ОСНОВЫ JAVASCRIPT

В этой части учебника мы изучаем собственно JavaScript, сам язык.
Но нам нужна рабочая среда для запуска наших скриптов, и, поскольку это онлайн-книга, то браузер будет хорошим выбором. В этой главе мы сократим количество специфичных для браузера команд (например, `alert`) до минимума, чтобы вы не тратили на них время, если планируете сосредоточиться на другой среде (например, Node.js).
Итак, сначала давайте посмотрим, как выполнить скрипт на странице. Для серверных сред (например, Node.js), вы можете выполнить скрипт с помощью команды типа `"node my.js"`.
Для браузера всё немного иначе.

Тег «script»

Программы на JavaScript могут быть вставлены в любое место HTML-документа с помощью тега `<script>`.
Для примера

```
<!DOCTYPE HTML>
<html>

<body>

  <p>Перед скриптом...</p>

  <script>
    alert( 'Привет, мир!' );
  </script>

  <p>...После скрипта.</p>

</body>

</html>
```

Вы можете запустить пример, нажав на кнопку «Play» в правом верхнем углу блока с кодом выше.
Тег `<script>` содержит JavaScript-код, который автоматически выполнится, когда браузер его обработает.

Современная разметка

Тег `<script>` имеет несколько атрибутов, которые редко используются, но всё ещё могут встретиться в старом коде:

Атрибут `type`: `<script type=…>`

Старый стандарт HTML, HTML4, требовал наличия этого атрибута в теге `<script>`. Обычно он имел значение `type="text/javascript"`. На текущий момент этого больше не требуется. Более того, в современном стандарте HTML смысл этого атрибута полностью изменился. Теперь он может использоваться для JavaScript-модулей. Но это тема не для начального уровня, и о ней мы поговорим в другой части учебника.

Атрибут `language`: `<script language=…>`

Этот атрибут должен был задавать язык, на котором написан скрипт. Но так как JavaScript является языком по умолчанию, в этом атрибуте уже нет необходимости.

Обёртывание скрипта в HTML-комментарии.

В очень древних книгах и руководствах вы сможете найти комментарии внутри тега `<script>`, например, такие:

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

Этот комментарий скрывал код JavaScript в старых браузерах, которые не знали, как обрабатывать тег `<script>`. Поскольку все браузеры, выпущенные за последние 15 лет, не содержат данной проблемы, такие комментарии уже не нужны. Если они есть, то это признак, что перед нами очень древний код.

Внешние скрипты

Если у вас много JavaScript-кода, вы можете поместить его в отдельный файл.

Файл скрипта можно подключить к HTML с помощью атрибута `src`:

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

Здесь `/path/to/script.js` – это абсолютный путь до скрипта от корня сайта. Также можно указать относительный путь от текущей страницы. Например, `src="script.js"` будет означать, что файл `"script.js"` находится в текущей папке.

Можно указать и полный URL-адрес. Например:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js"></script>
```

Для подключения нескольких скриптов используйте несколько тегов:

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

На заметку:

Как правило, только простейшие скрипты помещаются в HTML. Более сложные выделяются в отдельные файлы.

Польза от отдельных файлов в том, что браузер загрузит скрипт отдельно и сможет хранить его в кеше.

Другие страницы, которые подключают тот же скрипт, смогут брать его из кеша вместо повторной загрузки из сети. И таким образом файл будет загружаться с сервера только один раз.

Это сокращает расход трафика и ускоряет загрузку страниц.

Если атрибут `src` установлен, содержимое тега `script` будет игнорироваться.

В одном теге `<script>` нельзя использовать одновременно атрибут `src` и код внутри.

Нижеприведённый пример не работает:

```
<script src="file.js">
  alert(1); // содержимое игнорируется, так как есть атрибут src
</script>
```

Нужно выбрать: либо внешний скрипт `<script src="…">`, либо обычный код внутри тега `<script>`.

Вышеприведённый пример можно разделить на два скрипта:

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

Итого

- Для добавления кода JavaScript на страницу используется тег `<script>`
- Атрибуты `type` и `language` необязательны.
- Скрипт во внешнем файле можно вставить с помощью `<script src="path/to/script.js"></script>`.

Нам ещё многое предстоит изучить про браузерные скрипты и их взаимодействие со страницей. Но, как уже было сказано, эта часть учебника посвящена именно языку JavaScript, поэтому здесь мы постараемся не отвлекаться на детали реализации в браузере. Мы воспользуемся браузером для запуска JavaScript, это удобно для онлайн-демонстраций, но это только одна из платформ, на которых работает этот язык.

Структура кода

Начнём изучение языка с рассмотрения основных «строительных блоков» кода.

Инструкции

Инструкции – это синтаксические конструкции и команды, которые выполняют действия.
Мы уже видели инструкцию `alert('Привет, мир!')`, которая отображает сообщение «Привет, мир!».
В нашем коде может быть столько инструкций, сколько мы захотим. Инструкции могут отделяться точкой с запятой.
Например, здесь мы разделили сообщение «Привет Мир» на два вызова alert:

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

Обычно каждую инструкцию пишут на новой строке, чтобы код было легче читать:

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

Точка с запятой

В большинстве случаев точку с запятой можно не ставить, если есть переход на новую строку.
Так тоже будет работать:

```
alert('Привет')
alert('Мир')
```

В этом случае JavaScript интерпретирует перенос строки как «неявную» точку с запятой. Это называется автоматическая вставка точки с запятой.
В большинстве случаев новая строка подразумевает точку с запятой. Но «в большинстве случаев» не значит «всегда»!
В некоторых ситуациях новая строка всё же не означает точку с запятой. Например:

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

Код выведет `6`, потому что JavaScript не вставляет здесь точку с запятой. Интуитивно очевидно, что, если строка заканчивается знаком `"+"`, значит, это «незавершённое выражение», поэтому точка с запятой не требуется. И в этом случае всё работает, как задумано.
Но есть ситуации, где JavaScript «забывает» вставить точку с запятой там, где она нужна.
Ошибки, которые при этом появляются, достаточно сложно обнаруживать и исправлять.

Пример ошибки

Если вы хотите увидеть конкретный пример такой ошибки, обратите внимание на этот код:

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

Пока нет необходимости знать значение скобок `[]` и `forEach`. Мы изучим их позже.
Пока что просто запомните результат выполнения этого кода: выводится `1`, а затем `2`.
А теперь добавим `alert` перед кодом и *не* поставим в конце точку с запятой:

```
alert("Сейчас будет ошибка")

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

Теперь, если запустить код, выведется только первый `alert`, а затем мы получим ошибку!
Всё исправится, если мы поставим точку с запятой после `alert`:

```
alert("Теперь всё в порядке");

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

Теперь мы получим сообщение «Теперь всё в порядке», следом за которым будут `1` и `2`.
В первом примере без точки с запятой возникает ошибка, потому что JavaScript не вставляет точку с запятой перед квадратными скобками `[...]`. И поэтому код в первом примере выполняется, как одна инструкция. Вот как движок видит его:

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

Но это должны быть две отдельные инструкции, а не одна. Такое слияние в данном случае неправильное, оттого и ошибка. Это может произойти и в некоторых других ситуациях.

Мы рекомендуем ставить точку с запятой между инструкциями, даже если они отделены переносами строк. Это правило широко используется в сообществе разработчиков. Стоит отметить ещё раз – в большинстве случаев *можно* не ставить точку с запятой. Но безопаснее, особенно для новичка, ставить её.

Комментарии

Со временем программы становятся всё сложнее и сложнее. Возникает необходимость добавлять *комментарии*, которые бы описывали, что делает код и почему.

Комментарии могут находиться в любом месте скрипта. Они не влияют на его выполнение, поскольку движок просто игнорирует их.

Однострочные комментарии начинаются с двойной косой черты `//`.

Часть строки после `//` считается комментарием. Такой комментарий может как занимать строку целиком, так и находиться после инструкции.

Как здесь:

```
// Этот комментарий занимает всю строку
alert('Привет');

alert('Мир'); // Этот комментарий следует за инструкцией
```

Многострочные комментарии начинаются косой чертой со звёздочкой `/*` и заканчиваются звёздочкой с косой чертой `*/`.

Как вот здесь:

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

Содержимое комментария игнорируется, поэтому, если мы поместим внутри код `/* … */`, он не будет исполняться.

Это бывает удобно для временного отключения участка кода:

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

Используйте горячие клавиши!

В большинстве редакторов строку кода можно закомментировать, нажав комбинацию клавиш `Ctrl+/` для однострочного комментария и что-то, вроде `Ctrl+Shift+/` – для многострочных комментариев (выделите кусок кода и нажмите комбинацию клавиш). В системе Mac попробуйте `Cmd` вместо `Ctrl`.

Вложенные комментарии не поддерживаются!

Не может быть `/*...*/` внутри `/*...*/`.

Такой код «умрёт» с ошибкой:

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

Не стесняйтесь использовать комментарии в своём коде.

Комментарии увеличивают размер кода, но это не проблема. Есть множество инструментов, которые минифицируют код перед публикацией на рабочий сервер. Они убирают комментарии, так что они не содержатся в рабочих скриптах. Таким образом, комментарии никоим образом не вредят рабочему коду.

Строгий режим — "use strict"

На протяжении долгого времени JavaScript развивался без проблем с обратной совместимостью. Новые функции добавлялись в язык, в то время как старая функциональность не менялась.

Преимуществом данного подхода было то, что существующий код продолжал работать. А недостатком – что любая ошибка или несовершенное решение, принятое создателями JavaScript, застревали в языке навсегда.

Так было до 2009 года, когда появился ECMAScript 5 (ES5). Он добавил новые возможности в язык и изменил некоторые из существующих. Чтобы устаревший код работал, как и раньше, по умолчанию подобные изменения не применяются. Поэтому нам нужно явно их активировать с помощью специальной директивы: `"use strict"`.

«use strict»

Директива выглядит как строка: `"use strict"` или `'use strict'`. Когда она находится в начале скрипта, весь сценарий работает в «современном» режиме.

Например:

