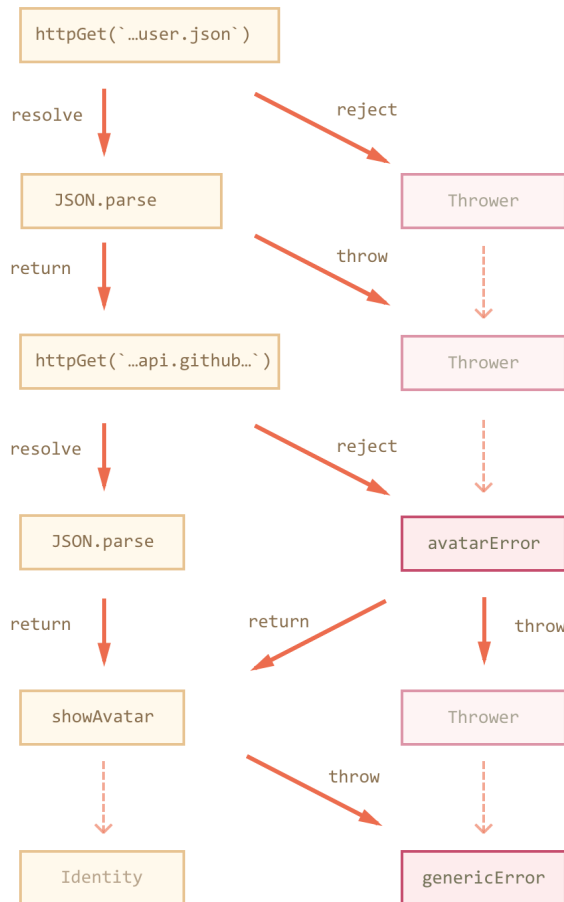
- Если страница на Github не найдена – можно продолжить выполнение, используя «аватар по умолчанию»
- Иначе – пробрасываем ошибку дальше.

Итого, после добавления оставшейся части цепочки, картина получается следующей:

```

'use strict';

httpGet('/article/promise/userNoGithub.json')
  .then(JSON.parse)
  .then(user => loadUrl(`https://api.github.com/users/${user.name}`))
  .then(
    JSON.parse,
    function githubError(error) {
      if (error.code == 404) {
        return {name: "NoGithub", avatar_url: '/article/promise/anon.png'};
      } else {
        throw error;
      }
    }
  )
  .then(function showAvatar(githubUser) {
    let img = new Image();
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.appendChild(img);
    setTimeout(() => img.remove(), 3000);
  })
  .catch(function genericError(error) {
    alert(error); // Error: Not Found
  });
  
```



В конце срабатывает общий обработчик `genericError`, который перехватывает любые ошибки. В данном случае ошибки, которые в него попадут, уже несут критический характер, что-то серьёзно не так. Чтобы посетитель не удивился отсутствию информации, мы показываем ему сообщение об этом.

Можно и как-то иначе вывести уведомление о проблеме, главное – не забыть обработать ошибки в конце. Если последнего `catch` не будет, а цепочка завершится с ошибкой, то посетитель об этом не узнает.

В консоли тоже ничего не будет, так как ошибка остаётся «внутри» промиса, ожидая добавления следующего обработчика `onRejected`, которому будет передана.

Итак, мы рассмотрели основные приёмы использования промисов. Далее – посмотрим некоторые полезные вспомогательные методы.

Параллельное выполнение

Что, если мы хотим осуществить несколько асинхронных процессов одновременно и обработать их результат?

В классе `Promise` есть следующие статические методы.

`Promise.all(iterable)`

Вызов `Promise.all(iterable)` получает массив (или другой итерируемый объект) промисов и возвращает промис, который ждёт, пока все переданные промисы завершатся, и переходит в состояние «выполнено» с массивом их результатов.

Например:

```

Promise.all([
  httpGet('/article/promise/user.json'),
  httpGet('/article/promise/guest.json')
]).then(results => {
  alert(results);
});
  
```

Допустим, у нас есть массив с URL.

```

let urls = [
  '/article/promise/user.json',
  '/article/promise/guest.json'
];
  
```

Чтобы загрузить их параллельно, нужно:

1. Создать для каждого URL соответствующий промис.
2. Обернуть массив таких промисов в `Promise.all`.

Получится так:

```
'use strict';

let urls = [
  '/article/promise/user.json',
  '/article/promise/guest.json'
];

Promise.all( urls.map(httpGet) )
  .then(results => {
    alert(results);
  });
```

Заметим, что если какой-то из промисов завершился с ошибкой, то результатом `Promise.all` будет эта ошибка. При этом остальные промисы игнорируются.

Например:

```
Promise.all([
  httpGet('/article/promise/user.json'),
  httpGet('/article/promise/guest.json'),
  httpGet('/article/promise/no-such-page.json') // (нет такой страницы)
]).then(
  result => alert("не работает"),
  error => alert("Ошибка: " + error.message) // Ошибка: Not Found
);
```

Promise.race(iterable)

Вызов `Promise.race`, как и `Promise.all`, получает итерируемый объект с промисами, которые нужно выполнить, и возвращает новый промис.

Но, в отличие от `Promise.all`, результатом будет только первый успешно выполнившийся промис из списка. Остальные игнорируются.

Например:

```
Promise.race([
  httpGet('/article/promise/user.json'),
  httpGet('/article/promise/guest.json')
]).then(firstResult => {
  firstResult = JSON.parse(firstResult);
  alert( firstResult.name ); // iliakan или guest, смотря что загрузится раньше
});
```

Promise.resolve(value)

Вызов `Promise.resolve(value)` создаёт успешно выполнившийся промис с результатом `value`.

Он аналогичен конструкции:

```
new Promise((resolve) => resolve(value))
```

`Promise.resolve` используют, когда хотят построить асинхронную цепочку, и начальный результат уже есть.

Например:

```
Promise.resolve(window.location) // начать с этого значения
  .then(httpGet) // вызвать для него httpGet
  .then(alert) // и вывести результат
```

Promise.reject(error)

Аналогично `Promise.resolve(value)` создаёт уже выполнившийся промис, но не с успешным результатом, а с ошибкой `error`.

Например:

```
Promise.reject(new Error("..."))
  .catch(alert) // Error: ...
```

Метод `Promise.reject` используется очень редко, гораздо реже чем `resolve`, потому что ошибка возникает обычно не в начале цепочки, а в процессе её выполнения.

Итого

- Промис – это специальный объект, который хранит своё состояние, текущий результат (если есть) и коллбэки.
- При создании `new Promise((resolve, reject) => ...)` автоматически запускается функция-аргумент, которая должна вызвать `resolve(result)` при успешном выполнении и `reject(error)` – при ошибке.
- Аргумент `resolve/reject` (только первый, остальные игнорируются) передаётся обработчикам на этом промисе.
- Обработчики назначаются вызовом `.then/catch`.
- Для передачи результата от одного обработчика к другому используется чейнинг.

У промисов есть некоторые ограничения. В частности, стандарт не предусматривает какой-то метод для «отмены» промиса, хотя в ряде ситуаций (http-запросы) это было бы довольно удобно. Возможно, он появится в следующей версии стандарта JavaScript.

В современной JavaScript-разработке сложные цепочки с промисами используются редко, так как они куда проще описываются при помощи генераторов с библиотекой `co`, которые рассмотрены в [соответствующей главе](#). Можно сказать, что промисы лежат в основе более продвинутых способов асинхронной разработки.

✔ Задачи

Промисифицировать `setTimeout`

Напишите функцию `delay(ms)`, которая возвращает промис, переходящий в состояние "resolved" через `ms` миллисекунд.

Пример использования:

```
delay(1000)
  .then(() => alert("Hello!"))
```

Такая функция полезна для использования в других промис-цепочках.

Вот такой вызов:

```
return new Promise((resolve, reject) => {
  setTimeout(() => {
    doSomething();
    resolve();
  }, ms)
});
```

Станет возможным переписать так:

```
return delay(ms).then(doSomething);
```

[К решению](#)

Загрузить массив последовательно

Есть массив URL:

```
'use strict';

let urls = [
  'user.json',
  'guest.json'
];
```

Напишите код, который все URL из этого массива загружает – один за другим (последовательно), и сохраняет в результаты в массиве `results`, а потом выводит.

Вариант с параллельной загрузкой выглядел бы так:

```
Promise.all(urls.map(httpGet) )
  .then(alert);
```

В этой задаче загрузку нужно реализовать последовательно.

[Открыть песочницу для задачи.](#) ↗

[К решению](#)

Генераторы

Генераторы – новый вид функций в современном JavaScript. Они отличаются от обычных тем, что могут приостанавливать своё выполнение, возвращать промежуточный результат и далее возобновлять его позже, в произвольный момент времени.

Создание генератора

Для объявления генератора используется новая синтаксическая конструкция: `function*` (функция со звёздочкой).

Её называют «функция-генератор» (generator function).

Выглядит это так:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

При запуске `generateSequence()` код такой функции не выполняется. Вместо этого она возвращает специальный объект, который как раз и называют «генератором».

```
// generator function создаёт generator
let generator = generateSequence();
```

Правильнее всего будет воспринимать генератор как «замороженный вызов функции»:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

При создании генератора код находится в начале своего выполнения.

Основным методом генератора является `next()`. При вызове он возобновляет выполнение кода до ближайшего ключевого слова `yield`. По достижении `yield` выполнение приостанавливается, а значение – возвращается во внешний код:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

```
let generator = generateSequence();
```

```
let one = generator.next();
```

```
alert(JSON.stringify(one)); // {value: 1, done: false}
```

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

Повторный вызов `generator.next()` возобновит выполнение и вернёт результат следующего `yield`:

```
let two = generator.next();
```

```
alert(JSON.stringify(two)); // {value: 2, done: false}
```

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

И, наконец, последний вызов завершит выполнение функции и вернёт результат `return`:

```
let three = generator.next();
```

```
alert(JSON.stringify(three)); // {value: 3, done: true}
```

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

Функция завершена. Внешний код должен увидеть это из свойства `done:true` и обработать `value:3`, как окончательный результат.

Новые вызовы `generator.next()` больше не имеют смысла. Впрочем, если они и будут, то не вызовут ошибки, но будут возвращать один и тот же объект: `{done: true}`.

«Открыть назад» завершившийся генератор нельзя, но можно создать новый ещё одним вызовом `generateSequence()` и выполнить его.

i `function* (...)` или `function *(...)` ?

Можно ставить звёздочку как сразу после `function`, так и позже, перед названием. В интернете можно найти обе эти формы записи, они верны:

```
function* f() {  
  // звёздочка после function  
}  
  
function *f() {  
  // звёздочка перед названием  
}
```

Технически, нет разницы, но писать то так то эдак – довольно странно, надо остановиться на чём-то одном.

Автор этого текста полагает, что правильнее использовать первый вариант `function*`, так как звёздочка относится к типу объявляемой сущности (`function*` – «функция-генератор»), а не к её названию. Конечно, это всего лишь рекомендация-мнение, не обязательное к выполнению, работать будет в любом случае.

Генератор – итератор

Как вы, наверно, уже догадались по наличию метода `next()`, генератор связан с [итераторами](#). В частности, он является итерируемым объектом.

Его можно перебирать и через `for..of`:

```
function* generateSequence() {  
  yield 1;  
  yield 2;  
  return 3;  
}  
  
let generator = generateSequence();  
  
for(let value of generator) {  
  alert(value); // 1, затем 2  
}
```

Заметим, однако, существенную особенность такого перебора!

При запуске примера выше будет выведено значение 1, затем 2. Значение 3 выведено не будет. Это потому что стандартный перебор итератора игнорирует `value` на последнем значении, при `done: true`. Так что результат `return` в цикле `for..of` не выводится.

Соответственно, если мы хотим, чтобы все значения возвращались при переборе через `for..of`, то надо возвращать их через `yield`:

```
function* generateSequence() {  
  yield 1;  
  yield 2;  
  yield 3;  
}  
  
let generator = generateSequence();  
  
for(let value of generator) {  
  alert(value); // 1, затем 2, затем 3  
}
```

...А зачем вообще `return` при таком раскладе, если его результат игнорируется? Он тоже нужен, но в других ситуациях. Перебор через `for..of` – в некотором смысле «исключение». Как мы увидим дальше, в других контекстах `return` очень даже востребован.

Композиция генераторов

Один генератор может включать в себя другие. Это называется композицией.

Разберём композицию на примере.

Пусть у нас есть функция `generateSequence`, которая генерирует последовательность чисел:

```
function* generateSequence(start, end) {  
  for (let i = start; i <= end; i++) {  
    yield i;  
  }  
}  
  
// Используем оператор ... для преобразования итерируемого объекта в массив  
let sequence = [...generateSequence(2,5)];  
  
alert(sequence); // 2, 3, 4, 5
```

Мы хотим на её основе сделать другую функцию `generateAlphaNumCodes()`, которая будет генерировать коды для буквенно-цифровых символов латинского алфавита:

- 48..57 – для 0..9
- 65..90 – для A..Z
- 97..122 – для a..z

Далее этот набор кодов можно превратить в строку и использовать, к примеру, для выбора из него случайного пароля. Только символы пунктуации ещё хорошо бы добавить для надёжности, но в этом примере мы будем без них.

Естественно, раз в нашем распоряжении есть готовый генератор `generateSequence`, то хорошо бы его использовать.

Конечно, можно внутри `generateAlphaNum` запустить несколько раз `generateSequence`, объединить результаты и вернуть. Так мы бы сделали с обычными функциями. Но композиция – это кое-что получше.

Она выглядит так:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}

function* generateAlphaNum() {
  // 0..9
  yield* generateSequence(48, 57);

  // A..Z
  yield* generateSequence(65, 90);

  // a..z
  yield* generateSequence(97, 122);
}

let str = '';

for(let code of generateAlphaNum()) {
  str += String.fromCharCode(code);
}

alert(str); // 0..9A..Za..z
```

Здесь использована специальная форма `yield*`. Она применима только к другому генератору и *делегировает* ему выполнение.

То есть, при `yield*` интерпретатор переходит внутрь генератора-аргумента, к примеру, `generateSequence(48, 57)`, выполняет его, и все `yield`, которые он делает, выходят из внешнего генератора.

Получается – как будто мы вставили код внутреннего генератора во внешний напрямую, вот так:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}

function* generateAlphaNum() {
  // yield* generateSequence(48, 57);
  for (let i = 48; i <= 57; i++) yield i;

  // yield* generateSequence(65, 90);
  for (let i = 65; i <= 90; i++) yield i;

  // yield* generateSequence(97, 122);
  for (let i = 97; i <= 122; i++) yield i;
}

let str = '';

for(let code of generateAlphaNum()) {
  str += String.fromCharCode(code);
}

alert(str); // 0..9A..Za..z
```

Код выше по поведению полностью идентичен варианту с `yield*`. При этом, конечно, переменные вложенного генератора не попадают во внешний, «делегирование» только выводит результаты `yield` во внешний поток.

Композиция – это естественное встраивание одного генератора в поток другого. При композиции значения из вложенного генератора выдаются «по мере готовности». Поэтому она будет работать даже если поток данных из вложенного генератора оказался бесконечным или ожидает какого-либо условия для завершения.

`yield` – дорога в обе стороны

До этого генераторы наиболее напоминали «итераторы на стероидах». Но, как мы сейчас увидим, это не так, есть фундаментальное различие, генераторы гораздо мощнее и гибче.

Всё дело в том, что `yield` – дорога в обе стороны: он не только возвращает результат наружу, но и может передавать значение извне в генератор.

Вызов `let result = yield value` делает следующее:

- Возвращает value во внешний код, приостанавливая выполнение генератора.
- Внешний код может обработать значение, и затем вызвать next с аргументом: generator.next(arg) .
- Генератор продолжит выполнение, аргумент next будет возвращён как результат yield (и записан в result) .

