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!

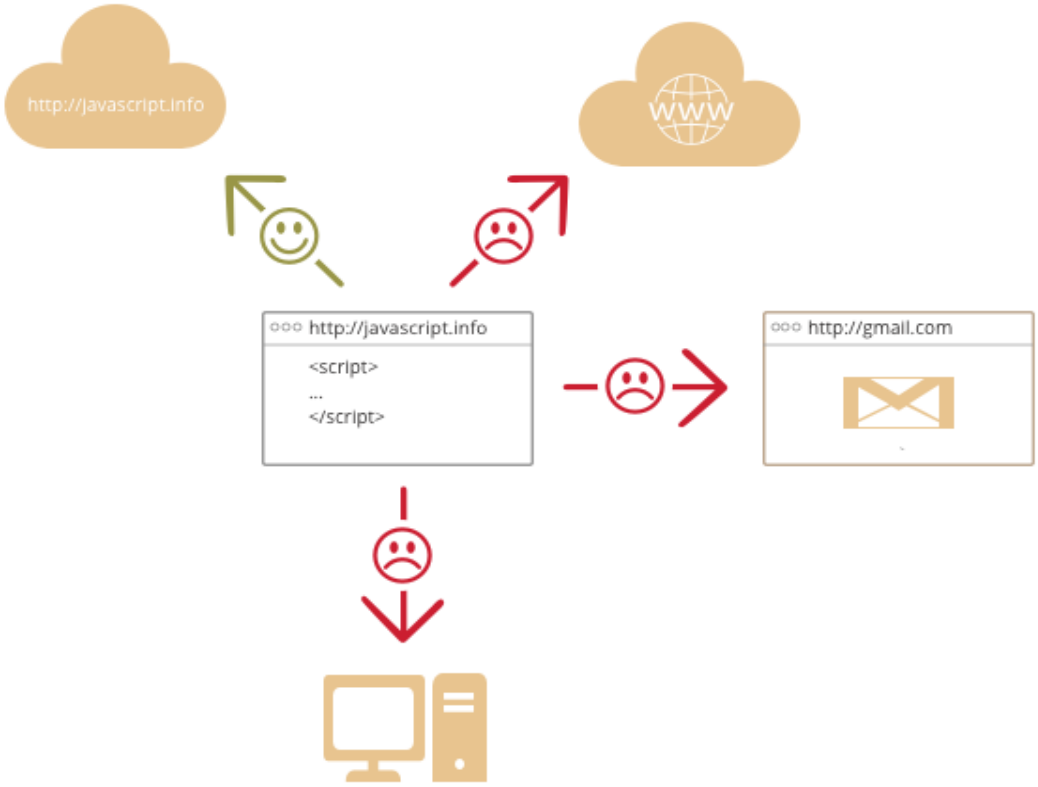
Что НЕ умеет JavaScript?

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 спрятан, 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-ролик – 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.

Основы JavaScript

Привет, мир!

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

Тег SCRIPT

А побыстрее?

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».

Кодировка и тег `META`

При попытке сделать такой же файл у себя на диске и запустить, вы можете столкнуться с проблемой – выводятся «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` необязателен.

Атрибут `<script language=…>`

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

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.

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"></scri
pt>
```

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 функциональность.

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.

`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.

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 строчку 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-слэшем «`*/`», 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' );
```

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 операнду. 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 вычислятся самое правое выражение `2+2`, присвоится в `c`, затем выполнится `b = c` и, наконец, `a = b`.

Оператор `"="` возвращает значение

Все операторы возвращают значение. Вызов `x = выражение` 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 ее значение в `a`. Так что в `a` попадёт значение `i` *после* увеличения.

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

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

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

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

```
var i = 0;

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

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

```
var i = 0;

alert( ++i ); // 1
```

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

```
var i = 0;

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

Инкремент/декремент можно использовать в любых выражениях

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

```
var i = 1;
alert( 2 * ++i ); // 4
var i = 1;
alert( 2 * i++ ); // 2,  выполнился раньше но значение вернул старое
alert( i ); // 2
alert( 2 * i++ ); // 4
alert( i ); // 3
```

При этом, нужно с осторожностью использовать такую запись, потому что в более длинной строке при быстром «вертикальном» чтении кода легко пропустить такой `i++`, и будет неочевидно, что переменая увеличивается.

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

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

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 запятая

Один из самых необычных операторов – запятая `','`.

Его можно вызвать явным образом, например:

```
var a = (5, 6);

alert( a );
```

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

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

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

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

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

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