```
"use strict";

// этот код работает в современном режиме
...
```

Позже мы изучим функции (способ группировки команд). Забегая вперёд, заметим, что вместо всего скрипта `"use strict"` можно поставить в начале большинства видов функций. Это позволяет включить строгий режим только в конкретной функции. Но обычно люди используют его для всего файла.

Убедитесь, что «use strict» находится в начале

Проверьте, что `"use strict"` находится в первой исполняемой строке скрипта, иначе строгий режим может не включиться.

Здесь строгий режим не включён:

```
alert("some code");
// "use strict" ниже игнорируется - он должен быть в первой строке

"use strict";
// строгий режим не активирован
```

Над `"use strict"` могут быть записаны только комментарии.

Нет никакого способа отменить `use strict`

Нет директивы типа `"no use strict"`, которая возвращала бы движок к старому поведению.

Как только мы входим в строгий режим, отменить это невозможно.

Консоль браузера

В будущем, когда вы будете использовать консоль браузера для тестирования функций, обратите внимание, что `use strict` по умолчанию в ней выключен.

Иногда, когда `use strict` имеет значение, вы можете получить неправильные результаты.

Можно использовать `Shift+Enter` для ввода нескольких строк и написать в верхней строке `use strict`:

```
'use strict'; <Shift+Enter для перехода на новую строку>
//  ...ваш код...
<Enter для запуска>
```

В большинстве браузеров, включая Chrome и Firefox, это работает.

В старых браузерах консоль не учитывает такой `use strict`, там можно «оборачивать» код в функцию, вот так:

```
(function() {
  'use strict';

  // ...ваш код...
})()
```

Всегда используйте «use strict»

Нам ещё предстоит рассмотреть различия между строгим режимом и режимом «по умолчанию».

В следующих главах, изучая особенности языка, мы будем отмечать различия между строгим и стандартным режимами. К счастью, их не так много, и они действительно делают нашу жизнь лучше.

На данный момент достаточно иметь общее понимание об этом режиме:

1. Директива `"use strict"` переключает движок в «современный» режим, изменяя поведение некоторых встроенных функций. Позже в учебнике мы увидим подробности.
2. Строгий режим включается путём размещения `"use strict"` в начале скрипта или функции. Некоторые функции языка, такие как «классы» и «модули», автоматически включают строгий режим.
3. Строгий режим поддерживается всеми современными браузерами.
4. Мы рекомендуем всегда начинать скрипты с `"use strict"`. Все примеры в этом руководстве предполагают строгий режим, если (очень редко) не указано иное.

Переменные

JavaScript-приложению обычно нужно работать с информацией. Например:

1. Интернет-магазин – информация может включать продаваемые товары и корзину покупок.
2. Чат – информация может включать пользователей, сообщения и многое другое.

Переменные используются для хранения этой информации.

Переменная

Переменная – это «именованное хранилище» для данных. Мы можем использовать переменные для хранения товаров, посетителей и других данных.

Для создания переменной в JavaScript используйте ключевое слово `let`.

Приведённая ниже инструкция создаёт (другими словами: *объявляет* или *определяет*) переменную с именем «message»:

```
let message;
```

Теперь можно поместить в неё данные, используя оператор присваивания `=`:

```
let message;
message = 'Hello'; // сохранить строку
```

Строка сохраняется в области памяти, связанной с переменной. Мы можем получить к ней доступ, используя имя переменной:

```
let message;
message = 'Hello!';

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

Для краткости можно совместить объявление переменной и запись данных в одну строку:

```
let message = 'Hello!'; // определяем переменную и присваиваем ей значение

alert(message); // Hello!
```

Мы также можем объявить несколько переменных в одной строке:

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

Такой способ может показаться короче, но мы не рекомендуем его. Для лучшей читаемости объявляйте каждую переменную на новой строке.

Многострочный вариант немного длиннее, но легче для чтения:

```
let user = 'John';
let age = 25;
let message = 'Hello';
```

Некоторые люди также определяют несколько переменных в таком вот многострочном стиле:

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

…Или даже с запятой в начале строки:

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

В принципе, все эти варианты работают одинаково. Так что это вопрос личного вкуса и эстетики.

`var` вместо `let`

В старых скриптах вы также можете найти другое ключевое слово: `var` вместо `let`:

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

Ключевое слово `var` – *почти* то же самое, что и `let`. Оно объявляет переменную, но немного по-другому, «устаревшим» способом.

Есть тонкие различия между `let` и `var`, но они пока не имеют для нас значения. Мы подробно рассмотрим их в главе Устаревшее ключевое слово "var".

Аналогия из жизни

Мы легко поймём концепцию «переменной», если представим её в виде «коробки» для данных с уникальным названием на ней.

Например, переменную `message` можно представить как коробку с названием `"message"` и значением `"Hello!"`внутри:

Мы можем положить любое значение в коробку.

Мы также можем изменить его столько раз, сколько захотим:

```
let message;

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

При изменении значения старые данные удаляются из переменной:

Мы также можем объявить две переменные и скопировать данные из одной в другую.

```
let hello = 'Hello world!';
let message;
// копируем значение 'Hello world' из переменной hello в переменную message
message = hello;
// теперь две переменные содержат одинаковые данные
alert(hello); // Hello world!
alert(message); // Hello world!
```

Функциональные языки программирования

Примечательно, что существуют функциональные языки программирования, такие как Scala или Erlang, которые запрещают изменять значение переменной.

В таких языках однажды сохранённое «в коробку» значение остаётся там навсегда. Если нам нужно сохранить что-то другое, язык заставляет нас создать новую коробку (объявить новую переменную). Мы не можем использовать старую переменную.

Хотя на первый взгляд это может показаться немного странным, эти языки вполне подходят для серьёзной разработки. Более того, есть такая область, как параллельные вычисления, где это ограничение даёт определённые преимущества. Изучение такого языка (даже если вы не планируете использовать его в ближайшее время) рекомендуется для расширения кругозора.

Имена переменных

В JavaScript есть два ограничения, касающиеся имён переменных:

1. Имя переменной должно содержать только буквы, цифры или символы `$` и `_`.
2. Первый символ не должен быть цифрой.

Примеры допустимых имён:

```
let userName;
let test123;
```

Если имя содержит несколько слов, обычно используется верблюжья нотация, то есть, слова следуют одно за другим, где каждое следующее слово начинается с заглавной буквы: `myVeryLongName`.

Самое интересное – знак доллара `'$'` и подчёркивание `'_'` также можно использовать в названиях. Это обычные символы, как и буквы, без какого-либо особого значения.

Эти имена являются допустимыми:

```
let $ = 1; // объявили переменную с именем "$"
let _ = 2; // а теперь переменную с именем "_"
alert($ + _); // 3
```

Примеры неправильных имён переменных:

```
let 1a; // не может начинаться с цифры
let my-name; // дефис '-' не разрешён в имени
```

Регистр имеет значение

Переменные с именами `apple` and `AppLE` – это две разные переменные.

Нелатинские буквы разрешены, но не рекомендуются

Можно использовать любой язык, включая кириллицу или даже иероглифы, например:

```
let имя = '...';
let 我 = '...';
```

Технически здесь нет ошибки, такие имена разрешены, но есть международная традиция использовать английский язык в именах переменных. Даже если мы пишем небольшой скрипт, у него может быть долгая жизнь впереди. Людям из других стран, возможно, придётся прочесть его не один раз.

Зарезервированные имена

Существует список зарезервированных слов, которые нельзя использовать в качестве имён переменных, потому что они используются самим языком.

Например: `let`, `class`, `return` и `function` зарезервированы.

Приведённый ниже код даёт синтаксическую ошибку:

```
let let = 5; // нельзя назвать переменную "let", ошибка!
let return = 5; // также нельзя назвать переменную "return",
ошибка!
```

Создание переменной без использования `use strict`

Обычно нам нужно определить переменную перед её использованием. Но в старые времена было технически возможно создать переменную простым присвоением значения без использования `let`. Это все ещё работает, если мы не включаем `use strict` в наших файлах, чтобы обеспечить совместимость со старыми скриптами.

```
// заметка: "use strict" в этом примере не используется

num = 5; // если переменная "num" раньше не существовала, она
создаётся
alert(num); // 5
```

Это плохая практика, которая приводит к ошибке в строгом режиме

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

Константы

Чтобы объявить константную, то есть, неизменяемую переменную, используйте `const` вместо `let`:

```
const myBirthday = '18.04.1982';
```

Переменные, объявленные с помощью `const`, называются «константами». Их нельзя изменить. Попытка сделать это приведёт к ошибке:

```
const myBirthday = '18.04.1982';
myBirthday = '01.01.2001'; // ошибка, константу нельзя
перезаписать!
```

Если программист уверен, что переменная никогда не будет меняться, он может гарантировать это и наглядно донести до каждого, объявив её через `const`.

Константы в верхнем регистре

Широко распространена практика использования констант в качестве псевдонимов для трудно запоминаемых значений, которые известны до начала исполнения скрипта.

Названия таких констант пишутся с использованием заглавных букв и подчёркивания.

Например, сделаем константы для различных цветов в «шестнадцатеричном формате»:

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
const COLOR_ORANGE = "#FF7F00";
// ...когда нам нужно выбрать цвет
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

Преимущества:

- `COLOR_ORANGE` гораздо легче запомнить, чем `"#FF7F00"`.
- Гораздо легче допустить ошибку при вводе `"#FF7F00"`, чем при вводе `COLOR_ORANGE`.
- При чтении кода `COLOR_ORANGE` намного понятнее, чем `#FF7F00`.

Когда мы должны использовать для констант заглавные буквы, а когда называть их нормально? Давайте разберёмся и с этим.

Название «константа» просто означает, что значение переменной никогда не меняется. Но есть константы, которые известны до выполнения (например, шестнадцатеричное значение для красного цвета), а есть константы, которые *вычисляются* во время выполнения сценария, но не изменяются после их первоначального назначения.

Например:

```
const pageLoadTime = /* время, потраченное на загрузку веб-страницы */;
```

Значение `pageLoadTime` неизвестно до загрузки страницы, поэтому её имя записано обычными, а не прописными буквами. Но это всё ещё константа, потому что она не изменяется после назначения.

Другими словами, константы с именами, записанными заглавными буквами, используются только как псевдонимы для «жёстко закодированных» значений.

Придумывайте правильные имена

В разговоре о переменных необходимо упомянуть, что есть ещё одна чрезвычайно важная вещь.

Название переменной должно иметь ясный и понятный смысл, говорить о том, какие данные в ней хранятся.

Именование переменных – это один из самых важных и сложных навыков в программировании. Быстрый взгляд на имена переменных может показать, какой код был написан новичком, а какой – опытным разработчиком.

В реальном проекте большая часть времени тратится на изменение и расширение существующей кодовой базы, а не на написание чего-то совершенно нового с нуля. Когда мы возвращаемся к коду после какого-то промежутка времени, гораздо легче найти информацию, которая хорошо размечена. Или, другими словами, когда переменные имеют хорошие имена.

Пожалуйста, потратьте время на обдумывание правильного имени переменной перед её объявлением. Делайте так, и будете вознаграждены.

Несколько хороших правил:

- Используйте легко читаемые имена, такие как `userName` или `shoppingCart`.
- Избегайте использования аббревиатур или коротких имён, таких как `a`, `b`, `c`, за исключением тех случаев, когда вы точно знаете, что так нужно.
- Делайте имена максимально описательными и лаконичными. Примеры плохих имён: `data` и `value`. Такие имена ничего не говорят. Их можно использовать только в том случае, если из контекста кода очевидно, какие данные хранит переменная.
- Договоритесь с вашей командой об используемых терминах. Если посетитель сайта называется «user», тогда мы должны называть связанные с ним переменные `currentUser` или `newUser`, а не, к примеру, `currentVisitor` или `newManInTown`.

Звучит просто? Действительно, это так, но на практике для создания описательных и кратких имён переменных зачастую требуется подумать. Действуйте.

Повторно использовать или создавать новую переменную?

И последняя заметка. Есть ленивые программисты, которые вместо объявления новых переменных повторно используют существующие.

В результате их переменные похожи на коробки, в которые люди бросают разные предметы, не меняя на них этикетки. Что сейчас находится внутри коробки? Кто знает? Нам необходимо подойти поближе и проверить.

Такие программисты немного экономят на объявлении переменных, но теряют в десять раз больше при отладке.
Дополнительная переменная – это добро, а не зло.
Современные JavaScript-минификаторы и браузеры оптимизируют код достаточно хорошо, поэтому он не создаёт проблем с производительностью. Использование разных переменных для разных значений может даже помочь движку оптимизировать ваш код.

Итого

Мы можем объявить переменные для хранения данных с помощью ключевых слов `var`, `let` или `const`.

- `let` – это современный способ объявления.
- `var` – это устаревший способ объявления. Обычно мы вообще не используем его, но мы рассмотрим тонкие отличия от `let` в главе Устаревшее ключевое слово "var" на случай, если это всё-таки вам понадобится.
- `const` – похоже на `let`, но значение переменной не может изменяться.

Переменные должны быть названы таким образом, чтобы мы могли легко понять, что у них внутри.

Типы данных

Переменная в JavaScript может содержать любые данные. В один момент там может быть строка, а в другой – число:

```
// Не будет ошибкой
let message = "hello";
message = 123456;
```

Языки программирования, в которых такое возможно, называются «динамически типизированными». Это значит, что типы данных есть, но переменные не привязаны ни к одному из них.
Есть восемь основных типов данных в JavaScript. В этой главе мы рассмотрим их в общем, а в следующих главах поговорим подробнее о каждом.

Число

```
let n = 123;
n = 12.345;
```

Числовой тип данных (`number`) представляет как целочисленные значения, так и числа с плавающей точкой.
Существует множество операций для чисел, например, умножение `*`, деление `/`, сложение `+`, вычитание `-` и так далее.
Кроме обычных чисел, существуют так называемые «специальные числовые значения», которые относятся к этому типу данных: `Infinity`, `-Infinity` и `NaN`.

- `Infinity` представляет собой математическую бесконечность ∞. Это особое значение, которое больше любого числа.
  Мы можем получить его в результате деления на ноль:
- `alert( 1 / 0 ); // Infinity`
- Или задать его явно:
- `alert( Infinity ); // Infinity`
- `NaN` означает вычислительную ошибку. Это результат неправильной или неопределённой математической операции, например:
- `alert( "не число" / 2 ); // NaN, такое деление является ошибкой`
- Значение `NaN` «прилипчиво». Любая операция с `NaN` возвращает `NaN`:
- `alert( "не число" / 2 + 5 ); // NaN`
- Если где-то в математическом выражении есть `NaN`, то результатом вычислений с его участием будет `NaN`.

Математические операции – безопасны

Математические операции в JavaScript «безопасны». Мы можем делать что угодно: делить на ноль, обращаться со строками как с числами и т.д.

Скрипт никогда не остановится с фатальной ошибкой (не «умрёт»). В худшем случае мы получим `NaN` как результат выполнения.
Специальные числовые значения относятся к типу «число». Конечно, это не числа в привычном значении этого слова.
Подробнее о работе с числами мы поговорим в главе Числа.

BigInt

В JavaScript тип «number» не может содержать числа больше, чем 2^{53} (или меньше, чем -2^{53} для отрицательных). Это техническое ограничение вызвано их внутренним представлением. 2^{53} – это достаточно большое число, состоящее из 16 цифр, поэтому чаще всего проблем не возникает. Но иногда нам нужны действительно гигантские числа, например в криптографии или при использовании метки времени («timestamp») с микросекундами.
Тип `BigInt` был добавлен в JavaScript, чтобы дать возможность работать с целыми числами произвольной длины.
Чтобы создать значение типа `BigInt`, необходимо добавить `n` в конец числового литерала:

```
// символ "n" в конце означает, что это BigInt
const bigInt = 1234567890123456789012345678901234567890n;
```

Так как `BigInt` числа нужны достаточно редко, мы рассмотрим их в отдельной главе BigInt.

Поддержка

В данный момент `BigInt` поддерживается только в браузерах Firefox и Chrome, но не поддерживается в Safari/IE/Edge.

Строка

Строка (`string`) в JavaScript должна быть заключена в кавычки.

```
let str = "Привет";
let str2 = 'Одинарные кавычки тоже подойдут';
let phrase = `Обратные кавычки позволяют встраивать переменные ${str}`;
```

В JavaScript существует три типа кавычек.

1. Двойные кавычки: `"Привет"`.
2. Одинарные кавычки: `'Привет'`.
3. Обратные кавычки: `` `Привет` ``.

Двойные или одинарные кавычки являются «простыми», между ними нет разницы в JavaScript.
Обратные кавычки же имеют «расширенную функциональность». Они позволяют нам встраивать выражения в строку, заключая их в `${…}`. Например

```
let name = "Иван";

// Вставим переменную
alert( `Привет, ${name}!` ); // Привет, Иван!

// Вставим выражение
alert( `результат: ${1 + 2}` ); // результат: 3
```

Выражение внутри `${…}` вычисляется, и его результат становится частью строки. Мы можем положить туда всё, что угодно: переменную `name` или выражение `1 + 2`, или что-то более сложное.
Обратите внимание, что это можно делать только в обратных кавычках. Другие кавычки не имеют такой функциональности встраивания!

```
alert( "результат: ${1 + 2}" ); // результат: ${1 + 2} (двойные кавычки ничего не делают)
```

Мы рассмотрим строки более подробно в главе Строки.

Нет отдельного типа данных для одного символа.

В некоторых языках, например C и Java, для хранения одного символа, например `"a"` или `"%"`, существует отдельный тип. В языках C и Java это `char`.
В JavaScript подобного типа нет, есть только тип `string`. Строка может содержать один символ или множество.

Булевый (логический) тип

Булевый тип (`boolean`) может принимать только два значения: `true` (истина) и `false` (ложь).
Такой тип, как правило, используется для хранения значений да/нет: `true` значит «да, правильно», а `false` значит «нет, не правильно».
Например:

```
let nameFieldChecked = true; // да, поле отмечено
let ageFieldChecked = false; // нет, поле не отмечено
```

Булевые значения также могут быть результатом сравнений:

```
let isGreater = 4 > 1;

alert( isGreater ); // true (результатом сравнения будет "да")
```

Мы рассмотрим булевые значения более подробно в главе Логические операторы.

Значение «null»

Специальное значение `null` не относится ни к одному из типов, описанных выше.
Оно формирует отдельный тип, который содержит только значение `null`:

```
let age = null;
```

В JavaScript `null` не является «ссылкой на несуществующий объект» или «нулевым указателем», как в некоторых других языках.
Это просто специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно».
В приведённом выше коде указано, что переменная `age` неизвестна или не имеет значения по какой-то причине.

Значение «undefined»

Специальное значение `undefined` также стоит особняком. Оно формирует тип из самого себя так же, как и `null`.
Оно означает, что «значение не было присвоено».
Если переменная объявлена, но ей не присвоено никакого значения, то её значением будет `undefined`:

```
let x;

alert(x); // выведет "undefined"
```

Технически мы можем присвоить значение `undefined` любой переменной:

```
let x = 123;

x = undefined;

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

…Но так делать не рекомендуется. Обычно `null` используется для присвоения переменной «пустого» или «неизвестного» значения, а `undefined` – для проверок, была ли переменная назначена.