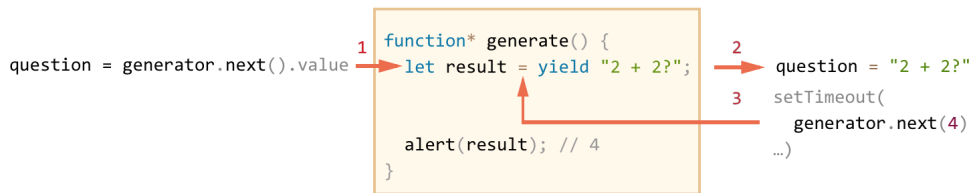
Продемонстрируем это на примере:

```
function* gen() {
  // Передать вопрос во внешний код и подождать ответа
  let result = yield "2 + 2?";
}
alert(result);

let generator = gen();
let question = generator.next().value;
// "2 + 2?"

setTimeout(() => generator.next(4), 2000);
```

На рисунке ниже прямоугольником изображён генератор, а вокруг него – «внешний код», который с ним взаимодействует:



На этой иллюстрации показано то, что происходит в генераторе:

1. Первый вызов `generator.next()` – всегда без аргумента, он начинает выполнение и возвращает результат первого `yield` («2+2?»). На этой точке генератор приостанавливает выполнение.
2. Результат `yield` переходит во внешний код (в `question`). Внешний код может выполнять любые асинхронные задачи, генератор стоит «на паузе».
3. Когда асинхронные задачи готовы, внешний код вызывает `generator.next(4)` с аргументом. Выполнение генератора возобновляется, а 4 выходит из присваивания как результат `let result = yield ...`

В примере выше – только два `next` .

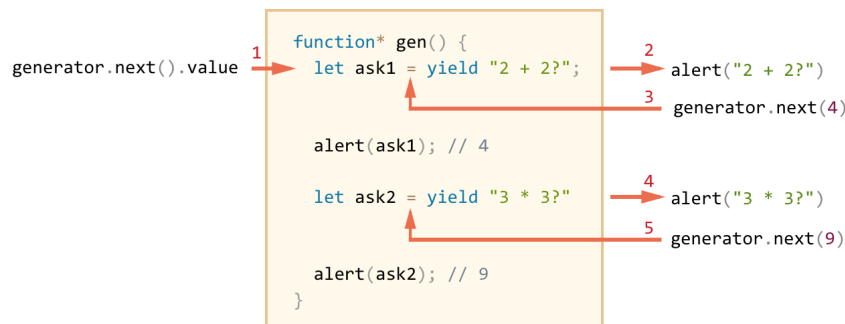
Увеличим их количество, чтобы стал более понятен общий поток выполнения:

```
function* gen() {
  let ask1 = yield "Сколько будет 2 + 2?";
  alert(ask1); // 4

  let ask2 = yield "3 * 3?"
  alert(ask2); // 9
}

let generator = gen();
alert( generator.next().value ); // "2 + 2?"
alert( generator.next(4).value ); // "3 * 3?"
alert( generator.next(9).done ); // true
```

Взаимодействие с внешним кодом:



1. Первый `.next()` начинает выполнение... Оно доходит до первого `yield` .
2. Результат возвращается во внешний код.
3. Второй `.next(4)` передаёт 4 обратно в генератор как результат первого `yield` и возобновляет выполнение.
4. ...Оно доходит до второго `yield` , который станет результатом `.next(4)` .

5. Третий `next(9)` передаёт 9 в генератор как результат второго `yield` и возобновляет выполнение, которое завершается окончанием функции, так что `done: true`.

Получается «пинг-понг»: каждый `next(value)` передаёт в генератор значение, которое становится результатом текущего `yield`, возобновляет выполнение и получает выражение из следующего `yield`. Исключением является первый вызов `next`, который не может передать значение в генератор, т.к. ещё не было ни одного `yield`.

generator.throw

Как мы видели в примерах выше, внешний код может вернуть генератору в качестве результата `yield` любое значение.

...Но «вернуть» можно не только результат, но и ошибку!

Для того, чтобы передать в `yield` ошибку, используется вызов `generator.throw(err)`. При этом на строке с `yield` возникает исключение.

Например, в коде ниже обращение к внешнему коду `yield` "Сколько будет 2 + 2" завершится с ошибкой:

```
function* gen() {
  try {
    // в этой строке возникнет ошибка
    let result = yield "Сколько будет 2 + 2?"; // (**)

    alert("выше будет исключение ^^^");
  } catch(e) {
    alert(e); // выведет ошибку
  }
}

let generator = gen();

let question = generator.next().value;

generator.throw(new Error("ответ не найден в моей базе данных")); // (*)
```

«Вброшенная» в строке (*) ошибка возникает в строке с `yield (**) .` Далее она обрабатывается как обычно. В примере выше она перехватывается `try..catch` и выводится.

Если ошибку не перехватить, то она «выпадет» из генератора. По стеку ближайший вызов, который инициировал выполнение – это строка с `.throw`. Можно перехватить её там, как и продемонстрировано в примере ниже:

```
function* gen() {
  // В этой строке возникнет ошибка
  let result = yield "Сколько будет 2 + 2?";
}

let generator = gen();

let question = generator.next().value;

try {
  generator.throw(new Error("ответ не найден в моей базе данных"));
} catch(e) {
  alert(e); // выведет ошибку
}
```

Если же ошибка и там не перехвачена, то дальше – как обычно, либо `try..catch` снаружи, либо она «повалит» скрипт.

Плоский асинхронный код

Одна из основных областей применения генераторов – написание «плоского» асинхронного кода.

Общий принцип такой:

- Генератор `yield`'ит не просто значения, а промисы.
- Есть специальная «функция-чернорабочий» `execute(generator)` которая запускает генератор, последовательными вызовами `next` получает из него промисы – один за другим, и, когда очередной промис выполнится, возвращает его результат в генератор следующим `next`.
- Последнее значение генератора (`done:true`) `execute` уже обрабатывает как окончательный результат – например, возвращает через промис куда-то ещё, во внешний код или просто использует, как в примере ниже.

Напишем такой код для получения аватара пользователя с `github` и его вывода, аналогичный рассмотренному в статье про [промисы](#).

Для AJAX-запросов будем использовать метод `fetch`, он как раз возвращает промисы.

```
// генератор для получения и показа аватара
// он yield'ит промисы
function* showUserAvatar() {

  let userFetch = yield fetch('/article/generator/user.json');
  let userInfo = yield userFetch.json();

  let githubFetch = yield fetch(`https://api.github.com/users/${userInfo.name}`);
  let githubUserInfo = yield githubFetch.json();

  let img = new Image();
  img.src = githubUserInfo.avatar_url;
  img.className = "promise-avatar-example";
  document.body.appendChild(img);
}
```

```

yield new Promise(resolve => setTimeout(resolve, 3000));

img.remove();

return img.src;
}

// вспомогательная функция-чернорабочий
// для выполнения промисов из generator
function execute(generator, yieldValue) {

  let next = generator.next(yieldValue);

  if (!next.done) {
    next.value.then(
      result => execute(generator, result),
      err => generator.throw(err)
    );
  } else {
    // обрабатываем результат return из генератора
    // обычно здесь вызов callback или что-то в этом духе
    alert(next.value);
  }
}

execute( showUserAvatar() );

```

Функция `execute` в примере выше – универсальная, она может работать с любым генератором, который `yield`'ит промисы.

Вместе с тем, это – всего лишь набросок, чтобы было понятно, как такая функция в принципе работает. Есть уже готовые реализации, обладающие большим количеством возможностей.

Одна из самых известных – это библиотека [co](#), которую мы рассмотрим далее.

Библиотека «co»

Библиотека `co`, как и `execute` в примере выше, получает генератор и выполняет его.

Начнём сразу с примера, а потом – детали и полезные возможности:

```

co(function*() {

  let result = yield new Promise(
    resolve => setTimeout(resolve, 1000, 1)
  );

  alert(result); // 1

})

```

Предполагается, что библиотека `co` подключена к странице, например, отсюда: <http://cdnjs.com/libraries/co/>. В примере выше `function*()` делает `yield` промиса с `setTimeout`, который через секунду возвращает `1`.

Вызов `co(...)` возвращает промис с результатом генератора. Если в примере выше `function*()` что-то возвратит, то это можно будет получить через `.then` в результате `co`:

```

co(function*() {

  let result = yield new Promise(
    resolve => setTimeout(resolve, 1000, 1)
  );

  return result; // return 1

}).then(alert); // 1

```

⚠ Обязательно нужен `catch`

Частая ошибка начинающих – вообще забывать про обработку результата `co`. Даже если результата нет, ошибки нужно обработать через `catch`, иначе они «подвиснут» в промисе.

Такой код ничего не выведет:

```
co(function*() {
  throw new Error("Sorry that happened");
})
```

Программист даже не узнает об ошибке. Особенно обидно, когда это опечатка или другая программная ошибка, которую обязательно нужно поправить.

Правильный вариант:

```
co(function*() {
  throw new Error("Sorry that happened");
}).catch(alert); // обработать ошибку как-либо
```

Большинство примеров этого `catch` не содержат, но это лишь потому, что в примерах ошибок нет. А в реальном коде обязательно нужен `catch`.

Библиотека `co` умеет выполнять не только промисы. Есть несколько видов значений, которые можно `yield`, и их обработает `co`:

- Промис.
- Объект-генератор.
- Функция-генератор `function*()` – `co` её выполнит, затем выполнит полученный генератор.
- Функция с единственным аргументом вида `function(callback)` – библиотека `co` её запустит со своей функцией- `callback` и будет ожидать, что при ошибке она вызовет `callback(err)`, а при успешном выполнении – `callback(null, result)`. То есть, в первом аргументе – будет ошибка (если есть), а втором – результат (если нет ошибки). После чего результат будет передан в генератор.
- Массив или объект из вышеперечисленного. При этом все задачи будут выполнены параллельно, и результат, в той же структуре, будет выдан наружу.

В примере ниже происходит `yield` всех этих видов значений. Библиотека `co` обеспечивает их выполнение и возврат результата в генератор:

```
Object.defineProperty(window, 'result', {
  // присвоение result=... будет выводить значение
  set: value => alert(JSON.stringify(value))
});

co(function*() {
  result = yield function*() { // генератор
    return 1;
  }();

  result = yield function*() { // функция-генератор
    return 2;
  };

  result = yield Promise.resolve(3); // промис

  result = yield function(callback) { // function(callback)
    setTimeout(() => callback(null, 4), 1000);
  };

  result = yield { // две задачи выполнит параллельно, как Promise.all
    one: Promise.resolve(1),
    two: function*() { return 2; }
  };

  result = yield [ // две задачи выполнит параллельно, как Promise.all
    Promise.resolve(1),
    function*() { return 2 }
  ];
});
```

📌 Устаревший `yield function(callback)`

Отдельно заметим вариант с `yield function(callback)`. Такие функции, с единственным-аргументом `callback`'ом, в англоязычной литературе называют «thunk».

Функция обязана выполниться и вызвать (асинхронно) либо `callback(err)` с ошибкой, либо `callback(null, result)` с результатом.

Использование таких функций в `yield` является устаревшим подходом, так как там, где можно использовать `yield function(callback)`, можно использовать и промисы. При этом промисы мощнее. Но в старом коде его ещё можно встретить.

Посмотрим пример посложнее, с композицией генераторов:

```
co(function*() {
  let result = yield* gen();
  alert(result); // hello
});
```

```

function* gen() {
  return yield* gen2();
}

function* gen2() {
  let result = yield new Promise( // (1)
    resolve => setTimeout(resolve, 1000, 'hello')
  );
  return result;
}

```

Это – отличный вариант для библиотеки `co`. Композиция `yield* gen()` вызывает `gen()` в потоке внешнего генератора. Аналогично делает и `yield* gen()`.

Поэтому `yield new Promise` из строки (1) в `gen2()` попадает напрямую в библиотеку `co`, как если бы он был сделан так:

```

co(function*() {
  // gen() и затем gen2 встраиваются во внешний генератор
  let result = yield new Promise(
    resolve => setTimeout(resolve, 1000, 'hello')
  );
  alert(result); // hello
});

```

Пример `showUserAvatar()` можно переписать с использованием `co` вот так:

```

// Загрузить данные пользователя с нашего сервера
function* fetchUser(url) {
  let userFetch = yield fetch(url);

  let user = yield userFetch.json();

  return user;
}

// Загрузить профиль пользователя с github
function* fetchGithubUser(user) {
  let githubFetch = yield fetch(`https://api.github.com/users/${user.name}`);
  let githubUser = yield githubFetch.json();

  return githubUser;
}

// Подождать ms миллисекунд
function sleep(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

// Использовать функции выше для получения аватара пользователя
function* fetchAvatar(url) {

  let user = yield* fetchUser(url);

  let githubUser = yield* fetchGithubUser(user);

  return githubUser.avatar_url;
}

// Использовать функции выше для получения и показа аватара
function* showUserAvatar() {

  let avatarUrl;

  try {
    avatarUrl = yield* fetchAvatar('/article/generator/user.json');
  } catch(e) {
    avatarUrl = '/article/generator/anon.png';
  }

  let img = new Image();
  img.src = avatarUrl;
  img.className = "promise-avatar-example";
  document.body.appendChild(img);

  yield sleep(2000);

  img.remove();

  return img.src;
}

co(showUserAvatar);

```

Заметим, что для перехвата ошибок при получении аватара используется `try..catch` вокруг `yield* fetchAvatar`:

```

try {
  avatarUrl = yield* fetchAvatar('/article/generator/user.json');
} catch(e) {
  avatarUrl = '/article/generator/anon.png';
}

```

Это – одно из главных удобств использования генераторов. Несмотря на то, что операции `fetch` – асинхронные, мы можем использовать обычный `try..catch` для обработки ошибок в них.

Для генераторов – только `yield*`

Библиотека `co` технически позволяет писать код так:

```
let user = yield fetchUser(url);
// вместо
// let user = yield* fetchUser(url);
```

То есть, можно сделать `yield` генератора, `co()` его выполнит и передаст значение обратно. Как мы видели выше, библиотека `co` – довольно всеядна. Однако, рекомендуется использовать для вызова функций-генераторов именно `yield*`.

Причин для этого несколько:

1. Делегирование генераторов `yield*` – это встроенный механизм JavaScript. Вместо возвращения значения обратно в `co`, выполнения кода библиотеки... Мы просто используем возможности языка. Это правильнее.
2. Поскольку не происходит лишних вызовов, это быстрее по производительности.
3. И, наконец, пожалуй, самое приятное – делегирование генераторов сохраняет стек.

Проиллюстрируем последнее на примере:

```
co(function*() {
  // при запуске в стеке не будет видно этой строки
  yield g(); // (*)
}).catch(function(err) {
  alert(err.stack);
});

function* g() {
  throw new Error("my error");
}
```

При запуске этого кода стек может выглядеть примерно так:

```
at g (eval at runJS ..., <anonymous>:13:9)
at GeneratorFunctionPrototype.next (native)
at onFulfilled (../co/./index.min.js:1:1136)
at ../co/./index.min.js:1:1076
at co (../co/./index.min.js:1:1039)
at toPromise (../co/./index.min.js:1:1740)
at next (../co/./index.min.js:1:1351)
at onFulfilled (../co/./index.min.js:1:1172)
at ../co/./index.min.js:1:1076
at co (../co/./index.min.js:1:1039)
```

Детали здесь не имеют значения, самое важное – почти весь стек находится внутри библиотеки `co`.

Из оригинального скрипта там только одна строка (первая):

```
at g (eval at runJS ..., <anonymous>:13:9)
```

То есть, стек говорит, что ошибка возникла в строке 13:

```
// строка 13 из кода выше
throw new Error("my error");
```

Что ж, спасибо. Но как мы оказались на этой строке? Об этом в стеке нет ни слова!

Заменяем в строке (*) вызов `yield` на `yield*`:

```
co(function*() {
  // заменили yield на yield*
  yield* g(); // (*)
}).catch(function(err) {
  alert(err.stack);
});

function* g() {
  throw new Error("my error");
}
```

Пример стека теперь:

```
at g (eval at runJS ..., <anonymous>:13:9)
at GeneratorFunctionPrototype.next (native)
at eval (eval at runJS ..., <anonymous>:6:10)
at GeneratorFunctionPrototype.next (native)
at onFulfilled (../co/./index.min.js:1:1136)
at ../co/./index.min.js:1:1076
at co (../co/./index.min.js:1:1039)
at eval (eval at runJS ..., <anonymous>:3:1)
at eval (native)
at runJS (...)
```


Если очистить от вспомогательных вызовов, то эти строки – как раз то, что нам надо:

```
at g (eval at runJS ..., <anonymous>:13:9)
at eval (eval at runJS ..., <anonymous>:6:10)
at eval (eval at runJS ..., <anonymous>:3:1)
```

Теперь видно, что (читаем снизу) исходный вызов был в строке 3, далее – вложенный в строке 6, и затем произошла ошибка в строке 13.

Почему вариант с простым `yield` не работает – достаточно очевидно, если внимательно посмотреть на код и воспроизвести в уме, как он функционирует. Оставляем это упражнение вдумчивому читателю.

Итого, рекомендация уже достаточно обоснована – при запуске вложенных генераторов используем `yield*`.

Итого

- Генераторы создаются при помощи функций-генераторов `function*(...) {...}`.
- Внутри генераторов и только внутри них разрешён оператор `yield`. Это иногда создаёт неудобства, поскольку в коллбэках `.map/.forEach` сделать `yield` нельзя. Впрочем, можно сделать `yield` массива (при использовании `co`).
- Внешний код и генератор обмениваются промежуточными результатами посредством вызовов `next/yield`.
- Генераторы позволяют писать плоский асинхронный код, при помощи библиотеки `co`.

Что касается кросс-браузерной поддержки – она стремительно приближается. Пока же можно использовать генераторы вместе с [Babel](#).

Модули

Концепция модулей как способа организации JavaScript-кода существовала давно.

Когда приложение сложное и кода много – мы пытаемся разбить его на файлы. В каждом файле описываем какую-то часть, а в дальнейшем – собираем эти части воедино.

Модули в стандарте ECMAScript предоставляют удобные средства для этого.

Такие средства предлагались сообществом и ранее, например:

- [AMD](#) – одна из самых древних систем организации модулей, требует лишь наличия клиентской библиотеки, к примеру, [require.js](#), но поддерживается и серверными средствами.
- [CommonJS](#) – система модулей, встроенная в сервер Node.JS. Требует поддержки на клиентской и серверной стороне.
- [UMD](#) – система модулей, которая предложена в качестве универсальной. UMD-модули будут работать и в системе AMD и в CommonJS.

Все перечисленные выше системы требуют различных библиотек или систем сборки для использования.

Новый стандарт отличается от них прежде всего тем, что это – стандарт. А значит, со временем, будет поддерживаться браузерами без дополнительных утилит.

Однако, сейчас браузерной поддержки почти нет. Поэтому ES-модули используются в сочетании с системами сборки, такими как [webpack](#), [brunch](#) и другими, при подключённом [Babel.JS](#). Мы рассмотрим это далее.

Что такое модуль?

Модулем считается файл с кодом.

В этом файле ключевым словом `export` помечаются переменные и функции, которые могут быть использованы снаружи.

Другие модули могут подключать их через вызов `import`.

export

Ключевое слово `export` можно ставить:

- перед объявлением переменных, функций и классов.
- отдельно, при этом в фигурных скобках указывается, что именно экспортируется.

Например, так экспортируется переменная `one`:

```
// экспорт прямо перед объявлением
export let one = 1;
```

Можно написать `export` и отдельно от объявления:

```
let two = 2;
export {two};
```

При этом в фигурных скобках указываются одна или несколько экспортируемых переменных.

Для двух переменных будет так:

```
export {one, two};
```

При помощи ключевого слова `as` можно указать, что переменная `one` будет доступна снаружи (экспортирована) под именем `once`, а `two` – под именем `twice`:

```
export {one as once, two as twice};
```

Экспорт функций и классов выглядит так же:

```
export class User {
  constructor(name) {
    this.name = name;
  }
};

export function sayHi() {
  alert("Hello!");
};

// отдельно от объявлений было бы так:
// export {User, sayHi}
```

i Для экспорта обязательно нужно имя

Заметим, что и у функции и у класса при таком экспорте должно быть имя.

Так будет ошибка:

```
// функция без имени
export function() { alert("Error"); };
```

В экспорте указываются именно имена, а не произвольные выражения.

import

Другие модули могут подключать экспортированные значения при помощи ключевого слова `import`.

Синтаксис:

```
import {one, two} from "./nums";
```

Здесь:

- `./nums` – модуль, как правило это путь к файлу модуля.
- `one`, `two` – импортируемые переменные, которые должны быть обозначены в `nums` словом `export`.

В результате импорта появятся локальные переменные `one`, `two`, которые будут содержать значения соответствующих экспортов.

Например, при таком файле `nums.js`:

```
export let one = 1;
export let two = 2;
```

Модуль ниже выведет «1 and 2»:

```
import {one, two} from "./nums";
alert( `${one} and ${two}` ); // 1 and 2
```

Импортировать можно и под другим именем, указав его в «`as`»:

```
// импорт one под именем item1, а two – под именем item2
import {one as item1, two as item2} from "./nums";
alert( `${item1} and ${item2}` ); // 1 and 2
```

i Импорт всех значений в виде объекта

Можно импортировать все значения сразу в виде объекта вызовом `import * as obj`, например:

```
import * as numbers from './nums';

// теперь экспортированные переменные - свойства numbers
alert( `${numbers.one} and ${numbers.two}` ); // 1 and 2
```

export default

Выше мы видели, что модуль может экспортировать выбранные переменные при помощи `export`.

Однако, как правило, код стараются организовать так, чтобы каждый модуль делал одну вещь. Иначе говоря, «один файл – одна сущность, которую он описывает». Например, файл `user.js` содержит `class User`, файл `login.js` – функцию `login()` для авторизации, и т.п.

При этом модули, разумеется, будут использовать друг друга. Например, `login.js`, скорее всего, будет импортировать класс `User` из модуля `user.js`.

Для такой ситуации, когда один модуль экспортирует одно значение, предусмотрено особое ключевое сочетание `export default`.

Если поставить после `export` слово `default`, то значение станет «экспортом по умолчанию».

Такое значение можно импортировать без фигурных скобок.

Например, файл `user.js`:

```
export default class User {
  constructor(name) {
    this.name = name;
  }
};
```

...А в файле `login.js`:

```
import User from './user';
new User("Бася");
```

«Экспорт по умолчанию» – своего рода «синтаксический сахар». Можно было бы и без него, импортировать значение обычным образом через фигурные скобки `{...}`. Если бы в `user.js` не было `default`, то в `login.js` необходимо было бы указать фигурные скобки:

```
// если бы user.js содержал
// export class User { ... }

// ...то при импорте User понадобились бы фигурные скобки:
import {User} from './user';

new User("Бася");
```

На практике этот «сахар» весьма приятен, так как позволяет легко видеть, какое именно значение экспортирует модуль, а также обойтись без лишних символов при импорте.

CommonJS

Если вы раньше работали с Node.JS или использовали систему сборки в синтаксисе CommonJS, то вот соответствия.

Для экспорта по умолчанию вместо:

```
module.exports = VARIABLE;
```

Пишем:

```
export default VARIABLE;
```

А при импорте из такого модуля вместо:

```
let VARIABLE = require('./file');
```

Пишем:

```
import VARIABLE from './file';
```

Для экспорта нескольких значений из модуля, вместо:

```
exports.NAME = VARIABLE;
```

Пишем в фигурных скобках, что надо экспортировать и под каким именем (без `as`, если имя совпадает):

```
export {VARIABLE as NAME};
```

При импорте – также фигурные скобки:

```
import {NAME} from './file';
```

Использование

Современный стандарт ECMAScript описывает, как импортировать и экспортировать значения из модулей, но он ничего не говорит о том, как эти модули искать, загружать и т.п.

Такие механизмы предлагались в процессе создания стандарта, но были убраны по причине недостаточной проработанности. Возможно, они появятся в будущем.

Сейчас используются системы сборки, как правило, в сочетании с Babel.JS.

Система сборки обрабатывает скрипты, находит в них `import/export` и заменяет их на свои внутренние JavaScript-вызовы. При этом, как правило, много файлов-модулей объединяются в один или несколько скриптов, смотря как указано в конфигурации сборки.

Ниже вы можете увидеть полный пример использования модулей с системой сборки [webpack](#).

В нём есть:

- `nums.js` – модуль, экспортирующий `one` и `two`, как описано выше.
- `main.js` – модуль, который импортирует `one`, `two` из `nums` и выводит их сумму.
- `webpack.config.js` – конфигурация для системы сборки.
- `bundle.js` – файл, который создала система сборки из `main.js` и `nums.js`.
- `index.html` – простой HTML-файл для демонстрации.



Итого

Современный стандарт описывает, как организовать код в модули, экспортировать и импортировать значения.

Экспорт:

- `export` можно поставить прямо перед объявлением функции, класса, переменной.
- Если `export` стоит отдельно от объявления, то значения в нём указываются в фигурных скобках: `export {...}`.
- Также можно экспортировать «значение по умолчанию» при помощи `export default`.

Импорт:

- В фигурных скобках указываются значения, а затем – модуль, откуда их брать: `import {a, b, c as d} from "module"`.
- Можно импортировать все значения в виде объекта при помощи `import * as obj from "module"`.
- Без фигурных скобок будет импортировано «значение по умолчанию»: `import User from "user"`.

На текущий момент модули требуют системы сборки на сервере. Автор этого текста преимущественно использует `webpack`, но есть и другие варианты.

Прoxy

Прокси (проxy) – особый объект, смысл которого – перехватывать обращения к другому объекту и, при необходимости, модифицировать их.

Синтаксис:

```
let proxy = new Proxy(target, handler)
```

Здесь:

- `target` – объект, обращения к которому надо перехватывать.
- `handler` – объект с «ловушками»: функциями-перехватчиками для операций к `target`.

Почти любая операция может быть перехвачена и обработана прокси до или даже вместо доступа к объекту `target`, например: чтение и запись свойств, получение списка свойств, вызов функции (если `target` – функция) и т.п.

Различных типов ловушек довольно много.

Сначала мы подробно рассмотрим самые важные «ловушки», а затем посмотрим и на их полный список.

i Если ловушки нет – операция идёт над **target**
Если для операции нет ловушки, то она выполняется напрямую над **target**.

get/set

Самыми частыми являются ловушки для чтения и записи в объект:

get(target, property, receiver)

Срабатывает при чтении свойства из прокси. Аргументы:

- **target** – целевой объект, тот же который был передан первым аргументом в `new Proxy`.
- **property** – имя свойства.
- **receiver** – объект, к которому было применено присваивание. Обычно сам прокси, либо прототипно наследующий от него. Этот аргумент используется редко.

set(target, property, value, receiver)

Срабатывает при записи свойства в прокси.

Аргументы:

- **target** – целевой объект, тот же который был передан первым аргументом в `new Proxy`.
- **property** – имя свойства.
- **value** – значение свойства.
- **receiver** – объект, к которому было применено присваивание, обычно сам прокси, либо прототипно наследующий от него.

Метод `set` должен вернуть `true`, если присвоение успешно обработано и `false` в случае ошибки (приведёт к генерации `TypeError`).

Пример с выводом всех операций чтения и записи:

```
'use strict';
let user = {};
let proxy = new Proxy(user, {
  get(target, prop) {
    alert(`Чтение ${prop}`);
    return target[prop];
  },
  set(target, prop, value) {
    alert(`Запись ${prop} ${value}`);
    target[prop] = value;
    return true;
  }
});
proxy.firstName = "Ilya"; // запись
proxy.firstName; // чтение
alert(user.firstName); // Ilya
```

При каждой операции чтения и записи свойств `proxy` в коде выше срабатывают методы `get/set`. Через них значение в конечном счёте попадает в объект (или считывается из него).

Можно сделать и позаковыристее.

Методы `get/set` позволяют реализовать доступ к произвольным свойствам, которых в объекте нет.

Например, в коде ниже словарь `dictionary` содержит различные фразы:

```
'use strict';
let dictionary = {
  'Hello': 'Привет',
  'Bye': 'Пока'
};
alert( dictionary['Hello'] ); // Привет
```

А что, если фразы нет? В этом случае будем возвращать фразу без перевода и, на всякий случай, писать об этом в консоль:

```
'use strict';
let dictionary = {
  'Hello': 'Привет',
  'Bye': 'Пока'
};
```

```
dictionary = new Proxy(dictionary, {
  get(target, phrase) {
    if (phrase in target) {
      return target[phrase];
    } else {
      console.log(`No phrase: ${phrase}`);
      return phrase;
    }
  }
})
```

```
// Обращаемся к произвольным свойствам словаря!
alert( dictionary['Hello'] ); // Привет
alert( dictionary['Welcome'] ); // Welcome (без перевода)
```

Аналогично и перехватчик `set` может организовать работу с произвольными свойствами.

has

Ловушка `has` срабатывает в операторе `in` и некоторых других случаях, когда проверяется наличие свойства.

В примере выше, если проверить наличие свойства `Welcome` в `dictionary`, то оператор `in` вернёт `false`:

```
alert( 'Hello' in dictionary ); // true
alert( 'Welcome' in dictionary ); // false, нет такого свойства
```

Это потому, что для перехвата `in` используется ловушка `has`. При отсутствии ловушки операция производится напрямую над исходным объектом `target`, что и даёт такой результат.

Синтаксис `has` аналогичен `get`.

Вот так `dictionary` будет всегда возвращать `true` для любой `in`-проверки:

```
'use strict';

let dictionary = {
  'Hello': 'Привет'
};

dictionary = new Proxy(dictionary, {
  has(target, phrase) {
    return true;
  }
});

alert("BlaBlaBla" in dictionary); // true
```

deleteProperty

Ловушка `deleteProperty` по синтаксису аналогична `get/has`.

Срабатывает при операции `delete`, должна вернуть `true`, если удаление было успешным.

В примере ниже `delete` не повлияет на исходный объект, так как все операции перехватываются и «аннигилируются» прокси:

```
'use strict';

let dictionary = {
  'Hello': 'Привет'
};

let proxy = new Proxy(dictionary, {
  deleteProperty(target, phrase) {
    return true; // ничего не делаем, но возвращает true
  }
});

// не удалит свойство
delete proxy['Hello'];

alert("Hello" in dictionary); // true

// будет то же самое, что и выше
// так как нет ловушки has, операция in сработает на исходном объекте
alert("Hello" in proxy); // true
```

enumerate

Ловушка `enumerate` перехватывает операции `for...in` и `for...of` по объекту.

Как и до ранее, если ловушки нет, то эти операторы работают с исходным объектом:

```
'use strict';

let obj = {a: 1, b: 1};

let proxy = new Proxy(obj, {});
```

```
// перечисление прокси работает с исходным объектом
for(let prop in proxy) {
  alert(prop); // Выведет свойства obj: a, b
}
```

Если же ловушка `enumerate` есть, то она будет вызвана с единственным аргументом `target` и сможет вернуть [итератор](#) для свойств.

В примере ниже прокси делает так, что итерация идёт по всем свойствам, кроме начинающихся с подчёркивания `_`:

```
'use strict';

let user = {
  name: "Ilya",
  surname: "Kantor",
  _version: 1,
  _secret: 123456
};

let proxy = new Proxy(user, {
  enumerate(target) {
    let props = Object.keys(target).filter(function(prop) {
      return prop[0] != '_';
    });

    return props[Symbol.iterator]();
  }
});

// отфильтрованы свойства, начинающиеся с _
for(let prop in proxy) {
  alert(prop); // Выведет свойства user: name, surname
}
```

Посмотрим внимательно, что происходит внутри `enumerate`:

1. Сначала получаем список интересующих нас свойств в виде массива.
2. Метод должен вернуть [итератор](#) по массиву. Встроенный итератор для массива получаем через вызов `props[Symbol.iterator]()`.

apply

Прокси умеет работать не только с обычными объектами, но и с функциями.

Если аргумент `target` прокси – функция, то становится доступна ловушка `apply` для её вызова.

Метод `apply(target, thisArgument, argumentsList)` получает:

- `target` – исходный объект.
- `thisArgument` – контекст `this` вызова.
- `argumentsList` – аргументы вызова в виде массива.

Она может обработать вызов сама и/или передать его функции.

```
'use strict';

function sum(a, b) {
  return a + b;
}

let proxy = new Proxy(sum, {
  // передаст вызов в target, предварительно сообщив о нём
  apply: function(target, thisArg, argumentsList) {
    alert(`Буду вычислять сумму: ${argumentsList}`);
    return target.apply(thisArg, argumentsList);
  }
});

// Выведет сначала сообщение из прокси,
// а затем уже сумму
alert( proxy(1, 2) );
```

Нечто подобное можно сделать через замыкания. Но прокси может гораздо больше. В том числе и перехватывать вызовы через `new`.

construct

Ловушка `construct(target, argumentsList)` перехватывает вызовы при помощи `new`.

Она получает исходный объект `target` и список аргументов `argumentsList`.