Объекты и символы

Тип `object` (объект) – особенный.
Все остальные типы называются «примитивными», потому что их значениями могут быть только простые значения (будь то строка или число, или что-то ещё). Объекты же используются для хранения коллекций данных или более сложных объектов. Мы разберёмся с ними позднее в главе Объекты после того, как узнаем больше о примитивах.
Тип `symbol` (символ) используется для создания уникальных идентификаторов объектов. Мы упоминаем здесь о нём для полноты картины, изучим этот тип после объектов.

Оператор typeof

Оператор `typeof` возвращает тип аргумента. Это полезно, когда мы хотим обрабатывать значения различных типов по-разному или просто хотим сделать проверку.
У него есть два синтаксиса:

1. Синтаксис оператора: `typeof x`.
2. Синтаксис функции: `typeof(x)`.

Другими словами, он работает со скобками или без скобок. Результат одинаковый.

Вызов `typeof x` возвращает строку с именем типа:

```
typeof undefined // "undefined"

typeof 0 // "number"

typeof 10n // "bigint"

typeof true // "boolean"

typeof "foo" // "string"

typeof Symbol("id") // "symbol"

typeof Math // "object"  (1)

typeof null // "object"  (2)

typeof alert // "function"  (3)
```

Последние три строки нуждаются в пояснении:

1. `Math` – это встроенный объект, который предоставляет математические операции и константы. Мы рассмотрим его подробнее в главе Числа. Здесь он служит лишь примером объекта.
2. Результатом вызова `typeof null` является `"object"`. Это неверно. Это официально признанная ошибка в `typeof`, сохранённая для совместимости. Конечно, `null` не является объектом. Это специальное значение с отдельным типом. Повторимся, это ошибка в языке.
3. Вызов `typeof alert` возвращает `"function"`, потому что `alert` является функцией. Мы изучим функции в следующих главах, где заодно увидим, что в JavaScript нет специального типа «функция». Функции относятся к объектному типу. Но `typeof` обрабатывает их особым образом, возвращая `"function"`. Формально это неверно, но очень удобно на практике.

Итого

В JavaScript есть 8 основных типов.

- `number` для любых чисел: целочисленных или чисел с плавающей точкой, целочисленные значения ограничены диапазоном $\pm 2^{53}$.
- `bigint` для целых чисел произвольной длины.
- `string` для строк. Строка может содержать один или больше символов, нет отдельного символьного типа.
- `boolean` для `true`/`false`.
- `null` для неизвестных значений – отдельный тип, имеющий одно значение `null`.
- `undefined` для неприсвоенных значений – отдельный тип, имеющий одно значение `undefined`.
- `object` для более сложных структур данных.
- `symbol` для уникальных идентификаторов.

Оператор `typeof` позволяет нам увидеть, какой тип данных сохранён в переменной.

- Имеет две формы: `typeof x` или `typeof(x)`.
- Возвращает строку с именем типа. Например, `"string"`.
- Для `null` возвращается `"object"` – это ошибка в языке, на самом деле это не объект.

Преобразование типов

Чаще всего операторы и функции автоматически приводят переданные им значения к нужному типу.
Например, `alert` автоматически преобразует любое значение к строке. Математические операторы преобразуют значения к числам.
Есть также случаи, когда нам нужно явно преобразовать значение в ожидаемый тип.

Пока что мы не говорим об объектах

Строковое преобразование

Строковое преобразование происходит, когда требуется представление чего-либо в виде строки.
Например, `alert(value)` преобразует значение к строке.
Также мы можем использовать функцию `String(value)`, чтобы преобразовать значение к строке:

```
let value = true;
alert(typeof value); // boolean

value = String(value); // теперь value это строка "true"
alert(typeof value); // string
```

Преобразование происходит очевидным образом. `false` становится `"false"`, `null` становится `"null"` и т.п.

Численное преобразование

Численное преобразование происходит в математических функциях и выражениях.
Например, когда операция деления `/` применяется не к числу:

```
alert( "6" / "2" ); // 3, Строки преобразуются в числа
```

Мы можем использовать функцию `Number(value)`, чтобы явно преобразовать `value` к числу:

```
let str = "123";
alert(typeof str); // string

let num = Number(str); // становится числом 123

alert(typeof num); // number
```

Явное преобразование часто применяется, когда мы ожидаем получить число из строкового контекста, например из текстовых полей форм.
Если строка не может быть явно приведена к числу, то результатом преобразования будет `NaN`. Например:

```
let age = Number("Любая строка вместо числа");

alert(age); // NaN, преобразование  не удалось
```

Правила численного преобразования:

Значение	Преобразуется в…
`undefined`	`NaN`
`null`	`0`
`true / false`	`1 / 0`
`string`	Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то `0`, иначе из непустой строки «считывается» число. При ошибке результат `NaN`.

Примеры:

```
alert( Number("   123   ") ); // 123
alert( Number("123z") );      // NaN (ошибка чтения числа в "z")
alert( Number(true) );        // 1
alert( Number(false) );       // 0
```

Учтите, что `null` и `undefined` ведут себя по-разному. Так, `null` становится нулём, тогда как `undefined` приводится к `NaN`.

Сложение „+“ объединяет строки

Почти все математические операторы выполняют численное преобразование.
Исключение составляет `+`. Если одно из слагаемых является строкой, тогда и все остальные приводятся к строкам.
Тогда они конкатенируются (присоединяются) друг к другу:

```
alert( 1 + '2' ); // '12' (строка справа)
```

```
alert( '1' + 2 ); // '12' (строка слева)
```
Так происходит, только если хотя бы один из аргументов является строкой. Во всех остальных случаях значения складываются как числа.

Логическое преобразование

Логическое преобразование самое простое.
Происходит в логических операторах (позже мы познакомимся с подобными конструкциями), но также может быть выполнено явно с помощью функции `Boolean(value)`.
Правило преобразования:

- Значения, которые интуитивно «пустые», вроде `0`, пустой строки, `null`, `undefined` и `NaN`, становятся `false`.
- Все остальные значения становятся `true`.

Например:

```
alert( Boolean(1) ); // true
alert( Boolean(0) ); // false

alert( Boolean("Привет!") ); // true
alert( Boolean("") ); // false
```

Заметим, что строчка с нулём `"0"` это `true`

Некоторые языки (к примеру, PHP) воспринимают строку `"0"` как `false`. Но в JavaScript, если строка не пустая, то она всегда `true`.

```
alert( Boolean("0") ); // true
alert( Boolean(" ") ); // пробел это тоже true (любая непустая строка это true)
```

Итого

Существует 3 наиболее широко используемых преобразований: строковое, численное и логическое.
`Строковое` – Происходит, когда нам нужно что-то вывести. Может быть вызвано с помощью `String(value)`. Для примитивных значений работает очевидным образом.
`Численное` – Происходит в математических операциях. Может быть вызвано с помощью `Number(value)`.
Преобразование подчиняется правилам:

Значение	Становится…
`undefined`	`NaN`
`null`	`0`
`true / false`	`1 / 0`
`string`	Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то `0`, иначе из непустой строки «считывается» число. При ошибке результат `NaN`.

`Логическое` – Происходит в логических операторах. Может быть вызвано с помощью `Boolean(value)`.
Подчиняется правилам:

Значение	Становится…
`0`, `null`, `undefined`, `NaN`, `""`	`false`
любое другое значение	`true`

Большую часть из этих правил легко понять и запомнить. Особые случаи, в которых часто допускаются ошибки:

- `undefined` при численном преобразовании становится `NaN`, не `0`.
- `"0"` и строки из одних пробелов типа `" "` при логическом преобразовании всегда `true`.

Операторы

Многие операторы знакомы нам ещё со школы: сложение `+`, умножение `*`, вычитание `-` и так далее.
В этой главе мы сконцентрируемся на операторах, которые в курсе математики не проходят.

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

Прежде, чем мы двинемся дальше, давайте разберёмся с терминологией.

- *Операнд* – то, к чему применяется оператор. Например, в умножении `5 * 2` есть два операнда: левый операнд равен `5`, а правый операнд равен `2`. Иногда их называют «аргументами» вместо «операндов».
- *Унарным* называется оператор, который применяется к одному операнду. Например, оператор унарный минус `"-"`меняет знак числа на противоположный

```
let 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:

```
let x = 1, y = 3;
```

- `alert( y - x ); // 2, бинарный минус`
- Формально мы говорим о двух разных операторах: унарное отрицание (один операнд: меняет знак) и бинарное вычитание (два операнда: вычитает).

Сложение строк, бинарный +

Давайте посмотрим специальные возможности операторов JavaScript, которые выходят за рамки школьной математики.
Обычно при помощи плюса `'+'` складывают числа.
Но если бинарный оператор `'+'` применить к строкам, то он их объединяет в одну:

```
let s = "моя" + "строка";
alert(s); // моястрока
```

Обратите внимание, если хотя бы один операнд является строкой, то второй будет также преобразован к строке.
Например:

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

Причём не важно, справа или слева находится операнд-строка. Правило простое: если хотя бы один из операндов является строкой, то второй будет также преобразован к строке.
Тем не менее, помните, что операции выполняются слева направо. Если перед строкой идут два числа, то числа будут сложены перед преобразованием в строку:

```
alert(2 + 2 + '1' ); // будет "41", а не "221"
```

Сложение и преобразование строк – это особенность бинарного плюса `+`. Другие арифметические операторы работают только с числами и всегда преобразуют операнды в числа.
Например, вычитание и деление:

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

Преобразование к числу, унарный плюс +

Плюс `+` существует в двух формах: бинарной, которую мы использовали выше, и унарной.
Унарный, то есть применённый к одному значению, плюс `+` ничего не делает с числами. Но если операнд не число, унарный плюс преобразует его в число.
Например:

```
// Не влияет на числа
let x = 1;
alert( +x ); // 1

let y = -2;
alert( +y ); // -2

// Преобразует нечисла в числа
```

```
alert( +true ); // 1
alert( +"" );   // 0
```
На самом деле это то же самое, что и `Number(...)`, только короче.
Необходимость преобразовывать строки в числа возникает очень часто. Например, обычно значения полей HTML-формы – это строки. А что, если их нужно, к примеру, сложить?
Бинарный плюс сложит их как строки:
```
let apples = "2";
let oranges = "3";

alert( apples + oranges ); // "23", так как бинарный плюс складывает строки
```
Поэтому используем унарный плюс, чтобы преобразовать к числу:
```
let apples = "2";
let oranges = "3";

// оба операнда предварительно преобразованы в числа
alert( +apples + +oranges ); // 5

// более длинный вариант
// alert( Number(apples) + Number(oranges) ); // 5
```
С точки зрения математика, такое изобилие плюсов выглядит странным. Но с точки зрения программиста – ничего особенного: 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 операторов

В том случае, если в выражении есть несколько операторов – порядок их выполнения определяется *приоритетом*, или, другими словами, существует определённый порядок выполнения операторов.
Из школы мы знаем, что умножение в выражении `2 * 2 + 1` выполнится раньше сложения.
Это как раз и есть «приоритет». Говорят, что умножение имеет более высокий приоритет, чем сложение.
Скобки важнее, чем приоритет, так что если мы не удовлетворены порядком по умолчанию, мы можем использовать их, чтобы изменить приоритет. Например, написать `(1 + 2) * 2`.
В JavaScript много операторов. Каждый оператор имеет соответствующий номер приоритета. Тот, у кого это число больше – выполнится раньше. Если приоритет одинаковый, то порядок выполнения – слева направо.
Отрывок из таблицы приоритетов (нет необходимости всё запоминать, обратите внимание, что у унарных операторов приоритет выше, чем у соответствующих бинарных):

Приоритет	Название	Обозначение
...	...	...
16	унарный плюс	`+`
16	унарный минус	`-`
14	умножение	`*`
14	деление	`/`
13	сложение	`+`
13	вычитание	`-`
...	...	...
3	присваивание	`=`
...	...	...

Так как «унарный плюс» имеет приоритет `16`, который выше, чем `13` у «сложения» (бинарный плюс), то в выражении `"+apples + +oranges"` сначала выполнятся унарные плюсы, а затем сложение.

Присваивание

Давайте отметим, что в таблице приоритетов также есть оператор присваивания `=`. У него один из самых низких приоритетов: `3`.

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

```
let x = 2 * 2 + 1;

alert( x ); // 5
```

Возможно присваивание по цепочке:

```
let a, b, c;

a = b = c = 2 + 2;

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

Такое присваивание работает справа-налево. Сначала вычисляется самое правое выражение `2 + 2`, и затем оно присваивается переменным слева: `c`, `b` и `a`. В конце у всех переменных будет одно значение.

Оператор `"="` возвращает значение

Все операторы возвращают значение. Для некоторых это очевидно, например сложение `+` или умножение `*`. Но и оператор присваивания не является исключением.

Вызов `x = value` записывает `value` в `x` *и возвращает его*.

Благодаря этому присваивание можно использовать как часть более сложного выражения:

```
let a = 1;
let b = 2;

let c = 3 - (a = b + 1);

alert( a ); // 3
alert( c ); // 0
```

В примере выше результатом `(a = b + 1)` будет значение, которое присваивается в `a` (то есть `3`). Потом оно используется для дальнейших вычислений.

Забавное применение присваивания, не так ли? Нам нужно понимать, как это работает, потому что иногда это можно увидеть в JavaScript-библиотеках, но писать самим в таком стиле не рекомендуется. Такие трюки не сделают ваш код более понятным или читабельным.

Остаток от деления %

Оператор взятия остатка `%`, 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
```

Возведение в степень **

Оператор возведения в степень `**` недавно добавили в язык.

Для натурального числа `b` результат `a ** b` равен `a`, умноженному на само себя `b` раз.

Например:

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

```
alert( 2 ** 3 ); // 8  (2 * 2 * 2)
alert( 2 ** 4 ); // 16 (2 * 2 * 2 * 2)
```
Оператор работает и для нецелых чисел.
Например:
```
alert( 4 ** (1/2) ); // 2 (степень 1/2 эквивалентна взятию квадратного корня)
alert( 8 ** (1/3) ); // 2 (степень 1/3 эквивалентна взятию кубического корня)
```
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:

```
let counter = 2;
counter++;          // работает как counter = counter + 1, просто запись короче
```

- `alert( counter ); // 3`
- **Декремент** `--` уменьшает на 1:

```
let counter = 2;
counter--;          // работает как counter = counter - 1, просто запись короче
```

- `alert( counter ); // 1`

Важно:

Инкремент/декремент можно применить только к переменной. Попытка использовать его на значении, типа 5++, приведёт к ошибке.

Операторы `++` и `--` могут быть расположены не только после, но и до переменной.

- Когда оператор идёт после переменной – это «постфиксная форма»: `counter++`.
- «Префиксная форма» – это когда оператор идёт перед переменной: `++counter`.

Обе эти формы записи делают одно и то же: увеличивают `counter` на `1`.
Есть ли разница между ними? Да, но увидеть её мы сможем, только если будем использовать значение, которое возвращают `++/--`.
Давайте проясним этот момент. Как мы знаем, все операторы возвращают значение. Операторы инкремент/декремент не исключение. Префиксная форма возвращает новое значение, в то время как постфиксная форма возвращает старое (до увеличения/уменьшения числа).
Чтобы увидеть разницу, вот небольшой пример:
```
let counter = 1;
let a = ++counter; // (*)
alert(a); // 2
```
В строке `(*)` *префиксная* форма увеличения `counter`, она возвращает новое значение `2`. Так что `alert` покажет `2`.
Теперь посмотрим на постфиксную форму:
```
let counter = 1;
let a = counter++; // (*) меняем ++counter на counter++
alert(a); // 1
```
В строке `(*)` *постфиксная* форма `counter++` также увеличивает `counter`, но возвращает *старое* значение (которое было до увеличения). Так что `alert` покажет `1`.
Подведём итоги:

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

```
let counter = 0;
counter++;
++counter;
```

- `alert( counter ); // 2, обе строки сделали одно и то же`
- Если хочется тут же использовать результат, то нужна префиксная форма:

```
let counter = 0;
```

- `alert( ++counter ); // 1`
- Если нужно увеличить и при этом получить значение переменной *до увеличения* – постфиксная форма:

```
let counter = 0;
```

- `alert( counter++ ); // 0`

Инкремент/декремент можно использовать в любых выражениях

Операторы `++/--` могут также использоваться внутри выражений. Их приоритет выше, чем у арифметических операций.

Например:

```
let counter = 1;
alert( 2 * ++counter ); // 4
```

Сравните с:

```
let counter = 1;
alert( 2 * counter++ ); // 2, потому что counter++ возвращает "старое" значение
```

Хотя технически всё в порядке, такая запись обычно делает код менее читабельным. Одна строка выполняет множество действий – нехорошо.

При беглом чтении кода можно с лёгкостью пропустить такой `counter++`, и будет неочевидно, что переменная увеличивается.

Лучше использовать стиль «одна строка – одно действие»:

```
let counter = 1;
alert( 2 * counter );
counter++;
```

Побитовые операторы

Побитовые операторы работают с 32-разрядными целыми числами (при необходимости приводят к ним), 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. Чтобы понять их, нам нужно углубиться в низкоуровневое представление чисел, и было бы неоптимально делать это прямо сейчас, тем более что они нам не понадобятся в ближайшее время. Если вам интересно, вы можете прочитать статью Побитовые операторы на MDN. Практично будет сделать это, когда возникнет реальная необходимость.

Сокращённая арифметика с присваиванием

Часто нужно применить оператор к переменной и сохранить результат в ней же.

Например:

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

Эту запись можно укоротить при помощи совмещённых операторов `+=` и `*=`:

```
let 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
```

Подобные краткие формы записи существуют для всех арифметических и побитовых операторов: `/=`, `-=` и так далее.

Вызов с присваиванием имеет в точности такой же приоритет, как обычное присваивание, то есть выполнится после большинства других операций:

```
let n = 2;
n *= 3 + 5;
alert( n ); // 16  (сначала выполнится правая часть, превратив выражение в n *= 8)
```

Оператор запятая

Оператор «запятая» `,` редко используется и является одним из самых необычных. Иногда он используется для написания более короткого кода, поэтому нам нужно знать его, чтобы понимать, что при этом происходит.

Оператор запятая предоставляет нам возможность вычислять несколько выражений, разделяя их запятой `,`. Каждое выражение выполняется, но возвращается результат только последнего.

Например:

```
let a = (1 + 2, 3 + 4);
alert( a ); // 7 (результат 3 + 4)
```

Первое выражение `1 + 2` выполняется, а результат отбрасывается. Затем идёт `3 + 4`, выражение выполняется и возвращается результат.

Запятая имеет очень низкий приоритет

Пожалуйста, обратите внимание, что оператор `,` имеет очень низкий приоритет, ниже `=`, поэтому скобки важны в приведённом выше примере.

Без них в `a = 1 + 2, 3 + 4` сначала выполнится `+`, суммируя числа в `a = 3, 7`, затем оператор присваивания `=` присвоит `a = 3`, а то, что идёт дальше, будет игнорировано. Всё так же, как в `(a = 1 + 2), 3 + 4`.

Зачем нам оператор, который отбрасывает всё, кроме последнего выражения?

Иногда его используют 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. Вот почему мы упоминаем их. Но обычно они не улучшают читабельность кода, поэтому стоит хорошо подумать, прежде чем их использовать.

Операторы сравнения

Многие операторы сравнения известны нам из математики:

- Больше/меньше: `a > b`, `a < b`.
- Больше/меньше или равно: `a >= b`, `a <= b`.
- Равно: `a == b`. Обратите внимание, для сравнения используется двойной знак равенства `=`. Один знак равенства `a = b` означал бы присваивание.
- Не равно. В математике обозначается символом `≠`. В JavaScript записывается как знак равенства с предшествующим ему восклицательным знаком: `a != b`.

Результат сравнения имеет логический тип

Операторы сравнения, как и другие операторы, возвращают значение. Это значение имеет логический тип.

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

Например:

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

Результат сравнения можно присвоить переменной, как и любое значение:

```
let result = 5 > 4; // результат сравнения присваивается переменной result
alert( result ); // true
```

Сравнение строк

Чтобы определить, что одна строка больше другой, JavaScript использует «алфавитный» или «лексикографический» порядок.

Другими словами, строки сравниваются посимвольно.

Например:

```
alert( 'Я' > 'А' ); // true
alert( 'Кот' > 'Код' ); // true
alert( 'Сонный' > 'Сон' ); // true
```

Алгоритм сравнения двух строк довольно прост:

1. Сначала сравниваются первые символы строк.
2. Если первый символ первой строки больше (меньше), чем первый символ второй, то первая строка больше (меньше) второй.
3. Если первые символы равны, то таким же образом сравниваются уже вторые символы строк.
4. Сравнение продолжается, пока не закончится одна из строк.
5. Если обе строки заканчиваются одновременно, то они равны. Иначе, большей считается более длинная строка.

В примерах выше сравнение `'Я' > 'А'` завершится на первом шаге, тогда как строки `"Кот"` и `"Код"` будут сравниваться посимвольно:

1. `К` равна `К`.
2. `о` равна `о`.
3. `т` больше чем `д`. На этом сравнение заканчивается. Первая строка больше.

Используется кодировка Unicode, а не настоящий алфавит

Приведённый выше алгоритм сравнения похож на алгоритм, используемый в словарях и телефонных книгах, но между ними есть и различия.

Например, в JavaScript имеет значение регистр символов. Заглавная буква `"А"` не равна строчной `"а"`. Какая же из них больше? Строчная `"а"`. Почему? Потому что строчные буквы имеют больший код во внутренней таблице кодирования, которую использует JavaScript (Unicode). Мы ещё поговорим о внутреннем представлении строк и его влиянии в главе Строки.

Сравнение разных типов

При сравнении значений разных типов JavaScript приводит каждое из них к числу.

Например:

```
alert( '2' > 1 ); // true, строка '2' становится числом 2
alert( '01' == 1 ); // true, строка '01' становится числом 1
```

Логическое значение `true` становится `1`, а `false` – `0`.

Например:

```
alert( true == 1 ); // true
alert( false == 0 ); // true
```

Забавное следствие

Возможна следующая ситуация:

- Два значения равны.
- Одно из них `true` как логическое значение, другое – `false`.

Например:

```
let a = 0;
alert( Boolean(a) ); // false

let b = "0";
alert( Boolean(b) ); // true
alert(a == b); // true!
```

С точки зрения JavaScript, результат ожидаем. Равенство преобразует значения, используя числовое преобразование, поэтому `"0"` становится `0`. В то время как явное преобразование с помощью `Boolean`использует другой набор правил.

Строгое сравнение

Использование обычного сравнения `==` может вызывать проблемы. Например, оно не отличает `0` от `false`:

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

Та же проблема с пустой строкой:

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

Это происходит из-за того, что операнды разных типов преобразуются оператором `==` к числу. В итоге, и пустая строка, и `false` становятся нулём.

Как же тогда отличать `0` от `false`?

Оператор строгого равенства `===` проверяет равенство без приведения типов.

Другими словами, если `a` и `b` имеют разные типы, то проверка `a === b` немедленно возвращает `false` без попытки их преобразования.

Давайте проверим:

```
alert( 0 === false ); // false, так как сравниваются разные типы
```

Ещё есть оператор строгого неравенства `!==`, аналогичный `!=`.

Оператор строгого равенства дольше писать, но он делает код более очевидным и оставляет меньше мест для ошибок.

Сравнение с null и undefined

Поведение `null` и `undefined` при сравнении с другими значениями – особое:

При строгом равенстве `===`

Эти значения различны, так как различны их типы.

```
alert( null === undefined ); // false
```

При нестрогом равенстве `==`

Эти значения равны друг другу и не равны никаким другим значениям. Это специальное правило языка.

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

При использовании математических операторов и других операторов сравнения `< > <= >=`

Значения `null/undefined` преобразуются к числам: `null` становится `0`, а `undefined` – `NaN`.

Посмотрим, какие забавные вещи случаются, когда мы применяем эти правила. И, что более важно, как избежать ошибок при их использовании.

Странный результат сравнения null и 0

Сравним `null` с нулём:

```
alert( null > 0 );  // (1) false
alert( null == 0 ); // (2) false
alert( null >= 0 ); // (3) true
```

С точки зрения математики это странно. Результат последнего сравнения говорит о том, что "`null` больше или равно нулю", тогда результат одного из сравнений выше должен быть `true`, но они оба ложны.

Причина в том, что нестрогое равенство и сравнения `> < >= <=` работают по-разному. Сравнения преобразуют `null` в число, рассматривая его как `0`. Поэтому выражение (3) `null >= 0` истинно, а `null > 0` ложно.

С другой стороны, для нестрогого равенства `==` значений `undefined` и `null` действует особое правило: эти значения ни к чему не приводятся, они равны друг другу и не равны ничему другому. Поэтому (2) `null == 0` ложно.

Несравнимое значение undefined

Значение `undefined` несравнимо с другими значениями:

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

Почему же сравнение `undefined` с нулём всегда ложно?

На это есть следующие причины:

- Сравнения `(1)` и `(2)` возвращают `false`, потому что `undefined` преобразуется в `NaN`, а `NaN` – это специальное числовое значение, которое возвращает `false` при любых сравнениях.

- Нестрогое равенство `(3)` возвращает `false`, потому что `undefined` равно только `null` и ничему больше.

Как избежать проблем

Зачем мы рассмотрели все эти примеры? Должны ли мы постоянно помнить обо всех этих особенностях? Не обязательно. Со временем все они станут вам знакомы, но можно избежать проблем, если следовать простому правилу.

Просто относитесь к любому сравнению с `undefined/null`, кроме строгого равенства `===`, с осторожностью.

Не используйте сравнения `>= > < <=` с переменными, которые могут принимать значения `null/undefined`, если вы не уверены в том, что делаете. Если переменная может принимать эти значения, то добавьте для них отдельные проверки.

Итого

- Операторы сравнения возвращают значения логического типа.
- Строки сравниваются посимвольно в лексикографическом порядке.
- Значения разных типов при сравнении приводятся к числу. Исключением является сравнение с помощью операторов строгого равенства/неравенства.
- Значения `null` и `undefined` равны `==` друг другу и не равны любому другому значению.
- Будьте осторожны при использовании операторов сравнений `>` и `<` с переменными, которые могут принимать значения `null/undefined`. Хорошей идеей будет сделать отдельную проверку на `null/undefined`.

Взаимодействие: alert, prompt, confirm

В этой части учебника мы разбираем «собственно JavaScript», без привязки к браузеру или другой среде выполнения.

Но так как мы будем использовать браузер как демо-среду, нам нужно познакомиться по крайней мере с несколькими функциями его интерфейса, а именно: `alert`, `prompt` и `confirm`.

alert

Синтаксис:

```
alert(message);
```

Этот код отобразит окно в браузере и приостановит дальнейшее выполнение скриптов до тех пор, пока пользователь не нажмёт кнопку «OK».

Например:

```
alert("Hello");
```

Это небольшое окно с сообщением называется *модальным окном*. Понятие *модальное* означает, что пользователь не может взаимодействовать с интерфейсом остальной части страницы, нажимать на другие кнопки и т.д. до тех пор, пока взаимодействует с окном. В данном случае – пока не будет нажата кнопка «OK».

prompt

Функция `prompt` принимает два аргумента:

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

Этот код отобразит модальное окно с текстом, полем для ввода текста и кнопками OK/Отмена.

`title`

Текст для отображения в окне.

`default`

Необязательный второй параметр, который устанавливает начальное значение в поле для текста в окне.

Пользователь может напечатать что-либо в поле ввода и нажать OK. Он также может отменить ввод нажатием на кнопку «Отмена» или нажав на клавишу `Esc`.

Вызов `prompt` вернёт текст, указанный в поле для ввода, или `null`, если ввод отменён пользователем.

Например:

```
let age = prompt('Сколько тебе лет?', 100);
alert(`Тебе ${age} лет!`); // Тебе 100 лет!
```

Для IE: всегда устанавливайте значение по умолчанию

Второй параметр является необязательным, но если не указать его, то Internet Explorer установит значение `"undefined"` в поле для ввода.

Запустите код в Internet Explorer и посмотрите на результат:

```
let test = prompt("Test");
```

Чтобы `prompt` хорошо выглядел в IE, рекомендуется всегда указывать второй параметр:

```
let test = prompt("Test", ''); // <-- для IE
```

confirm

Синтаксис:

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

Функция `confirm` отображает модальное окно с текстом вопроса `question` и двумя кнопками: OK и Отмена.

Результат `true`, если нажата кнопка OK. В других случаях – `false`.

Например:

```
let isBoss = confirm("Ты здесь главный?");
alert( isBoss ); // true, если нажата OK
```

Итого

Мы рассмотрели 3 функции браузера для взаимодействия с пользователем:

`alert`
показывает сообщение.

`prompt`
показывает сообщение и запрашивает ввод текста от пользователя. Возвращает напечатанный текст в поле ввода или `null`, если была нажата кнопка «Отмена» или `Esc` с клавиатуры.

`confirm`
показывает сообщение и ждёт, пока пользователь нажмёт OK или Отмена. Возвращает `true`, если нажата OK, и `false`, если нажата кнопка «Отмена» или `Esc` с клавиатуры.

Все эти методы являются модальными: останавливают выполнение скриптов и не позволяют пользователю взаимодействовать с остальной частью страницы до тех пор, пока окно не будет закрыто.

На все указанные методы распространяются два ограничения:

1. Расположение окон определяется браузером. Обычно окна находятся в центре.
2. Визуальное отображение окон зависит от браузера, и мы не можем изменить их вид.

Такова цена простоты. Есть другие способы показать более приятные глазу окна с богатой функциональностью для взаимодействия с пользователем, но если «навороты» не имеют значения, то данные методы работают отлично.

Условные операторы: if, '?'

Иногда нам нужно выполнить различные действия в зависимости от условий.

Для этого мы можем использовать оператор `if` и условный оператор `?`, который также называют «оператор вопросительный знак».

Оператор «if»

Оператор `if(...)` вычисляет условие в скобках и, если результат `true`, то выполняет блок кода.

Например:

```
let year = prompt('В каком году появилась спецификация ECMAScript-2015?', '');
if (year == 2015) alert( 'Вы правы!' );
```

В примере выше, условие – это простая проверка на равенство (`year == 2015`), но оно может быть и гораздо более сложным.

Если мы хотим выполнить более одной инструкции, то нужно заключить блок кода в фигурные скобки:

```
if (year == 2015) {
```

```
  alert( "Правильно!" );
  alert( "Вы такой умный!" );
}
```

Мы рекомендуем использовать фигурные скобки `{}` всегда, когда вы используете оператор `if`, даже если выполняется только одна команда. Это улучшает читабельность кода.

Преобразование к логическому типу

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

Давайте вспомним правила преобразования типов из главы Преобразование типов:

- Число `0`, пустая строка `""`, `null`, `undefined` и `NaN` становятся `false`. Из-за этого их называют «ложными» («falsy») значениями.
- Остальные значения становятся `true`, поэтому их называют «правдивыми» («truthy»).

Таким образом, код при таком условии никогда не выполнится:

```
if (0) { // 0 is falsy
  ...
}
```

…а при таком – выполнится всегда:

```
if (1) { // 1 is truthy
  ...
}
```

Мы также можем передать заранее вычисленное в переменной логическое значение в `if`, например так:

```
let cond = (year == 2015); // преобразуется к true или false

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

Блок «else»

Оператор `if` может содержать необязательный блок «else» («иначе»). Выполняется, когда условие ложно.

Например:

```
let year = prompt('В каком году появилась спецификация ECMAScript-2015?', '');

if (year == 2015) {
  alert( 'Да вы знаток!' );
} else {
  alert( 'А вот и неправильно!' ); // любое значение, кроме 2015
}
```

Несколько условий: «else if»

Иногда, нужно проверить несколько вариантов условия. Для этого используется блок `else if`.

Например:

```
let year = prompt('В каком году появилась спецификация ECMAScript-2015?', '');

if (year < 2015) {
  alert( 'Это слишком рано...' );
} else if (year > 2015) {
  alert( 'Это поздновато' );
} else {
  alert( 'Верно!' );
}
```

В приведённом выше коде, JavaScript сначала проверит `year < 2015`. Если это неверно, он переходит к следующему условию `year > 2015`. Если оно тоже ложно, тогда сработает последний `alert`.
Блоков `else if` может быть и больше. Присутствие блока `else` не является обязательным.

Условный оператор „?“

Иногда, нам нужно назначить переменную в зависимости от условия.
Например:

```
let accessAllowed;
let age = prompt('Сколько вам лет?', '');

if (age > 18) {
  accessAllowed = true;
} else {
  accessAllowed = false;
}
alert(accessAllowed);
```

Так называемый «условный» оператор «вопросительный знак» позволяет нам сделать это более коротким и простым способом.
Оператор представлен знаком вопроса `?`. Его также называют «тернарный», так как этот оператор, единственный в своём роде, имеет три аргумента.
Синтаксис:

```
let result = условие ? значение1 : значение2;
```

Сначала вычисляется `условие`: если оно истинно, тогда возвращается `значение1`, в противном случае – `значение2`.
Например:

```
let accessAllowed = (age > 18) ? true : false;
```

Технически, мы можем опустить круглые скобки вокруг `age > 18`. Оператор вопросительного знака имеет низкий приоритет, поэтому он выполняется после сравнения `>`.
Этот пример будет делать то же самое, что и предыдущий:

```
// оператор сравнения "age > 18" выполняется первым в любом случае
// (нет необходимости заключать его в скобки)
let accessAllowed = age > 18 ? true : false;
```

Но скобки делают код более читабельным, поэтому мы рекомендуем их использовать.

На заметку:

В примере выше, вы можете избежать использования оператора вопросительного знака `?`, т.к. сравнение само по себе уже возвращает `true/false`:

```
// то же самое
  let accessAllowed = age > 18;
```

Несколько операторов „?“

Последовательность операторов вопросительного знака `?` позволяет вернуть значение, которое зависит от более чем одного условия.
Например:

```
let age = prompt('Возраст?', 18);

let message = (age < 3) ? 'Здравствуй, мальш!' :
  (age < 18) ? 'Привет!' :
  (age < 100) ? 'Здравствуйте!' :
  'Какой необычный возраст!';
alert( message );
```

Поначалу может быть сложно понять, что происходит. Но при ближайшем рассмотрении мы видим, что это обычная последовательная проверка:

1. Первый знак вопроса проверяет `age < 3`.
2. Если верно – возвращает `'Здравствуй, мальш!'`. В противном случае, проверяет выражение после двоеточия „":"“, вычисляет `age < 18`.

3. Если это верно – возвращает `'Привет!'`. В противном случае, проверяет выражение после следующего двоеточия „":"“, вычисляет `age < 100`.
4. Если это верно – возвращает `'Здравствуйте!'`. В противном случае, возвращает выражение после последнего двоеточия – `'Какой необычный возраст!'`.

Вот как это выглядит при использовании `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 `?` используется в качестве замены `if`:

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

(company == 'Netscape') ?
   alert('Верно!') : alert('Неправильно.');
```

В зависимости от условия `company == 'Netscape'`, будет выполнена либо первая, либо вторая часть после `?`.

Здесь мы не присваиваем результат переменной. Вместо этого мы выполняем различный код в зависимости от условия.

Не рекомендуется использовать оператор вопросительного знака таким образом.

Несмотря на то, что такая запись короче, чем эквивалентное выражение `if`, она менее читабельна.

Вот, для сравнения, тот же код, использующий `if`:

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

if (company == 'Netscape') {
  alert('Верно!');
} else {
  alert('Неправильно.');
}
```

При чтении глаза сканируют код по вертикали. Блоки кода, занимающие несколько строк, воспринимаются гораздо легче, чем длинный горизонтальный набор инструкций.

Смысл оператора вопросительный знак `?` – вернуть то или иное значение, в зависимости от условия. Пожалуйста, используйте его именно для этого. Когда вам нужно выполнить разные ветви кода – используйте `if`.

Логические операторы

В JavaScript есть три логических оператора: `||` (ИЛИ), `&&` (И) и `!` (НЕ).

Несмотря на своё название, данные операторы могут применяться к значениям любых типов. Полученные результаты также могут иметь различный тип.

Давайте рассмотрим их подробнее.

|| (ИЛИ)

Оператор «ИЛИ» выглядит как двойной символ вертикальной черты:

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

Традиционно в программировании ИЛИ предназначено только для манипулирования булевыми значениями: в случае, если какой-либо из аргументов `true`, он вернёт `true`, в противоположной ситуации возвращается `false`.

В JavaScript, как мы увидим далее, этот оператор работает несколько иным образом. Но давайте сперва посмотрим, что происходит с булевыми значениями.

Существует всего четыре возможные логические комбинации:

```
alert( true || true );   // true
alert( false || true );  // true
alert( true || false );  // true
alert( false || false ); // false
```
Как мы можем наблюдать, результат операций всегда равен `true`, за исключением случая, когда оба аргумента `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) { // работает как if( true || false )
  alert( 'truthy!' );
}
```
Обычно оператор `||` используется в `if` для проверки истинности любого из заданных условий.
К примеру:
```
let hour = 9;

if (hour < 10 || hour > 18) {
  alert( 'Офис закрыт.' );
}
```
Можно передать и больше условий:
```
let hour = 12;
let isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {
  alert( 'Офис закрыт.' ); // это выходной
}
```
ИЛИ «||» находит первое истинное значение
Описанная выше логика соответствует традиционной. Теперь давайте поработаем с «дополнительными» возможностями JavaScript.
Расширенный алгоритм работает следующим образом.
При выполнении ИЛИ || с несколькими значениями:
```
result = value1 || value2 || value3;
```
Оператор `||` выполняет следующие действия:

- Вычисляет операнды слева направо.
- Каждый операнд конвертирует в логическое значение. Если результат `true`, останавливается и возвращает исходное значение этого операнда.
- Если все операнды являются ложными (`false`), возвращает последний из них.

Значение возвращается в исходном виде, без преобразования.
Другими словами, цепочка ИЛИ `"||"` возвращает первое истинное значение или последнее, если такое значение не найдено.
Например:
```
alert( 1 || 0 ); // 1
alert( true || 'no matter what' ); // true

alert( null || 1 ); // 1 (первое истинное значение)
alert( null || 0 || 1 ); // 1 (первое истинное значение)
alert( undefined || null || 0 ); // 0 (поскольку все ложно, возвращается последнее значение)
```
Это делает возможным более интересное применение оператора по сравнению с «чистым, традиционным, только булевым ИЛИ».

1. **Получение первого истинного значения из списка переменных или выражений.** Представим, что у нас имеется ряд переменных, которые могут содержать данные или быть `null/undefined`. Как мы можем найти первую переменную с данными?
   С помощью `||`:
2. ```
   let currentUser = null;
   let defaultUser = "John";
   ```
   ```

```
let name = currentUser || defaultUser || "unnamed";
```

3. `alert( name ); // выбирается "John" – первое истинное значение`
4. Если бы и `currentUser`, и `defaultUser` были ложными, в качестве результата мы бы наблюдали `"unnamed"`.
5. **Сокращённое вычисление.**

Операндами могут быть как отдельные значения, так и произвольные выражения. ИЛИ вычисляет их слева направо. Вычисление останавливается при достижении первого истинного значения. Этот процесс называется «сокращённым вычислением», поскольку второй операнд вычисляется только в том случае, если первого недостаточно для вычисления всего выражения.

Это хорошо заметно, когда выражение, указанное в качестве второго аргумента, имеет побочный эффект, например, изменение переменной.

В приведённом ниже примере `x` не изменяется

6.
```
let x;
true || (x = 1);
```

7. `alert(x); // undefined, потому что (x = 1) не вычисляется`
8. Если бы первый аргумент имел значение `false`, то `||` приступил бы к вычислению второго и выполнил операцию присваивания:
9.
```
    let x;

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

10. `alert(x); // 1`
11. Присваивание – лишь один пример. Конечно, могут быть и другие побочные эффекты, которые не проявятся, если вычисление до них не дойдёт.