Пример ниже передаёт операцию создания исходному классу или функции-конструктору, выводя сообщение об этом:

```
'use strict';

function User(name, surname) {
  this.name = name;
  this.surname = surname;
}

let UserProxy = new Proxy(User, {
```

```
// передаст вызов new User, предварительно сообщив о нём
construct: function(target, argumentsList) {
  alert(`Запуск new с аргументами: ${argumentsList}`);
  return new target(...argumentsList);
}
});

let user = new UserProxy("Ilya", "Kantor");

alert( user.name ); // Ilya
```

Полный список

Полный список возможных функций-перехватчиков, которые может задавать handler :

- [getPrototypeOf](#) – перехватывает обращение к методу `getPrototypeOf` .
- [setPrototypeOf](#) – перехватывает обращение к методу `setPrototypeOf` .
- [isExtensible](#) – перехватывает обращение к методу `isExtensible` .
- [preventExtensions](#) – перехватывает обращение к методу `preventExtensions` .
- [getOwnPropertyDescriptor](#) – перехватывает обращение к методу `getOwnPropertyDescriptor` .
- [defineProperty](#) – перехватывает обращение к методу `defineProperty` .
- [has](#) – перехватывает проверку существования свойства, которая используется в операторе `in` и в некоторых других методах встроенных объектов.
- [get](#) – перехватывает чтение свойства.
- [set](#) – перехватывает запись свойства.
- [deleteProperty](#) – перехватывает удаление свойства оператором `delete` .
- [enumerate](#) – срабатывает при вызове `for..in` или `for..of` , возвращает итератор для свойств объекта.
- [ownKeys](#) – перехватывает обращения к методу `getOwnPropertyNames` .
- [apply](#) – перехватывает вызовы `target()` .
- [construct](#) – перехватывает вызовы `new target()` .

Каждый перехватчик запускается с `handler` в качестве `this` . Это означает, что `handler` кроме ловушек может содержать и другие полезные свойства и методы.

Каждый перехватчик получает в аргументах `target` и дополнительные параметры в зависимости от типа.

Если перехватчик в `handler` не указан, то операция совершается, как если бы была вызвана прямо на `target` .

Итого

Прoxy позволяет модифицировать поведение объекта как угодно, перехватывать любые обращения к его свойствам и методам, включая вызовы для функций.

Особенно приятна возможность перехватывать обращения к отсутствующим свойствам, разработчики ожидали её уже давно.

Что касается поддержки, то возможности полифиллов здесь ограничены. «Переписать» прокси на старом JavaScript сложновато, учитывая низкоуровневые возможности, которые он даёт.

Поэтому нужна именно браузерная поддержка. [Постепенно](#) она реализуется.

Решения

Привет, мир!

Выведите alert

Код страницы:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>

  <script>
    alert( 'Я - JavaScript!' );
  </script>

</body>

</html>
```

[Открыть решение в песочнице.](#)

[К условию](#)

Внешние скрипты, порядок исполнения

Вывести alert внешним скриптом

Код для HTML-файла:

```
<!DOCTYPE html>
<html>
  <body>
    <script src="alert.js"></script>
  </body>
</html>
```

Для файла `alert.js` из той же директории:

```
alert("I'm JavaScript!");
```

[К условию](#)

Какой скрипт выполнится первым?

Ответы:

1. Первым выполнится `big.js`, это нормальная последовательность выполнения подряд идущих скриптов.
2. Первым выполнится `small.js`, так как скрипты из-за `async` ведут себя совершенно независимо друг от друга, страница тоже от них не зависит.
3. Первым выполнится `big.js`, так как скрипты, подключённые через `defer`, сохраняют порядок выполнения относительно друг друга.

[К условию](#)

Переменные

Работа с переменными

Каждая строчка решения соответствует одному шагу задачи:

```
var admin, name; // две переменных через запятую
name = "Василий";
admin = name;
alert( admin ); // "Василий"
```

[К условию](#)

Правильный выбор имени переменной

Объявление переменных

Каждая строчка решения соответствует одному шагу задачи:

```
var ourPlanetName = "Земля"; // буквально "название нашей планеты"
var userName = "Петя"; // "имя посетителя"
```

Названия переменных можно бы сократить, например, до `planet` и `name`, но тогда станет менее понятно, о чем речь.

Насколько допустимы такие сокращения – зависит от скрипта, его размера и сложности, от того, есть ли другие планеты и пользователи. В общем, лучше не жалеть букв и называть переменные так, чтобы по имени можно было легко понять, что в ней находится, и нельзя было перепутать переменные.

[К условию](#)

Инкремент, порядок срабатывания

Разъяснения

[К условию](#)

Результат присваивания

Ответ: $x = 5$.

Оператор присваивания возвращает значение, которое будет записано в переменную, например:

```
var a = 2;  
alert( a *= 2 ); // 4
```

Отсюда $x = 1 + 4 = 5$.

[К условию](#)

Побитовые операторы

Побитовый оператор и значение

1. Операция a^b ставит бит результата в 1, если на соответствующей битовой позиции в a или b (но не одновременно) стоит 1.

Так как в 0 везде стоят нули, то биты берутся в точности как во втором аргументе.

2. Первое побитовое НЕ \sim превращает 0 в 1, а 1 в 0. А второе НЕ превращает ещё раз, в итоге получается как было.

[К условию](#)

Проверка, целое ли число

Один из вариантов такой функции:

```
function isInteger(num) {  
    return (num ^ 0) === num;  
}  
  
alert( isInteger(1) ); // true  
alert( isInteger(1.5) ); // false  
alert( isInteger(-0.5) ); // false
```

Обратите внимание: num^0 – в скобках! Это потому, что приоритет операции $^$ очень низкий. Если не поставить скобку, то $===$ работает раньше. Получится $num ^ (0 === num)$, а это уже совсем другое дело.

[К условию](#)

Симметричны ли операции $^$, $|$, $\&$?

Операция над числами, в конечном итоге, сводится к битам.

Посмотрим, можно ли поменять местами биты слева и справа.

Например, таблица истинности для $^$:

a	b	результат
0	0	0
0	1	1
1	0	1
1	1	0

Случаи 0^0 и 1^1 заведомо не изменятся при перемене мест, поэтому нас не интересуют. А вот 0^1 и 1^0 эквивалентны и равны 1.

Аналогично можно увидеть, что и другие операторы симметричны.

Ответ: да.

[К условию](#)

Почему результат разный?

Всё дело в том, что побитовые операции преобразуют число в 32-битное целое.

Обычно число в JavaScript имеет 64-битный формат с плавающей точкой. При этом часть битов (52) отведены под цифры, часть (11) отведены под хранение номера позиции, на которой стоит десятичная точка, и один бит – знак числа.

Это означает, что максимальное целое число, которое можно хранить, занимает 52 бита.

Число 12345678912345 в двоичном виде: 101100111010011100111100111001011011011001 (44 цифры).

Побитовый оператор \wedge преобразует его в 32-битное путём отбрасывания десятичной точки и «лишних» старших цифр. При этом, так как число большое и старшие биты здесь ненулевые, то, естественно, оно изменится.

Вот ещё пример:

```
// в двоичном виде 100000000000000000000000000000 (31 цифры)
alert( Math.pow(2, 30) ); // 1073741824
alert( Math.pow(2, 30) ^ 0 ); // 1073741824, всё ок, длины хватает

// в двоичном виде 100000000000000000000000000000 (33 цифры)
alert( Math.pow(2, 32) ); // 4294967296
alert( Math.pow(2, 32) ^ 0 ); // 0, отброшены старшие цифры, остались нули

// пограничный случай
// в двоичном виде 100000000000000000000000000000 (32 цифры)
alert( Math.pow(2, 31) ); // 2147483648
alert( Math.pow(2, 31) ^ 0 ); // -2147483648, ничего не отброшено,
// но первый бит 1 теперь стоит в начале числа и является знаковым
```

[К условию](#)

Взаимодействие с пользователем: alert, prompt, confirm

Простая страница

JS-код:

```
var name = prompt("Ваше имя?", "");
alert( name );
```

Полная страница:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>

  <script>
    var name = prompt("Ваше имя?", "");
    alert( name );
  </script>

</body>

</html>
```

[К условию](#)

Условные операторы: if, '?'

if (строка с нулём)

Да, выведется, т.к. внутри `if` стоит строка `"0"` .

Любая строка, кроме пустой (а здесь она не пустая), в логическом контексте является `true` .

Можно запустить и проверить:

```
if ("0") {  
  alert( 'Привет' );  
}
```

[К условию](#)

Проверка стандарта

```
<!DOCTYPE html>  
<html>  
  
<body>  
  <script>  
    'use strict';  
  
    let value = prompt('What is the "official" name of JavaScript?', '');  
  
    if (value == 'ECMAScript') {  
      alert('Right!');  
    } else {  
      alert("Didn't know? ECMAScript!");  
    }  
  </script>  
  
</body>  
</html>
```

[К условию](#)

Получить знак числа

```
var value = prompt('Введите число', 0);  
  
if (value > 0) {  
  alert( 1 );  
} else if (value < 0) {  
  alert( -1 );  
} else {  
  alert( 0 );  
}
```

[К условию](#)

Проверка логина

```
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.

Впрочем, такое поведение Safari является некорректным, надеемся, что скоро его исправят.

Кроме того, обратите внимание на дополнительные вертикальные отступы внутри `if` . Они не обязательны, но полезны для лучшей читаемости кода.

[К условию](#)

Перепишите 'if' в '?'

```
result = (a + b < 4) ? 'Мало' : 'Много';
```

[К условию](#)

Перепишите 'if..else' в '?'

```
var message = (login == 'Вася') ? 'Привет' :  
(login == 'Директор') ? 'Здравствуйтесь' :  
(login == '') ? 'Нет логина' :  
'';
```

[К условию](#)

Логические операторы

Что выведет alert (ИЛИ)?

Ответ: 2, это первое значение, которое в логическом контексте даст true.

```
alert( null || 2 || undefined );
```

[К условию](#)

Что выведет alert (ИЛИ)?

Ответ: сначала 1, затем 2.

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

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

1. Первый оператор ИЛИ || выполнит первый alert(1), получит undefined и пойдёт дальше, ко второму операнду.
2. Так как второй операнд 2 является истинным, то вычисления завершатся, результатом undefined || 2 будет 2, которое будет выведено внешним alert(....).

Второй оператор || не будет выполнен, выполнение до alert(3) не дойдёт, поэтому 3 выведено не будет.

[К условию](#)

Что выведет alert (И)?

Ответ: null, это первое ложное значение из списка.

```
alert( 1 && null && 2 );
```

[К условию](#)

Что выведет alert (И)?

Ответ: 1, а затем undefined.

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

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

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

[К условию](#)

Что выведет этот код?

Ответ: 3 .

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

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

Последовательность вычислений:

```
null || 2 && 3 || 4
null || 3 || 4
3
```

[К условию](#)

Проверка if внутри диапазона

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

[К условию](#)

Проверка if вне диапазона

Первый вариант:

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

Второй вариант:

```
if (age < 14 || age > 90)
```

[К условию](#)

Вопрос про "if"

Ответ: первое и третье выполнятся.

Детали:

```
// Выполнится
// Результат -1 || 0 = -1, в логическом контексте true
if (-1 || 0) alert( 'первое' );

// Не выполнится
// -1 && 0 = 0, в логическом контексте false
if (-1 && 0) alert( 'второе' );

// Выполнится
// оператор && имеет больший приоритет, чем ||
// так что -1 && 1 выполнится раньше
// вычисления: null || -1 && 1 -> null || 1 -> 1
if (null || -1 && 1) alert( 'третье' );
```

[К условию](#)

Преобразование типов для примитивов

Вопросник по преобразованиям, для примитивов

```
"" + 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.

[К условию](#)

Циклы while, for

Последнее значение цикла

Ответ: 1.

```
var i = 3;
while (i) {
  alert( i-- );
}
```

Каждое выполнение цикла уменьшает i. Проверка while(i) даст сигнал «стоп» при i = 0.

Соответственно, шаги цикла:

```
var i = 3
alert( i-- ); // выведет 3, затем уменьшит i до 2
alert(i--) // выведет 2, затем уменьшит i до 1
alert(i--) // выведет 1, затем уменьшит i до 0
// все, проверка while(i) не даст выполняться циклу дальше
```

[К условию](#)

Какие значения i выведет цикл while?

1. От 1 до 4

```
var i = 0;
while (++i < 5) alert( i );
```

Первое значение: i=1, так как операция ++i сначала увеличит i, а потом уже произойдёт сравнение и выполнение alert.

Далее 2,3,4.. Значения выводятся одно за другим. Для каждого значения сначала происходит увеличение, а потом – сравнение, так как ++ стоит перед переменной.

При i=4 произойдет увеличение i до 5, а потом сравнение while(5 < 5) – это неверно. Поэтому на этом цикл остановится, и значение 5 выведено не будет.

2. От 1 до 5

```
var i = 0;
while (i++ < 5) alert( i );
```

Первое значение: i=1. Остановимся на нём подробнее. Оператор i++ увеличивает i, возвращая старое значение, так что в сравнении i++ < 5 будет участвовать старое i=0.

Но последующий вызов `alert` уже не относится к этому выражению, так что получит новый `i=1`.

Далее 2,3,4... Для каждого значения сначала происходит сравнение, а потом – увеличение, и затем срабатывание `alert`.

Окончание цикла: при `i=4` произойдет сравнение `while(4 < 5)` – верно, после этого сработает `i++`, увеличив `i` до 5, так что значение 5 будет выведено. Оно станет последним.

[К условию](#)

Какие значения `i` выведет цикл `for`?

Ответ: от 0 до 4 в обоих случаях.

```
for (var i = 0; i < 5; ++i) alert( i );
```

```
for (var i = 0; i < 5; i++) alert( i );
```

Такой результат обусловлен алгоритмом работы `for`:

1. Выполнить присвоение `i=0`
2. Проверить `i<5`
3. Если верно – выполнить тело цикла `alert(i)`, затем выполнить `i++`

Увеличение `i++` выполняется отдельно от проверки условия (2), значение `i` при этом не используется, поэтому нет никакой разницы между `i++` и `++i`.

[К условию](#)

Выведите чётные числа

```
for (var i = 2; i <= 10; i++) {  
  if (i % 2 == 0) {  
    alert( i );  
  }  
}
```

Чётность проверяется по остатку при делении на 2, используя оператор «деление с остатком» `%`: `i % 2`.

[К условию](#)

Замените `for` на `while`

```
var i = 0;  
while (i < 3) {  
  alert( "номер " + i + "!" );  
  i++;  
}
```

[К условию](#)

Повторять цикл, пока ввод неверен

```
var num;  
do {  
  num = 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 ещё 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.

[К условию](#)

Вывести простые числа

Решение

[К условию](#)

Конструкция switch

Напишите "if", аналогичный "switch"

Если совсем точно следовать условию, то сравнение должно быть строгим '===' .

В реальном случае, скорее всего, подойдёт обычное сравнение '==' .

```
if(browser == 'IE') {
  alert('О, да у вас IE!');
} else if (browser == 'Chrome'
|| browser == 'Firefox'
|| browser == 'Safari'
|| browser == 'Opera') {
  alert('Да, и эти браузеры мы поддерживаем');
} else {
  alert('Мы надеемся, что и в вашем браузере все ок!');
}
```

Обратите внимание: конструкция browser == 'Chrome' || browser == 'Firefox' ... разбита на несколько строк для лучшей читаемости.

Но всё равно запись через switch нагляднее.

[К условию](#)

Переписать if'ы в switch

Первые две проверки – обычный case , третья разделена на два case :

```
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.

[К условию](#)

Функции

Обязателен ли "else"?

Оба варианта функции работают одинаково, отличий нет.

[К условию](#)

Перепишите функцию, используя оператор '?' или '||'

Используя оператор '?' :

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

Используя оператор `||` (самый короткий вариант):

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

[К условию](#)

Функция min

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.

[К условию](#)

Функция pow(x,n)

```
/**
 * Возводит x в степень n (комментарий JSDoc)
 *
 * @param {number} x число, которое возводится в степень
 * @param {number} n степень, должна быть целым числом больше 1
 *
 * @return {number} x в степени n
 */
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) );
}
```

[К условию](#)

Рекурсия, стек

Вычислить сумму чисел до данного

Решение с использованием цикла:

```
function sumTo(n) {
  var sum = 0;
  for (var i = 1; i <= n; i++) {
    sum += i;
  }
  return sum;
}
```

```
alert( sumTo(100) );
```

Решение через рекурсию:

```
function sumTo(n) {  
  if (n == 1) return 1;  
  return n + sumTo(n - 1);  
}  
  
alert( sumTo(100) );
```

Решение по формуле: $sumTo(n) = n*(n+1)/2$:

```
function sumTo(n) {  
  return n * (n + 1) / 2;  
}  
  
alert( sumTo(100) );
```

P.S. Надо ли говорить, что решение по формуле работает быстрее всех? 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.

[К условию](#)

Вычислить факториал

По свойствам факториала, как описано в условии, $n!$ можно записать как $n * (n-1)!$.

То есть, результат функции для n можно получить как n , умноженное на результат функции для $n-1$, и так далее до $1!$:

```
function factorial(n) {  
  return (n != 1) ? n * factorial(n - 1) : 1;  
}  
  
alert( factorial(5) ); // 120
```

Базисом рекурсии является значение 1 . А можно было бы сделать базисом и 0 . Тогда код станет чуть короче:

```
function factorial(n) {  
  return n ? n * factorial(n - 1) : 1;  
}  
  
alert( factorial(5) ); // 120
```

В этом случае вызов `factorial(1)` сведётся к $1*factorial(0)$, будет дополнительный шаг рекурсии.

[К условию](#)

Числа Фибоначчи

Вычисление рекурсией (медленное)

Алгоритм вычисления в цикле

Код для вычисления в цикле

[К условию](#)

Именованные функциональные выражения

Проверка на NFE

Первый код выведет `function ...`, второй – ошибку во всех браузерах, кроме IE8-.

```
// обычное объявление функции (Function Declaration)
function g() { return 1; };

alert(g); // функция
```

Во втором коде скобки есть, значит функция внутри является не `Function Declaration`, а частью выражения, то есть `Named Function Expression`. Его имя видно только внутри, снаружи переменная `g` не определена.

```
// Named Function Expression!
(function g() { return 1; });

alert(g); // Ошибка!
```

Все браузеры, кроме IE8-, поддерживают это ограничение видимости и выведут ошибку, `"undefined variable"`.

[К условию](#)

Советы по стилю кода

Ошибки в стиле

Ответ

[К условию](#)

Автоматические тесты при помощи `chai` и `mocha`

Сделать `pow` по спецификации

```
function pow(x, n) {
  if (n < 0) return NaN;
  if (Math.round(n) !== n) return NaN;

  var result = 1;
  for (var i = 0; i < n; i++) {
    result *= x;
  }
  return result;
}
```

[Открыть решение с тестами в песочнице.](#) ↗

[К условию](#)

Добавьте тест к задаче

Новый тест может быть, к примеру, таким:

```
it("любое число в степени 0 равно 1", function() {
  assert.equal(pow(123, 0), 1);
});
```

Конечно, желательно проверить на нескольких числах.

Поэтому лучше будет создать блок `describe`, аналогичный тому, что мы делали для произвольных чисел:

```
describe("любое число, кроме нуля, в степени 0 равно 1", function() {
  function makeTest(x) {
    it("при возведении " + x + " в степень 0 результат: 1", function() {
      assert.equal(pow(x, 0), 1);
    });
  }

  for (var x = -5; x <= 5; x += 2) {
    makeTest(x);
  }
});
```

И не забудем добавить отдельный тест для нуля:

```
...
it("ноль в нулевой степени даёт NaN", function() {
  assert( isNaN(pow(0, 0)), "0 в степени 0 не NaN");
});
...
```

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Что не так в тесте?

Этот тест демонстрирует один из соблазнов, которые ожидают начинающего автора тестов.

Вместо того, чтобы написать три различных теста, он изложил их в виде одного потока вычислений, с несколькими `assert`.

Иногда так написать легче и проще, однако при ошибке в тесте гораздо менее очевидно, что же пошло не так.

Если в сложном тесте произошла ошибка где-то посередине потока вычислений, то придётся выяснять, какие конкретно были входные и выходные данные на этот момент, то есть по сути – отлаживать код самого теста.

Гораздо лучше будет разбить тест на несколько блоков `it`, с чётко прописанными входными и выходными данными.

```
describe("Возводит x в степень n", function() {
  it("5 в степени 1 равно 5", function() {
    assert.equal(pow(5, 1), 5);
  });

  it("5 в степени 2 равно 25", function() {
    assert.equal(pow(5, 2), 25);
  });

  it("5 в степени 3 равно 125", function() {
    assert.equal(pow(5, 3), 125);
  });
});
```

Можно использовать цикл для генерации блоков `it`, в этом случае важно, чтобы сам код такого цикла был достаточно простым. Иногда проще записать несколько блоков `it` вручную, как сделано выше, чем «городить огород» из синтаксических конструкций.

[К условию](#)

Числа

Интерфейс суммы

```
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"`.

[К условию](#)

Почему `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.

[К условию](#)

Сложение цен

Есть два основных подхода.

1. 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.
2. 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) );
```

[К условию](#)

Бесконечный цикл по ошибке

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 );
}
```

Ни одно из них в точности не равно `10`.

[К условию](#)

Как получить дробную часть числа?

Функция

[К условию](#)

Формула Бине

```
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 $\sqrt{5}$ – бесконечная дробь.

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

[К условию](#)

Случайное из интервала (0, max)

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

```
var max = 10;
alert( Math.random() * max );
```

[К условию](#)

Случайное из интервала (min, max)

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

```
var min = 5,
    max = 10;

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

[К условию](#)

Случайное целое от min до max

Очевидное неверное решение (round)

Верное решение с round

Решение с floor

[К условию](#)

Строки

Сделать первый символ заглавным

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;

  return str[0].toUpperCase() + str.slice(1);
}

alert( ucFirst("вася") );
```

P.S. Возможны и более короткие решения, использующие методы для работы со строками, которые мы пройдем далее.

[Открыть решение с тестами в песочнице.](#) ↗

[К условию](#)

Проверьте спам

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

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

```
function checkSpam(str) {
  var lowerStr = str.toLowerCase();

  return !(~lowerStr.indexOf('viagra') || ~lowerStr.indexOf('xxx'));
}

alert( checkSpam('buy ViAgRA now') );
alert( checkSpam('free xxxxx') );
alert( checkSpam("innocent rabbit") );
```

[Открыть решение с тестами в песочнице.](#) ↗

[К условию](#)

Усечение строки

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: `…`), который можно использовать вместо трёх точек. Если его использовать, то можно отрезать только один символ.

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Выделить число

Возьмём часть строки после первого символа и приведём к числу: `+str.slice(1)`.

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Объекты как ассоциативные массивы

Первый объект

```
var user = {};
user.name = "Вася";
user.surname = "Петров";
user.name = "Сергей";
delete user.name;
```

[К условию](#)

Объекты: перебор свойств

Определите, пуст ли объект

```
function isEmpty(obj) {
  for (var key in obj) {
    return false;
  }
  return true;
}

var schedule = {};

alert( isEmpty(schedule) ); // true

schedule["8:30"] = "подъём";

alert( isEmpty(schedule) ); // false
```

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Сумма свойств

```
"use strict";

var salaries = {
  "Вася": 100,
  "Петя": 300,
  "Даша": 250
};

var sum = 0;
for (var name in salaries) {
  sum += salaries[name];
}

alert( sum );
```

[К условию](#)

Свойство с наибольшим значением

```
"use strict";

var salaries = {
  "Вася": 100,
  "Петя": 300,
  "Даша": 250
};

var max = 0;
var maxName = "";
for (var name in salaries) {
  if (max < salaries[name]) {
    max = salaries[name];
    maxName = name;
  }
}

alert( maxName || "нет сотрудников" );
```

[К условию](#)

Умножьте численные свойства на 2

```
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 );
```

[Открыть решение с тестами в песочнице.](#) ↗

[К условию](#)

Массивы с числовыми индексами

Получить последний элемент массива

Последний элемент имеет индекс на 1 меньше, 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
```

[К условию](#)

Добавить новый элемент в массив

Текущий последний элемент имеет индекс `goods.length-1`. Значит, индексом нового элемента будет `goods.length` :

```
goods[goods.length] = 'Компьютер'
```

[К условию](#)

Создание массива

```
var styles = ["Джаз", "Блюз"];
styles.push("Рок-н-Ролл");
styles[styles.length - 2] = "Классика";
alert( styles.shift() );
styles.unshift("Рэн", "Регги");
```

[К условию](#)

Получить случайное значение из массива

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] );
```

[К условию](#)

Создайте калькулятор для введённых значений

В решение ниже обратите внимание: мы не приводим `value` к числу сразу после `prompt`, так как если сделать `value = +value`, то после этого отличить пустую строку от нуля уже никак нельзя. А нам здесь нужно при пустой строке прекращать ввод, а при нуле – продолжать.

```
var numbers = [];
while (true) {
  var value = prompt("Введите число", 0);
  if (value === "" || value === null || isNaN(value)) break;
  numbers.push(+value);
}
var sum = 0;
for (var i = 0; i < numbers.length; i++) {
  sum += numbers[i];
}
alert( sum );
```

[К условию](#)

Чему равен элемент массива?

```
var arr = [1, 2, 3];
var arr2 = arr; // (*)
arr2[0] = 5;
alert( arr[0] );
alert( arr2[0] );
```

Код выведет 5 в обоих случаях, так как массив является объектом. В строке (*) в переменную arr2 копируется ссылка на него, а сам объект в памяти по-прежнему один, в нём отражаются изменения, внесенные через arr2 или arr .

В частности, сравнение arr2 == arr даст true .

Если нужно именно скопировать массив, то это можно сделать, например, так:

```
var arr2 = [];  
for (var i = 0; i < arr.length; i++) arr2[i] = arr[i];
```

[К условию](#)

Поиск в массиве

Возможное решение:

```
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](#) , 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 .

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Фильтр диапазона

Алгоритм решения

Решение

[К условию](#)

Решето Эратосфена

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 );
```

[К условию](#)

Подмассив наибольшей суммы

Подсказка (медленное решение)

Медленное решение

Подсказка (быстрое решение)

Быстрое решение

[К условию](#)

Массивы: методы

Добавить класс в строку

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(' ') : [];

  for (var i = 0; i < classes.length; i++) {
    if (classes[i] == cls) return; // класс уже есть
  }

  classes.push(cls); // добавить

  obj.className = classes.join(' '); // и обновить свойство
}

var obj = {
  className: 'open menu'
};

addClass(obj, 'new');
addClass(obj, 'open');
addClass(obj, 'me');
alert(obj.className) // open menu new me
```

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 тудеу"`.

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 = ""`?

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Перевести текст вида `border-left-width` в `borderLeftWidth`

Идея

Решение

[К условию](#)

Функция `removeClass`

Решение заключается в том, чтобы разбить `className` в массив классов, а затем пройтись по нему циклом. Если класс есть – удаляем его `splice`, заново объединяем массив в строку и присваиваем объекту.

```
function removeClass(obj, cls) {
  var classes = obj.className.split(' ');

  for (var i = 0; i < classes.length; i++) {
    if (classes[i] == cls) {
      classes.splice(i, 1); // удалить класс
      i--; // (*)
    }
  }
  obj.className = classes.join(' ');
}

var obj = {
  className: 'open menu menu'
}

removeClass(obj, 'blabla');
removeClass(obj, 'menu');
alert(obj.className) // open
```

В примере выше есть тонкий момент. Элементы массива проверяются один за другим. При вызове `splice` удаляется текущий, *i*-й элемент, и те элементы, которые идут дальше, сдвигаются на его место.

Таким образом, на месте *i* оказывается новый, непроверенный элемент.

Чтобы это учесть, строчка (*) уменьшает *i*, чтобы следующая итерация цикла заново проверила элемент с номером *i*. Без нее функция будет работать с ошибками.

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Фильтрация массива "на месте"

```
function filterRangeInPlace(arr, a, b) {
  for (var i = 0; i < arr.length; i++) {
    var val = arr[i];
    if (val < a || val > b) {
      arr.splice(i--, 1);
    }
  }
}

var arr = [5, 3, 8, 1];

filterRangeInPlace(arr, 1, 4);
alert( arr ); // [3, 1]
```

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Сортировать в обратном порядке

```
var arr = [5, 2, 1, -10, 8];

function compareReversed(a, b) {
  return b - a;
}

arr.sort(compareReversed);

alert( arr );
```

[К условию](#)

Скопировать и отсортировать массив

Для копирования массива используем `slice()`, и тут же – сортировку:

```
var arr = ["HTML", "JavaScript", "CSS"];

var arrSorted = arr.slice().sort();

alert( arrSorted );
alert( arr );
```

[К условию](#)

Случайный порядок в массиве

Подсказка

Решение

[К условию](#)

Сортировка объектов

Для сортировки объявим и передадим в `sort` анонимную функцию, которая сравнивает объекты по полю `age`:

```
// Наша функция сравнения
function compareAge(personA, personB) {
  return personA.age - personB.age;
}

// проверка
var vasya = { name: "Вася", age: 23 };
var masha = { name: "Маша", age: 18 };
var vovochka = { name: "Вовочка", age: 6 };

var people = [ vasya , masha , vovochka ];

people.sort(compareAge);

// вывести
for(var i = 0; i < people.length; i++) {
  alert(people[i].name); // Вовочка Маша Вася
}
```

[К условию](#)

Вывести односвязный список

Вывод списка в цикле

Вывод списка с рекурсией

Обратный вывод с рекурсией

Обратный вывод без рекурсии

[К условию](#)

Отфильтровать анаграммы

Решение

[К условию](#)

Оставить уникальные элементы массива

Решение перебором (медленное)

Решение с объектом (быстрое)

[К условию](#)

Массив: перебирающие методы

Перепишите цикл через map

```
var arr = ["Есть", "жизнь", "на", "Марсе"];  
  
var arrLength = arr.map(function(item) {  
  return item.length;  
});  
  
alert( arrLength ); // 4,5,2,5
```

[К условию](#)

Массив частичных сумм

Метод `arr.reduce` подходит здесь идеально. Достаточно пройти по массиву слева-направо, 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);  
    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;  
  
  var totalSum = arr.reduce(function(sum, item) {  
    result.push(sum);  
    return sum + item;  
  });  
  result.push(totalSum);  
  
  return result;  
}  
  
alert(getSums([1,2,3,4,5])); // 1,3,6,10,15  
alert(getSums([-2,-1,0,1])); // -2,-3,-3,-2
```

[Открыть решение с тестами в песочнице.](#) ↗

[К условию](#)

Псевдомассив аргументов "arguments"

Проверка на аргумент-undefined

Узнать количество реально переданных аргументов можно по значению `arguments.length` :

```
function f(x) {
  alert( arguments.length ? 1 : 0 );
}

f(undefined);
f();
```

[К условию](#)

Сумма аргументов

```
function sum() {
  var result = 0;

  for (var i = 0; i < arguments.length; i++) {
    result += arguments[i];
  }

  return result;
}

alert( sum() ); // 0
alert( sum(1) ); // 1
alert( sum(1, 2) ); // 3
alert( sum(1, 2, 3) ); // 6
alert( sum(1, 2, 3, 4) ); // 10
```

[К условию](#)

Дата и Время

Создайте дату

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 );
```

[К условию](#)

Имя дня недели

Метод `getDay()` позволяет получить номер дня недели, начиная с воскресенья.

Запишем имена дней недели в массив, чтобы можно было их достать по номеру:

```
function getWeekDay(date) {
  var days = ['вс', 'пн', 'вт', 'ср', 'чт', 'пт', 'сб'];

  return days[date.getDay()];
}

var date = new Date(2014, 0, 3); // 3 января 2014
alert( getWeekDay(date) ); // 'пт'
```

В современных браузерах можно использовать и `toLocaleString` :

```
var date = new Date(2014, 0, 3); // 3 января 2014
alert( date.toLocaleString('ru', {weekday: 'short'}) ); // 'пт'
```


[Открыть решение с тестами в песочнице.](#)

[К условию](#)

День недели в европейской нумерации

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;
}

alert( getLocalDay(new Date(2012, 0, 3)) ); // 2
```

Если удобнее, чтобы день недели начинался с нуля, то можно возвращать в функции `day - 1`, тогда дни будут от 0 (пн) до 6(вс).

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

День указанное количество дней назад

Из даты `date` нужно вычесть указанное количество дней. Это просто:

```
function getDateAgo(date, days) {
    date.setDate(date.getDate() - days);
    return date.getDate();
}
```

Ситуацию усложняет 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();
}

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)
```

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Последний день месяца?

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();
}

alert( getLastDayOfMonth(2012, 0) ); // 31
alert( getLastDayOfMonth(2012, 1) ); // 29
alert( getLastDayOfMonth(2013, 1) ); // 28
```

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Сколько секунд уже прошло сегодня?

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();
};
```

[К условию](#)

Сколько секунд - до завтра?

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
}
```

[К условию](#)

Вывести дату в формате дд.мм.гг

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;
```

2. `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;
```

3. `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;
}

var d = new Date(2014, 0, 30); // 30 Янв 2014
alert( formatDate(d) ); // '30.01.14'
```

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Относительное форматирование даты

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(':');
}

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"
```

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Глобальный объект

Window и переменная

Ответ: 1.

```
if ("a" in window) {
  var a = 1;
}
alert( a );
```

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 работает
}
alert( a );
```

В результате `a` становится 1.

[К условию](#)

Window и переменная 2

Ответ: ошибка.

Переменной `a` нет, так что условие `"a" in window` не выполнится. В результате на последней строчке – обращение к неопределенной переменной.

```
if ("a" in window) {
  a = 1;
}
alert( a ); // <-- error!
```

[К условию](#)

Window и переменная 3

Ответ: 1 .

Переменная `a` создается до начала выполнения кода, так что условие `"a" in window` выполнится и сработает `a = 1`.

```
if ("a" in window) {
  a = 1;
}
var a;

alert( a ); // 1
```

[К условию](#)

Замыкания, функции изнутри

Что выведет say в начале кода?

Ошибки не будет, выведет `"Вася, undefined"` .

```
say('Вася'); // Что выведет? Не будет ли ошибки?

var phrase = 'Привет';

function say(name) {
  alert( name + ", " + phrase );
}
```

Переменная как таковая существует, вот только на момент запуска функции она равна `undefined` .

[К условию](#)

В какую переменную будет присвоено значение?

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.

[К условию](#)

var window

Результатом будет `undefined` , затем 5 .

```
function test() {
  alert( window );

  var window = 5;

  alert( window );
}

test();
```

Такой результат получился потому, что `window` – это глобальная переменная, но ничто не мешает объявить такую же локальную.

Директива `var window` обрабатывается 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` .

[К условию](#)

Вызов "на месте"

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.

[К условию](#)

Перекрытие переменной

Нет, нельзя.

Локальная переменная полностью перекрывает внешнюю.

[К условию](#)

Глобальный счётчик

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.

```
var currentCount = 1;  
  
function makeCounter() {  
  return function() {  
    return currentCount++;  
  };  
}  
  
var counter = makeCounter();  
var counter2 = makeCounter();  
  
alert( counter() ); // ?  
alert( counter() ); // ?
```

```
alert( counter2() ); // ?
alert( counter2() ); // ?
```

[К условию](#)

Локальные переменные для объекта

Сумма через замыкание

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
  };
}

alert( sum(1)(2) );
alert( sum(5)(-1) );
```

[К условию](#)

Функция - строковый буфер

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.

[Открыть решение с тестами в песочнице.](#) ↗

[К условию](#)

Строковый буфер с очисткой

```
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() ); // ""
```

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Сортировка

```
var users = [{
  name: "Вася",
  surname: 'Иванов',
  age: 20
}, {
  name: "Петя",
  surname: 'Чапаяев',
  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) {
  alert( user.name );
});
```

```
users.sort(byField('age'));
users.forEach(function(user) {
  alert( user.name );
});
```

[К условию](#)

Фильтрация через функцию

Функция фильтрации

Фильтр inBetween

Фильтр inArray

[К условию](#)

Армия функций

Что происходит в этом коде

Почему ошибка

Исправление (3 варианта)

[К условию](#)

Устаревшая конструкция "with"

With + функция

Вторая (2), т.к. при обращении к любой переменной внутри with – она ищется прежде всего в объекте.

Соответственно, будет выведено 2 :

```
function f() {
  alert(1)
}

var obj = {
  f: function() {
    alert(2)
  }
};

with(obj) {
  f();
}
```

[К условию](#)

With + переменные

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 );
}
```

[К условию](#)

Методы объектов, this

Вызов в контексте массива

Вызов `arr[2]()` – это обращение к методу объекта `obj[method]()`, в роли `obj` выступает `arr`, а в роли метода: `2`.

Поэтому, как это бывает при вызове функции как метода, функция `arr[2]` получит `this = arr` и выведет массив:

```
var arr = ["a", "b"];
arr.push(function() {
  alert( this );
})

arr[2](); // "a","b",function
```

[К условию](#)

Проверка синтаксиса

Ошибка!

Попробуйте:

```
var obj = {
  go: function() {
    alert(this)
  }
}

(obj.go)() // error!
```

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.

[К условию](#)

Почему this присваивается именно так?

1. Обычный вызов функции в контексте объекта.
2. То же самое, скобки ни на что не влияют.
3. Здесь не просто вызов obj.method() , а более сложный вызов вида (выражение).method() . Такой вызов работает, как если бы он был разбит на две строки:

```
f = obj.go; // сначала вычислить выражение
f();       // потом вызвать то, что получилось
```

При этом f() выполняется как обычная функция, без передачи this .

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

В спецификации это объясняется при помощи специального внутреннего типа [Reference Type](#) ↗.

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

1. Сначала получить свойство obj.go .
2. 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.

[К условию](#)

Значение this в объявлении объекта

Ответ: **undefined** .

```
var user = {
  firstName: "Василий",
  export: this // (*)
};
alert( user.export.firstName );
```

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.

[К условию](#)

Возврат this

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" .

[К условию](#)

Возврат объекта с this

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` .

```
var name = "";  
  
var user = {  
  name: "Василий",  
  
  export: function() {  
    return {  
      value: this  
    };  
  }  
};  
  
alert( user.export().value == user ); // true
```

[К условию](#)

Создайте калькулятор

```
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() );
```

[Открыть решение с тестами в песочнице.](#) ↗

[К условию](#)

Цепочка вызовов

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

[К условию](#)

Преобразование объектов: toString и valueOf

`['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
```

[К условию](#)

Преобразование

Первый `alert(foo)`

Второй `alert(foo + 1)`

Третий `alert(foo + „3“)`

[К условию](#)

Почему `[] == []` неверно, а `[] == ![]` верно?

Ответ по первому равенству

Ответ по второму равенству

[К условию](#)

Вопросник по преобразованиям, для объектов

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

- `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(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"`.
- 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.
- Массив преобразуется в строку `"1"`. Оператор `+` при сложении со строкой приводит второй аргумент к строке – значит будет `"1" + "1" = "11"`.
- Массивы приводятся к строке и складываются.
- Массив преобразуется в пустую строку `""` + `null + 1`, оператор `+` видит, что слева строка и преобразует `null` к строке, получается `"null" + 1`, и в итоге `"null1"`.
- `[[0]]` – это вложенный массив `[0]` внутри внешнего `[]`. Затем мы берём от него нулевой элемент, и потом еще раз.

Если это непонятно, то посмотрите на такой пример:

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

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

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

[К условию](#)

Сумма произвольного количества скобок

Подсказка

Решение

[К условию](#)

Создание объектов через "new"

Две функции один объект

Да, возможны.

Они должны возвращать одинаковый объект. При этом если функция возвращает объект, то `this` не используется.

Например, они могут вернуть один и тот же объект `obj`, определённый снаружи:

```
var obj = {};  
  
function A() { return obj; }  
function B() { return obj; }  
  
var a = new A;  
var b = new B;  
  
alert( a == b ); // true
```

[К условию](#)

Создать Calculator при помощи конструктора

```
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() );
```

[Открыть решение с тестами в песочнице.](#) ↗

[К условию](#)

Создать Accumulator при помощи конструктора

```
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 );
```

[Открыть решение с тестами в песочнице.](#) ↗

Создайте калькулятор

```
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;
  };
};

var calc = new Calculator;

calc.addMethod("*", function(a, b) {
  return a * b;
});
calc.addMethod("/", function(a, b) {
  return a / b;
});
calc.addMethod("**", function(a, b) {
  return Math.pow(a, b);
});

var result = calc.calculate("2 ** 3");
alert( result ); // 8
```

- Обратите внимание на хранение методов. Они просто добавляются к внутреннему объекту.
- Все проверки и преобразование к числу производятся в методе `calculate`. В дальнейшем он может быть расширен для поддержки более сложных выражений.

[Открыть решение с тестами в песочнице.](#) ↗

Дескрипторы, геттеры и сеттеры свойств

Добавить get/set-свойства

```
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
```

[К условию](#)

Статические и фабричные методы

Счетчик объектов

Решение (как вариант):

```
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 );
};

new Article();
new Article();

Article.showStats(); // Всего: 2, Последняя: (дата)

new Article();

Article.showStats(); // Всего: 3, Последняя: (дата)
```

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Явное указание this: "call", "apply"

Перепишите суммирование аргументов

Первый вариант

Второй вариант

[К условию](#)

Примените функцию к аргументам

```
function sum() {
  return [].reduce.call(arguments, function(a, b) {
    return a + b;
  });
}

function mul() {
  return [].reduce.call(arguments, function(a, b) {
    return a * b;
  });
}

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
```

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Привязка контекста и карринг: "bind"

Кросс-браузерная эмуляция bind

Страшновато выглядит, да? Работает так (по строкам):

1. Вызов `bind` сохраняет дополнительные аргументы `args` (они идут со 2-го номера) в массив `bindArgs`.
2. ... и возвращает обертку `wrapper`.
3. Эта обёртка делает из `arguments` массив `args` и затем, используя метод `concat`, прибавляет их к аргументам `bindArgs` (карринг).
4. Затем передаёт вызов `func` с контекстом и общим массивом аргументов.

[К условию](#)

Запись в объект после bind

Ответ: Hello .

```
function f() {
  alert( this );
}

var user = {
  g: f.bind("Hello")
}

user.g();
```

Так как вызов идёт в контексте объекта `user.g()`, то внутри функции `g` контекст `this = user`.

Однако, функции `g` совершенно без разницы, какой `this` она получила.

Её единственное предназначение – это передать вызов в `f` вместе с аргументами и ранее указанным контекстом "Hello", что она и делает.

Эта задача демонстрирует, что изменить однажды привязанный контекст уже нельзя.

[К условию](#)

Повторный bind

Ответ: "Вася" .

```
function f() {
  alert(this.name);
}

f = f.bind( {name: "Вася"} ).bind( {name: "Петя"} );

f(); // Вася
```

Первый вызов `f.bind(..Вася..)` возвращает «обёртку», которая устанавливает контекст для `f` и передаёт вызов `f`.

Следующий вызов `bind` будет устанавливать контекст уже для этой обёртки. Это ни на что не повлияет.

Чтобы это проще понять, используем наш собственный вариант `bind` вместо встроенного:

```
function bind(func, context) {
  return function() {
    return func.apply(context, arguments);
  };
}
```

Код станет таким:

```
function f() {
  alert(this.name);
}

f = bind(f, {name: "Вася"}); // (1)
f = bind(f, {name: "Петя"}); // (2)

f(); // Вася
```

Здесь видно, что первый вызов `bind`, в строке (1), возвращает обёртку вокруг `f`, которая выглядит так (выделена):

```
function bind(func, context) {
  return function() {
    // здесь this не используется
    return func.apply(context, arguments);
  };
}
```

В этой обёртке нигде не используется `this`, контекст `context` берётся из замыкания. Посмотрите на код, там нигде нет `this`.

Поэтому следующий `bind` в строке (2), который выполняется уже над обёрткой и фиксирует в ней `this`, ни на что не влияет. Какая разница, что будет в качестве `this` в функции, которая этот `this` не использует? Контекст `context`, как видно в коде выше, она получает через замыкание из аргументов первого `bind`.

[К условию](#)

Свойство функции после bind

Ответ: `undefined`.

Результатом работы `bind` является функция-обёртка над `sayHi`. Эта функция – самостоятельный объект, у неё уже нет свойства `test`.

[К условию](#)

Использование функции вопросов

Решение с `bind`

Решение через замыкание

[К условию](#)

Использование функции вопросов с каррингом

Решение с `bind`

Решение с локальной переменной

[К условию](#)

Функции-обёртки, декораторы

Логирующий декоратор (1 аргумент)

Возвратим декоратор `wrapper` который будет записывать аргумент в `log` и передавать вызов в `f`:

```
function work(a) {
  /*...*/ // work - произвольная функция, один аргумент
}

function makeLogging(f, log) {
  function wrapper(a) {
    log.push(a);
    return f.call(this, a);
  }

  return wrapper;
}

var log = [];
work = makeLogging(work, log);

work(1); // 1
work(5); // 5

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

Обратите внимание, вызов функции осуществляется как `f.call(this, a)`, а не просто `f(a)`.

Передача контекста необходима, чтобы декоратор корректно работал с методами объекта. Например:

```
user.method = makeLogging(user.method, log);
```

Теперь при вызове `user.method(...)` в декоратор будет передаваться контекст `this`, который надо передать исходной функции через `call/apply`.

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Логирующий декоратор (много аргументов)

Решение аналогично задаче [Логирующий декоратор \(1 аргумент\)](#), разница в том, что в лог вместо одного аргумента идет весь объект `arguments`.

Для передачи вызова с произвольным количеством аргументов используем `f.apply(this, arguments)`.

```
function work(a, b) {
  alert( a + b ); // work - произвольная функция
}

function makeLogging(f, log) {

  function wrapper() {
    log.push(...slice.call(arguments));
    return f.apply(this, arguments);
  }

  return wrapper;
}

var log = [];
work = makeLogging(work, log);

work(1, 2); // 3
work(4, 5); // 9

for (var i = 0; i < log.length; i++) {
  var args = log[i]; // массив из аргументов i-го вызова
  alert( 'Лог:' + args.join() ); // "Лог:1,2", "Лог:4,5"
}
```

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Кеширующий декоратор

Запоминать результаты вызова функции будем в замыкании, в объекте `cache: { ключ:значение }`.

```
function f(x) {
  return Math.random()*x;
}

function makeCaching(f) {
  var cache = {};

  return function(x) {
    if (!(x in cache)) {
      cache[x] = f.call(this, x);
    }
    return cache[x];
  };
}

f = makeCaching(f);

var a = f(1);
var b = f(1);
alert( a == b ); // true (значение закешировано)

b = f(2);
alert( a == b ); // false, другой аргумент => другое значение
```

Обратите внимание: проверка на наличие уже подсчитанного значения выглядит так: `if (x in cache)`. Менее универсально можно проверить так: `if (cache[x])`, это если мы точно знаем, что `cache[x]` никогда не будет `false`, `0` и т.п.

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Типы данных: `[[Class]]`, `instanceof` и утки

Полиморфная функция formatDate

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

Примеры его работы:

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

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

Для отличия `Array` используем вызов `Array.isArray`. Если он неверен, значит у нас дата.

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Формат JSON, метод toJSON

Превратите объект в JSON

```
var leader = {
  name: "Василий Иванович",
  age: 35
};

var leaderStr = JSON.stringify(leader);
leader = JSON.parse(leaderStr);
```

[К условию](#)

Превратите объекты со ссылками в JSON

Ответ на первый вопрос

Варианты решения

[К условию](#)

setTimeout и setInterval

Вывод чисел каждые 100 мс

```
function printNumbersInterval() {
  var i = 1;
  var timerId = setInterval(function() {
    console.log(i);
    if (i == 20) clearInterval(timerId);
    i++;
  }, 100);
}

// вызов
printNumbersInterval();
```

[К условию](#)

Вывод чисел каждые 100 мс, через setTimeout

```
function printNumbersTimeout20_100() {
  var i = 1;
  var timerId = setTimeout(function go() {
    console.log(i);
    if (i < 20) setTimeout(go, 100);
    i++;
  }, 100);
}
```

```
    }, 100);  
  }  
  
  // вызов  
  printNumbersTimeout20_100();
```

[К условию](#)

Для подсветки setInterval или setTimeout?

Нужно выбрать вариант 2, который гарантирует браузеру свободное время между выполнениями `highlight`.

Первый вариант может загрузить процессор на 100%, если `highlight` занимает время, близкое к 10 мс или, тем более, большее чем 10 мс, т.к. таймер не учитывает время выполнения функции.

Что интересно, в обоих случаях браузер не будет выводить предупреждение о том, что скрипт занимает много времени. Но от 100% загрузки процессора возможны притормаживания других операций. В общем, это совсем не то, что мы хотим, поэтому вариант 2.

[К условию](#)

Что выведет setTimeout?

Ответы:

- `alert` выведет `100000000`.
- 3, срабатывание будет после окончания работы `hardWork`.

Так будет потому, что вызов планируется на 100 мс от времени вызова `setTimeout`, но функция выполняется больше, чем 100 мс, поэтому к моменту ее окончания время уже подошло и отложенный вызов выполняется тут же.

[К условию](#)

Что выведет после setInterval?

Вызов `alert(i)` в `setTimeout` выведет `100000001`.

Можете проверить это запуском:

```
var timer = setInterval(function() {  
  i++;  
}, 10);  
  
setTimeout(function() {  
  clearInterval(timer);  
  alert( i ); // (*)  
}, 50);  
  
var i;  
  
function f() {  
  // точное время выполнения не играет роли  
  // здесь оно заведомо больше 100 мс  
  for (i = 0; i < 1e8; i++) f[i % 2] = i;  
}  
  
f();
```

Правильный вариант срабатывания: 3 (сразу же по окончании `f` один раз).

Планирование `setInterval` будет вызывать функцию каждые 10 мс после текущего времени. Но так как интерпретатор занят долгой функцией, то до конца ее работы никакого вызова не происходит.

За время выполнения `f` может пройти время, на которое запланированы несколько вызовов `setInterval`, но в этом случае остается только один, т.е. накопления вызовов не происходит. Такова логика работы `setInterval`.

После окончания текущего скрипта интерпретатор обращается к очереди запланированных вызовов, видит в ней `setInterval` и выполняет. А затем тут же выполняется `setTimeout`, очередь которого тут же подошла.

Итого, как раз и видим, что `setInterval` выполнился ровно 1 раз по окончании работы функции. Такое поведение кросс-браузерно.

[К условию](#)

Кто быстрее?

Задача – с небольшим «нюансом».

Есть браузеры, в которых на время работы JavaScript таймер «застывает», например таков IE. В них количество шагов будет почти одинаковым, ±1.

В других браузерах (Chrome) первый бегун будет быстрее.

Создадим реальные объекты Runner и запустим их для проверки:

```
function Runner() {
  this.steps = 0;

  this.step = function() {
    this.doSomethingHeavy();
    this.steps++;
  };

  function fib(n) {
    return n <= 1 ? n : fib(n - 1) + fib(n - 2);
  }

  this.doSomethingHeavy = function() {
    for (var i = 0; i < 25; i++) {
      this[i] = fib(i);
    }
  };
}

var runner1 = new Runner();
var runner2 = new Runner();

// запускаем бегунов
var t1 = setInterval(function() {
  runner1.step();
}, 15);

var t2 = setTimeout(function go() {
  runner2.step();
  t2 = setTimeout(go, 15);
}, 15);

// кто сделает больше шагов?
setTimeout(function() {
  clearInterval(t1);
  clearTimeout(t2);
  alert( runner1.steps );
  alert( runner2.steps );
}, 5000);
```

Если бы в шаге step() не было вызова doSomethingHeavy(), то есть он бы не требовал времени, то количество шагов было бы почти равным.

Но так как у нас шаг, всё же, что-то делает, и функция doSomethingHeavy() специально написана таким образом, что она требует (небольшого) времени, то первый бегун успеет сделать больше шагов. Ведь в setTimeout пауза 15 мс будет между шагами, а setInterval шагает равномерно, каждые 15 мс. Получается чаще.

[К условию](#)

Функция-задержка

```
function delay(f, ms) {
  return function() {
    var savedThis = this;
    var savedArgs = arguments;

    setTimeout(function() {
      f.apply(savedThis, savedArgs);
    }, ms);
  };
}

function f(x) {
  alert( x );
}

var f1000 = delay(f, 1000);
var f1500 = delay(f, 1500);

f1000("тест"); // выведет "тест" через 1000 миллисекунд
f1500("тест2"); // выведет "тест2" через 1500 миллисекунд
```

Обратим внимание на то, как работает обёртка:

```
return function() {
  var savedThis = this;
  var savedArgs = arguments;

  setTimeout(function() {
    f.apply(savedThis, savedArgs);
  }, ms);
};
```

Именно обёртка возвращается декоратором delay и будет вызвана. Чтобы передать аргумент и контекст функции, вызываемой через ms миллисекунд, они копируются в локальные переменные savedThis и savedArgs.

Это один из самых простых, и в то же время удобных способов передать что-либо в функцию, вызываемую через `setTimeout`.

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Вызов не чаще чем в N миллисекунд

```
function debounce(f, ms) {
  var state = null;
  var COOLDOWN = 1;

  return function() {
    if (state) return;

    f.apply(this, arguments);

    state = COOLDOWN;

    setTimeout(function() { state = null }, ms);
  }
}

function f(x) { alert(x) }
var f = debounce(f, 1000);

f(1); // 1, выполнится сразу же
f(2); // игнор

setTimeout( function() { f(3) }, 100); // игнор (прошло только 100 мс)
setTimeout( function() { f(4) }, 1100); // 4, выполнится
setTimeout( function() { f(5) }, 1500); // игнор
```

Вызов `debounce` возвращает функцию-обёртку. Все необходимые данные для неё хранятся в замыкании.

При вызове ставится таймер и состояние `state` меняется на константу `COOLDOWN` («в процессе охлаждения»).

Последующие вызовы игнорируются, пока таймер не обнулит состояние.

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Тормозилка

```
function throttle(func, ms) {
  var isThrottled = false,
      savedArgs,
      savedThis;

  function wrapper() {
    if (isThrottled) { // (2)
      savedArgs = arguments;
      savedThis = this;
      return;
    }

    func.apply(this, arguments); // (1)

    isThrottled = true;

    setTimeout(function() {
      isThrottled = false; // (3)
      if (savedArgs) {
        wrapper.apply(savedThis, savedArgs);
        savedArgs = savedThis = null;
      }
    }, ms);
  }

  return wrapper;
}
```

Шаги работы этой функции:

1. Декоратор `throttle` возвращает функцию-обёртку `wrapper`, которая при первом вызове запускает `func` и переходит в состояние «паузы» (`isThrottled = true`).
2. В этом состоянии все новые вызовы запоминаются в замыкании через `savedArgs/savedThis`. Обратим внимание, что и контекст вызова и аргументы для нас одинаково важны и запоминаются одновременно. Только зная и то и другое, можно воспроизвести вызов правильно.
3. Далее, когда пройдёт таймаут `ms` миллисекунд – пауза будет снята, а `wrapper` – запущен с последними аргументами и контекстом (если во время паузы были вызовы).

Шаг (3) запускает именно не саму функцию, а снова `wrapper`, так как необходимо не только выполнить `func`, но и снова поставить выполнение на паузу. Получается последовательность «вызов – пауза... вызов – пауза ... вызов – пауза ...», каждое выполнение в обязательном

порядке сопровождается паузой после него. Это удобно описывается рекурсией.

[Открыть решение с тестами в песочнице.](#)

[К условию](#)

Запуск кода из строки: eval

Eval-калькулятор

Вычислить любое выражение нам поможет `eval` :

```
var expr = prompt("Введите выражение?", '2*3+2');
alert( eval(expr) );
```

При этом посетитель потенциально может делать все, что угодно.

Чтобы ограничить выражения только математикой, вводимую строку нужно проверять при помощи [регулярных выражений](#) на наличие любых символов, кроме букв, пробелов и знаков пунктуации.

[К условию](#)

Перехват ошибок, "try..catch"

Finally или просто код?

Разница в поведении станет очевидной, если рассмотреть код внутри функции.

Поведение будет различным, если управление каким-то образом выпрыгнет из `try..catch` .

Например, `finally` сработает после `return` , но до передачи управления внешнему коду:

```
function f() {
  try {
    ...
    return result;
  } catch (e) {
    ...
  } finally {
    очистить ресурсы
  }
}
```

Или же управление может выпрыгнуть из-за `throw` :

```
function f() {
  try {
    ...
  } catch (e) {
    ...
    if(не умею обрабатывать эту ошибку) {
      throw e;
    }
  } finally {
    очистить ресурсы
  }
}
```

В этих случаях именно `finally` гарантирует выполнение кода до окончания работы `f` , просто код не будет вызван.

[К условию](#)

Eval-калькулятор с ошибками

Вычислить любое выражение нам поможет `eval` :

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

Считываем выражение в цикле `while(true)` . Если при вычислении возникает ошибка – ловим её в `try..catch` .

Ошибкой считается, в том числе, получение NaN из eval, хотя при этом исключение не возникает. Можно бросить своё исключение в этом случае.

Код решения:

```
var expr, res;

while (true) {
  expr = prompt("Введите выражение?", '2-');
  if (expr == null) break;

  try {
    res = eval(expr);
    if (isNaN(res)) {
      throw new Error("Результат неопределён");
    }

    break;
  } catch (e) {
    alert( "Ошибка: " + e.message + ", повторите ввод" );
  }
}

alert( res );
```

[К условию](#)

Внутренний и внешний интерфейс

Добавить метод и свойство кофеварке

Кофеварка с новым методом:

```
function CoffeeMachine(power) {
  this.waterAmount = 0;

  var WATER_HEAT_CAPACITY = 4200;
  var timerId;
  var self = this;

  function getBoilTime() {
    return self.waterAmount * WATER_HEAT_CAPACITY * 80 / power;
  }

  function onReady() {
    alert( 'Кофе готово!' );
  }

  this.run = function() {
    timerId = setTimeout(onReady, getBoilTime());
  };

  this.stop = function() {
    clearTimeout(timerId)
  };
}

var coffeeMachine = new CoffeeMachine(50000);
coffeeMachine.waterAmount = 200;

coffeeMachine.run();
coffeeMachine.stop(); // кофе приготовлен не будет
```

[К условию](#)

Геттеры и сеттеры

Написать объект с геттерами и сеттерами

Решение:

```
function User() {

  var firstName, surname;

  this.setFirstName = function(newFirstName) {
    firstName = newFirstName;
  };

  this.setSurname = function(newSurname) {
    surname = newSurname;
  };

  this.getFullName = function() {
    return firstName + ' ' + surname;
  }
}
```

```
var user = new User();
user.setFirstName("Петя");
user.setSurname("Иванов");

alert( user.getFullName() ); // Петя Иванов
```

Обратим внимание, что для «геттера» `getFullName` нет соответствующего свойства объекта, он конструирует ответ «на лету». Это нормально. Одна из целей существования геттеров/сеттеров – как раз и есть изоляция внутренних свойств объекта, чтобы можно было их как угодно менять, генерировать «на лету», а внешний интерфейс оставался тем же.

[К условию](#)

Добавить геттер для power

```
function CoffeeMachine(power, capacity) {
  //...
  this.setWaterAmount = function(amount) {
    if (amount < 0) {
      throw new Error("Значение должно быть положительным");
    }
    if (amount > capacity) {
      throw new Error("Нельзя залить воды больше, чем " + capacity);
    }

    waterAmount = amount;
  };

  this.getWaterAmount = function() {
    return waterAmount;
  };

  this.getPower = function() {
    return power;
  };
}
```

[К условию](#)

Добавить публичный метод кофеварке

В решении ниже `addWater` будет просто вызывать `setWaterAmount`.

```
function CoffeeMachine(power, capacity) {
  var waterAmount = 0;

  var WATER_HEAT_CAPACITY = 4200;

  function getTimeToBoil() {
    return waterAmount * WATER_HEAT_CAPACITY * 80 / power;
  }

  this.setWaterAmount = function(amount) {
    if (amount < 0) {
      throw new Error("Значение должно быть положительным");
    }
    if (amount > capacity) {
      throw new Error("Нельзя залить больше, чем " + capacity);
    }

    waterAmount = amount;
  };

  this.addWater = function(amount) {
    this.setWaterAmount(waterAmount + amount);
  };

  function onReady() {
    alert( 'Кофе готов!' );
  }

  this.run = function() {
    setTimeout(onReady, getTimeToBoil());
  };
}

var coffeeMachine = new CoffeeMachine(100000, 400);
coffeeMachine.addWater(200);
coffeeMachine.addWater(100);
coffeeMachine.addWater(300); // Нельзя залить больше..
coffeeMachine.run();
```

[К условию](#)

Создать сеттер для onReady

```
function CoffeeMachine(power, capacity) {
  var waterAmount = 0;

  var WATER_HEAT_CAPACITY = 4200;
```



```

function getTimeToBoil() {
    return waterAmount * WATER_HEAT_CAPACITY * 80 / power;
}

this.setWaterAmount = function(amount) {
    // ... проверки пропущены для краткости
    waterAmount = amount;
};

this.getWaterAmount = function(amount) {
    return waterAmount;
};

function onReady() {
    alert( 'Кофе готов!' );
}

```

```

this.setOnReady = function(newOnReady) {
    onReady = newOnReady;
};

```

```

this.run = function() {
    setTimeout(function() {
        onReady();
    }, getTimeToBoil());
};

```

```

}

```

```

var coffeeMachine = new CoffeeMachine(20000, 500);
coffeeMachine.setWaterAmount(150);

```

```

coffeeMachine.run();

```

```

coffeeMachine.setOnReady(function() {
    var amount = coffeeMachine.getWaterAmount();
    alert( 'Готов кофе: ' + amount + 'мл' ); // Готов кофе: 150 мл
});

```

Обратите внимание на два момента в решении:

1. В сеттере `setOnReady` параметр называется `newOnReady`. Мы не можем назвать его `onReady`, так как тогда изнутри сеттера мы никак не доберёмся до внешнего (старого значения):

```

// нерабочий вариант
this.setOnReady = function(onReady) {
    onReady = onReady; // ??? внешняя переменная onReady недоступна
};

```

2. Чтобы `setOnReady` можно было вызывать в любое время, в `setTimeout` передаётся не `onReady`, а анонимная функция `function() { onReady() }`, которая возьмёт текущий (установленный последним) `onReady` из замыкания.

[К условию](#)

Добавить метод `isRunning`

Код решения модифицирует функцию `run` и добавляет приватный идентификатор таймера `timerId`, по наличию которого мы судим о состоянии кофеварки:

```

function CoffeeMachine(power, capacity) {
    var waterAmount = 0;

    var timerId;

    this.isRunning = function() {
        return !!timerId;
    };

    var WATER_HEAT_CAPACITY = 4200;

    function getTimeToBoil() {
        return waterAmount * WATER_HEAT_CAPACITY * 80 / power;
    }

    this.setWaterAmount = function(amount) {
        // ... проверки пропущены для краткости
        waterAmount = amount;
    };

    this.getWaterAmount = function(amount) {
        return waterAmount;
    };

    function onReady() {
        alert( 'Кофе готов!' );
    }

    this.setOnReady = function(newOnReady) {
        onReady = newOnReady;
    };

    this.run = function() {
        timerId = setTimeout(function() {
            timerId = null;
            onReady();
        }, getTimeToBoil());
    };
}

```

```
}

var coffeeMachine = new CoffeeMachine(20000, 500);
coffeeMachine.setWaterAmount(100);

alert( 'До: ' + coffeeMachine.isRunning() ); // До: false

coffeeMachine.run();
alert( 'В процессе: ' + coffeeMachine.isRunning() ); // В процессе: true

coffeeMachine.setOnReady(function() {
  alert( "После: " + coffeeMachine.isRunning() ); // После: false
});
```

[К условию](#)

Функциональное наследование

Запускать только при включённой кофеварке

Изменения в методе run :

```
this.run = function() {
  if (!this._enabled) {
    throw new Error("Кофеварка выключена");
  }

  setTimeout(onReady, 1000);
};
```

[Открыть решение в песочнице.](#)

[К условию](#)

Останавливать кофеварку при выключении

[Открыть решение в песочнице.](#)

[К условию](#)

Унаследуйте холодильник

Решение:

```
function Fridge(power) {
  // унаследовать
  Machine.apply(this, arguments);

  var food = []; // приватное свойство food

  this.addFood = function() {
    if (!this._enabled) {
      throw new Error("Холодильник выключен");
    }
    if (food.length + arguments.length >= this._power / 100) {
      throw new Error("Нельзя добавить, не хватает мощности");
    }
    for (var i = 0; i < arguments.length; i++) {
      food.push(arguments[i]); // добавить всё из arguments
    }
  };

  this.getFood = function() {
    // копируем еду в новый массив, чтобы манипуляции с ним не меняли food
    return food.slice();
  };
}
```

[К условию](#)

Добавьте методы в холодильник

```
function Machine(power) {
  this._power = power;
  this._enabled = false;

  var self = this;

  this.enable = function() {
    self._enabled = true;
  };
}
```

```

    this.disable = function() {
        self._enabled = false;
    };
}

function Fridge(power) {
    // унаследовать
    Machine.apply(this, arguments);

    var food = []; // приватное свойство food

    this.addFood = function() {
        if (!this._enabled) {
            throw new Error("Холодильник выключен");
        }
        if (food.length + arguments.length >= this._power / 100) {
            throw new Error("Нельзя добавить, не хватает мощности");
        }
        for (var i = 0; i < arguments.length; i++) {
            food.push(arguments[i]); // добавить всё из arguments
        }
    };

    this.getFood = function() {
        // копируем еду в новый массив, чтобы манипуляции с ним не меняли food
        return food.slice();
    };

    this.filterFood = function(filter) {
        return food.filter(filter);
    };

    this.removeFood = function(item) {
        var idx = food.indexOf(item);
        if (idx != -1) food.splice(idx, 1);
    };
}

var fridge = new Fridge(500);
fridge.enable();
fridge.addFood({
    title: "котлета",
    calories: 100
});
fridge.addFood({
    title: "сок",
    calories: 30
});
fridge.addFood({
    title: "зелень",
    calories: 10
});
fridge.addFood({
    title: "варенье",
    calories: 150
});

var dietItems = fridge.filterFood(function(item) {
    return item.calories < 50;
});

fridge.removeFood("нет такой еды"); // без эффекта
alert( fridge.getFood().length ); // 4

dietItems.forEach(function(item) {
    alert( item.title ); // сок, зелень
    fridge.removeFood(item);
});

alert( fridge.getFood().length ); // 2

```

[К условию](#)

Переопределите disable

```

function Machine(power) {
    this._power = power;
    this._enabled = false;

    var self = this;

    this.enable = function() {
        self._enabled = true;
    };

    this.disable = function() {
        self._enabled = false;
    };
}

function Fridge(power) {
    Machine.apply(this, arguments);

    var food = []; // приватное свойство food

    this.addFood = function() {
        if (!this._enabled) {
            throw new Error("Холодильник выключен");
        }
        if (food.length + arguments.length >= this._power / 100) {
            throw new Error("Нельзя добавить, не хватает мощности");
        }
        for (var i = 0; i < arguments.length; i++) {

```

```

    food.push(arguments[i]); // добавить всё из arguments
  }
};

this.getFood = function() {
  // копируем еду в новый массив, чтобы манипуляции с ним не меняли food
  return food.slice();
};

this.filterFood = function(filter) {
  return food.filter(filter);
};

this.removeFood = function(item) {
  var idx = food.indexOf(item);
  if (idx !== -1) food.splice(idx, 1);
};

var parentDisable = this.disable;
this.disable = function() {
  if (food.length) {
    throw new Error("Нельзя выключить: внутри еда");
  }
  parentDisable();
};
}

var fridge = new Fridge(500);
fridge.enable();
fridge.addFood("кус-кус");
fridge.disable(); // ошибка, в холодильнике есть еда

```

[К условию](#)

Прототип объекта

Чему равно свойство после delete?

1. true, свойство взято из rabbit.
2. null, свойство взято из animal.
3. undefined, свойства больше нет.

[К условию](#)

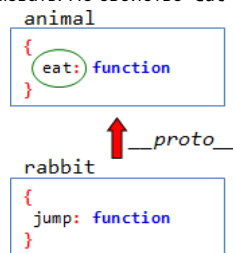
Прототип и this

Ответ: свойство будет записано в **rabbit**.

Если коротко – то потому что **this** будет указывать на **rabbit**, а прототип при записи не используется.

Если в деталях – посмотрим как выполняется **rabbit.eat()**:

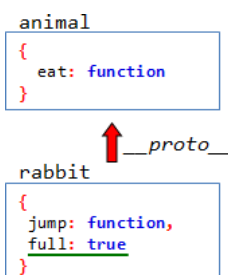
1. Интерпретатор ищет **rabbit.eat**, чтобы его вызвать. Но свойство **eat** отсутствует в объекте **rabbit**, поэтому он идет по ссылке



rabbit.__proto__ и находит это свойство там.

2. Функция **eat** запускается. Контекст ставится равным объекту перед точкой, т.е. **this = rabbit**.

Итак – получается, что команда **this.full = true** устанавливает свойство **full** в самом объекте **rabbit**. Итог:



Эта задача демонстрирует, что несмотря на то, в каком прототипе находится свойство, это никак не влияет на установку `this`, которая осуществляется по своим, независимым правилам.

[К условию](#)

Алгоритм для поиска

1. Расставим `__proto__`:

```
var head = {
  glasses: 1
};

var table = {
  pen: 3
};
table.__proto__ = head;

var bed = {
  sheet: 1,
  pillow: 2
};
bed.__proto__ = table;

var pockets = {
  money: 2000
};
pockets.__proto__ = bed;

alert( pockets.pen ); // 3
alert( bed.glasses ); // 1
alert( table.money ); // undefined
```

2. В современных браузерах, с точки зрения производительности, нет разницы, брать свойство из объекта или прототипа. Они запоминают, где было найдено свойство и в следующий раз при запросе, к примеру, `pockets.glasses` начнут искать сразу в прототипе (`head`).

[К условию](#)

Свойство `F.prototype` и создание объектов через `new`

Прототип после создания

Результат: `true`, из прототипа

Результат: `true`. Свойство `prototype` всего лишь задаёт `__proto__` у новых объектов. Так что его изменение не повлияет на `rabbit.__proto__`. Свойство `eats` будет получено из прототипа.

Результат: `false`. Свойство `Rabbit.prototype` и `rabbit.__proto__` указывают на один и тот же объект. В данном случае изменения вносятся в сам объект.

Результат: `true`, так как `delete rabbit.eats` попытается удалить `eats` из `rabbit`, где его и так нет. А чтение в `alert` произойдёт из прототипа.

Результат: `undefined`. Удаление осуществляется из самого прототипа, поэтому свойство `rabbit.eats` больше взять неоткуда.

[К условию](#)

Аргументы по умолчанию

Можно прототипно унаследовать от `options` и добавлять/менять опции в наследнике:

```
function Menu(options) {
  options = Object.create(options);
  options.width = options.width || 300;

  alert( options.width ); // возьмёт width из наследника
  alert( options.height ); // возьмёт height из исходного объекта
  ...
}
```

Все изменения будут происходить не в самом `options`, а в его наследнике, при этом исходный объект останется незатронутым.

[К условию](#)

Есть ли разница между вызовами?

Совместимость

[К условию](#)

Создать объект тем же конструктором

Да, можем, но только если уверены, что кто-то позаботился о том, чтобы значение `constructor` было верным.

В частности, без вмешательства в прототип код точно работает, например:

```
function User(name) {
  this.name = name;
}

var obj = new User('Вася');
var obj2 = new obj.constructor('Петя');

alert( obj2.name ); // Петя (сработало)
```

Сработало, так как `User.prototype.constructor == User`.

Но если кто-то, к примеру, перезапишет `User.prototype` и забудет указать `constructor`, то такой фокус не пройдет, например:

```
function User(name) {
  this.name = name;
}
User.prototype = {}; // (*)

var obj = new User('Вася');
var obj2 = new obj.constructor('Петя');

alert( obj2.name ); // undefined
```

Почему `obj2.name` равен `undefined`? Вот как это работает:

1. При вызове `new obj.constructor('Петя')`, `obj` ищет у себя свойство `constructor` – не находит.
2. Обращается к своему свойству `__proto__`, которое ведёт к прототипу.
3. Прототипом будет `(*)`, пустой объект.
4. Далее здесь также ищется свойство `constructor` – его нет.
5. Где ищем дальше? Правильно – у следующего прототипа выше, а им будет `Object.prototype`.
6. Свойство `Object.prototype.constructor` существует, это встроенный конструктор объектов, который, вообще говоря, не предназначен для вызова с аргументом-строкой, поэтому создаст совсем не то, что ожидается, но то же самое, что вызов `new Object('Петя')`, и у такого объекта не будет `name`.

[К условию](#)

Встроенные "классы" в JavaScript

Добавить функциям `defer`

```
Function.prototype.defer = function(ms) {
  setTimeout(this, ms);
}

function f() {
  alert( "привет" );
}

f.defer(1000); // выведет "привет" через 1 секунду
```

[К условию](#)

Добавить функциям `defer` с аргументами

```
Function.prototype.defer = function(ms) {
  var f = this;
  return function() {
    var args = arguments,
        context = this;
    setTimeout(function() {
      f.apply(context, args);
    }, ms);
  };
}
```

```
    }, ms);
  }
}

// проверка
function f(a, b) {
  alert( a + b );
}

f.defer(1000)(1, 2); // выведет 3 через 1 секунду.
```

[К условию](#)

Свои классы на прототипах

Перепишите в виде класса

```
function CoffeeMachine(power) {
  // свойства конкретной кофеварки
  this._power = power;
  this._waterAmount = 0;
}

// свойства и методы для всех объектов класса
CoffeeMachine.prototype.WATER_HEAT_CAPACITY = 4200;

CoffeeMachine.prototype._getTimeToBoil = function() {
  return this._waterAmount * this.WATER_HEAT_CAPACITY * 80 / this._power;
};

CoffeeMachine.prototype.run = function() {
  setTimeout(function() {
    alert( 'Кофе готов!' );
  }, this._getTimeToBoil());
};

CoffeeMachine.prototype.setWaterAmount = function(amount) {
  this._waterAmount = amount;
};

var coffeeMachine = new CoffeeMachine(10000);
coffeeMachine.setWaterAmount(50);
coffeeMachine.run();
```

[К условию](#)

Хомяки с __proto__

Почему возникает проблема

[К условию](#)

Наследование классов в JavaScript

Найдите ошибку в наследовании

Ошибка в строке:

```
Rabbit.prototype = Animal.prototype;
```

Эта ошибка приведёт к тому, что `Rabbit.prototype` и `Animal.prototype` – один и тот же объект. В результате методы `Rabbit` будут помещены в него и, при совпадении, перезапишут методы `Animal`.

Получится, что все животные прыгают, вот пример:

```
function Animal(name) {
  this.name = name;
}

Animal.prototype.walk = function() {
  alert("ходит " + this.name);
};

function Rabbit(name) {
  this.name = name;
}

Rabbit.prototype = Animal.prototype;

Rabbit.prototype.walk = function() {
  alert("прыгает! и ходит: " + this.name);
};
```

```
};  
  
var animal = new Animal("Хрюшка");  
animal.walk(); // прыгает! и ходит Хрюшка
```

Правильный вариант этой строки:

```
Rabbit.prototype = Object.create(Animal.prototype);
```

Если так написать, то в `Rabbit.prototype` будет отдельный объект, который прототипно наследует от `Animal.prototype`, но может содержать и свои свойства, специфичные для кроликов.

[К условию](#)

В чём ошибка в наследовании

Ошибка – в том, что метод `walk` присваивается в конструкторе `Animal` самому объекту вместо прототипа.

Поэтому, если мы решим перезаписать этот метод своим, специфичным для кролика, то он не сработает:

```
// ...  
  
// записывается в прототип  
Rabbit.prototype.walk = function() {  
  alert( "прыгает " + this.name );  
};
```

Метод `this.walk` из `Animal` записывается в сам объект, и поэтому он всегда будет первым, игнорируя цепочку прототипов.

Правильно было бы определять `walk` как `Animal.prototype.walk`.

Тем более, что этот метод является общим для всех объектов, тратить память и время на запись его в каждый конструктор определённо ни к чему.

[К условию](#)

Класс "часы"

```
function Clock(options) {  
  this._template = options.template;  
}  
  
Clock.prototype._render = function render() {  
  var date = new Date();  
  
  var hours = date.getHours();  
  if (hours < 10) hours = '0' + hours;  
  
  var min = date.getMinutes();  
  if (min < 10) min = '0' + min;  
  
  var sec = date.getSeconds();  
  if (sec < 10) sec = '0' + sec;  
  
  var output = this._template.replace('h', hours).replace('m', min).replace('s', sec);  
  
  console.log(output);  
};  
  
Clock.prototype.stop = function() {  
  clearInterval(this._timer);  
};  
  
Clock.prototype.start = function() {  
  this._render();  
  var self = this;  
  this._timer = setInterval(function() {  
    self._render();  
  }, 1000);  
};
```

[Открыть решение в песочнице.](#)

[К условию](#)

Класс "расширенные часы"

Наследник:

```
function ExtendedClock(options) {  
  Clock.apply(this, arguments);  
  this._precision = +options.precision || 1000;  
}
```



```
ExtendedClock.prototype = Object.create(Clock.prototype);

ExtendedClock.prototype.start = function() {
  this._render();
  var self = this;
  this._timer = setInterval(function() {
    self._render();
  }, this._precision);
};
```

[Открыть решение в песочнице.](#)

[К условию](#)

Меню с таймером для анимации

Обратите внимание: константы состояний перенесены в прототип, чтобы `AnimatingMenu` их тоже унаследовал.

[Открыть решение в песочнице.](#)

[К условию](#)

Что содержит constructor?

Нет, не распознает, выведет `false`.

Свойство `constructor` содержится в `prototype` функции по умолчанию, интерпретатор не поддерживает его корректность. Посмотрим, чему оно равно и откуда оно будет взято в данном случае.

Порядок поиска свойства `rabbit.constructor`, по цепочке прототипов:

1. `rabbit` – это пустой объект, в нём нет.
2. `Rabbit.prototype` – в него при помощи `Object.create` записан пустой объект, наследующий от `Animal.prototype`. Поэтому `constructor`'а в нём также нет.
3. `Animal.prototype` – у функции `Animal` свойство `prototype` никто не менял. Поэтому оно содержит `Animal.prototype.constructor == Animal`.

```
function Animal() {}

function Rabbit() {}
Rabbit.prototype = Object.create(Animal.prototype);

var rabbit = new Rabbit();
```

```
alert( rabbit.constructor == Rabbit ); // false
alert( rabbit.constructor == Animal ); // true
```

[К условию](#)

Проверка класса: "instanceof"

Странное поведение instanceof

Да, это выглядит достаточно странно, поскольку объект `a` не создавался функцией `B`.

Но методу `instanceof` на самом деле вообще не важна функция. Он смотрит на её `prototype` и сверяет его с цепочкой `__proto__` объекта.

В данном случае `a.__proto__ == B.prototype`, поэтому `instanceof` возвращает `true`.

По логике `instanceof` именно прототип задаёт «тип объекта», поэтому `instanceof` работает именно так.

[К условию](#)

Что выведет instanceof?

Да, распознает.

Он проверяет наследование с учётом цепочки прототипов.

```
function Animal() {}

function Rabbit() {}
Rabbit.prototype = Object.create(Animal.prototype);
```

```
var rabbit = new Rabbit();

alert( rabbit instanceof Rabbit ); // true
alert( rabbit instanceof Animal ); // true
alert( rabbit instanceof Object ); // true
```

[К условию](#)

Свои ошибки, наследование от Error

Унаследуйте от SyntaxError

```
function FormatError(message) {
  this.name = "FormatError";

  this.message = message;

  if (Error.captureStackTrace) {
    Error.captureStackTrace(this, this.constructor);
  } else {
    this.stack = (new Error()).stack;
  }
}

FormatError.prototype = Object.create(SyntaxError.prototype);
FormatError.prototype.constructor = FormatError;

// Использование

var err = new FormatError("ошибка форматирования");

alert( err.message ); // ошибка форматирования
alert( err.name ); // FormatError
alert( err.stack ); // стек на момент генерации ошибки

alert( err instanceof SyntaxError ); // true
```

[К условию](#)

Promise

Промисифицировать setTimeout

```
function delay(ms) {
  return new Promise((resolve, reject) => {
    setTimeout(resolve, ms);
  });
}
```

[К условию](#)

Загрузить массив последовательно

Для последовательной загрузки нужно организовать промисы в цепочку, чтобы они выполнялись строго – один после другого.

Вот код, который это делает:

```
// начало цепочки
let chain = Promise.resolve();

let results = [];

// в цикле добавляем задачи в цепочку
urls.forEach(function(url) {
  chain = chain
    .then(() => httpGet(url))
    .then((result) => {
      results.push(result);
    });
});

// в конце – выводим результаты
chain.then(() => {
  alert(results);
});
```

Использование `Promise.resolve()` как начала асинхронной цепочки – очень распространённый приём.

[Открыть решение в песочнице.](#) ↗