Как мы видим, этот вариант использования `||` является "аналогом `if`". Первый операнд преобразуется в логический. Если он оказывается ложным, начинается вычисление второго.

В большинстве случаев лучше использовать «обычный» `if`, чтобы облегчить понимание кода, но иногда это может быть удобно.

&& (И)

Оператор И пишется как два амперсанда `&&`:

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

В традиционном 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`:

```
let hour = 12;
let minute = 30;

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

Как и в случае с ИЛИ, любое значение допускается в качестве операнда И:

```
if (1 && 0) { // вычисляется как true && false
  alert( "не сработает, так как результат ложный" );
}
```

И «&&» находит первое ложное значение

При нескольких подряд операторах И:

```
result = value1 && value2 && value3;
```

Оператор `&&` выполняет следующие действия:

- Вычисляет операнды слева направо.
- Каждый операнд преобразует в логическое значение. Если результат `false`, останавливается и возвращает исходное значение этого операнда.
- Если все операнды были истинными, возвращается последний.

Другими словами, И возвращает первое ложное значение. Или последнее, если ничего не найдено.
Вышеуказанные правила схожи с поведением ИЛИ. Разница в том, что И возвращает первое *ложное* значение, а ИЛИ– первое *истинное*.
Примеры:

```
// Если первый операнд истинный,
// И возвращает второй:
alert( 1 && 0 ); // 0
alert( 1 && 5 ); // 5

// Если первый операнд ложный,
// И возвращает его. Второй операнд игнорируется
alert( null && 5 ); // null
alert( 0 && "no matter what" ); // 0
```

Можно передать несколько значений подряд. В таком случае возвратится первое «ложное» значение, на котором остановились вычисления.

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

Когда все значения верны, возвращается последнее

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

Приоритет оператора `&&` больше, чем у `||`

Приоритет оператора И `&&` больше, чем ИЛИ `||`, так что он выполняется раньше.
Таким образом, код `a && b || c && d` по существу такой же, как если бы выражения `&&` были в круглых скобках: `(a && b) || (c && d)`.

Как и оператор ИЛИ, И `&&` иногда может заменять `if`.
К примеру:

```
let x = 1;

(x > 0) && alert( 'Greater than zero!' );
```

Действие в правой части `&&` выполнится только в том случае, если до него дойдут вычисления. То есть, `alert` сработает, если в левой части `(x > 0)` будет `true`.
Получился аналог:

```
let x = 1;

if (x > 0) {
  alert( 'Greater than zero!' );
}
```

Однако, как правило, вариант с `if` лучше читается и воспринимается.
Он более очевиден, поэтому лучше использовать его.

! (НЕ)

Оператор НЕ представлен восклицательным знаком `!`.
Синтаксис довольно прост:

```
result = !value;
```

Оператор принимает один аргумент и выполняет следующие 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( !!"non-empty string" ); // true
alert( !!null ); // false
```

То есть первое НЕ преобразует значение в логическое значение и возвращает обратное, а второе НЕ снова инвертирует его. В конце мы имеем простое преобразование значения в логическое.
Есть немного более подробный способ сделать то же самое – встроенная функция `Boolean`:
```
alert( Boolean("non-empty string") ); // true
alert( Boolean(null) ); // false
```
Приоритет НЕ `!` является наивысшим из всех логических операторов, поэтому он всегда выполняется первым, перед `&&`или `||`.

Циклы while и for

При написании скриптов зачастую встаёт 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 следующий синтаксис:
```
while (condition) {
  // код
  // также называемый "телом цикла"
}
```
Код из тела цикла выполняется, пока условие `condition` истинно.
Например, цикл ниже выводит `i`, пока `i < 3`:
```
let i = 0;
while (i < 3) { // выводит 0, затем 1, затем 2
  alert( i );
  i++;
}
```
Одно выполнение тела цикла по-научному называется *итерация*. Цикл в примере выше совершает три итерации.
Если бы строка `i++` отсутствовала в примере выше, то цикл бы повторялся (в теории) вечно. На практике, конечно, браузер не позволит такому случиться, он предоставит пользователю возможность остановить «подвисший» скрипт, а JavaScript на стороне сервера придётся «убить» процесс.
Любое выражение или переменная может быть условием цикла, а не только сравнение: условие `while` вычисляется и преобразуется в логическое значение.
Например, `while (i)` – более краткий вариант `while (i != 0)`:
```
let i = 3;
while (i) { // когда i будет равно 0, условие станет ложным, и цикл остановится
  alert( i );
  i--;
}
```
Фигурные скобки не требуются для тела цикла из одной строки

Если тело цикла состоит лишь из одной инструкции, мы можем опустить фигурные скобки `{…}`:
```
let i = 3;
  while (i) alert(i--);
```

Цикл «do…while»

Проверку условия можно разместить под телом цикла, используя специальный синтаксис `do..while`:
```
do {
  // тело цикла
} while (condition);
```
Цикл сначала выполнит тело, а затем проверит условие `condition`, и пока его значение равно `true`, он будет выполняться снова и снова.

Например:
```
let i = 0;
do {
  alert( i );
  i++;
} while (i < 3);
```
Такая форма синтаксиса оправдана, если вы хотите, чтобы тело цикла выполнилось **хотя бы один раз**, даже если условие окажется ложным. На практике чаще используется форма с предусловием: `while(…) {…}`.

Цикл «for»

Более сложный, но при этом самый распространённый цикл — цикл `for`.
Выглядит он так:
```
for (начало; условие; шаг) {
  // ... тело цикла ...
}
```
Давайте разберёмся, что означает каждая часть, на примере. Цикл ниже выполняет `alert(i)` для `i` от `0` до (но не включая) `3`:
```
for (let i = 0; i < 3; i++) { // выведет 0, затем 1, затем 2
  alert(i);
}
```
Рассмотрим конструкцию `for` подробней:

часть		
начало	`i = 0`	Выполняется один раз при входе в цикл
условие	`i < 3`	Проверяется *перед* каждой итерацией цикла. Если оно вычислится в `false`, цикл остановится.
шаг	`i++`	Выполняется *после* тела цикла на каждой итерации *перед* проверкой условия.
тело	`alert(i)`	Выполняется снова и снова, пока условие вычисляется в `true`.

В целом, алгоритм работы цикла выглядит следующим образом:
```
Выполнить *начало*
→ (Если *условие* == true → Выполнить *тело*, Выполнить *шаг*)
→ (Если *условие* == true → Выполнить *тело*, Выполнить *шаг*)
→ (Если *условие* == true → Выполнить *тело*, Выполнить *шаг*)
→ ...
```
То есть, *начало* выполняется один раз, а затем каждая итерация заключается в проверке *условия*, после которой выполняется *тело* и *шаг*.
Если тема циклов для вас нова, может быть полезным вернуться к примеру выше и воспроизвести его работу на листе бумаги, шаг за шагом.
Вот в точности то, что происходит в нашем случае:
```
// for (let i = 0; i < 3; i++) alert(i)

// Выполнить начало
let i = 0;
// Если условие == true → Выполнить тело, Выполнить шаг
if (i < 3) { alert(i); i++ }
// Если условие == true → Выполнить тело, Выполнить шаг
if (i < 3) { alert(i); i++ }
// Если условие == true → Выполнить тело, Выполнить шаг
if (i < 3) { alert(i); i++ }
// ...конец, потому что теперь i == 3
```

Встроенное объявление переменной

В примере переменная счётчика `i` была объявлена прямо в цикле. Это так называемое «встроенное» объявление переменной. Такие переменные существуют только внутри цикла

```
for (let i = 0; i < 3; i++) {
  alert(i); // 0, 1, 2
}
   alert(i); // ошибка, нет такой переменной
```
Вместо объявления новой переменной мы можем использовать уже существующую:
```
let i = 0;

for (i = 0; i < 3; i++) { // используем существующую переменную
  alert(i); // 0, 1, 2
}

   alert(i); // 3, переменная доступна, т.к. была объявлена
   снаружи цикла
```
Пропуск частей «for»
Любая часть `for` может быть пропущена.
Для примера, мы можем пропустить `начало` если нам ничего не нужно делать перед стартом цикла.
Вот так:
```
let i = 0; // мы уже имеем объявленную i с присвоенным значением

for (; i < 3; i++) { // нет необходимости в "начале"
  alert( i ); // 0, 1, 2
}
```
Можно убрать и `шаг`:
```
let i = 0;

for (; i < 3;) {
  alert( i++ );
}
```
Это сделает цикл аналогичным `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.
Прерывание цикла: «break»
Обычно цикл завершается при вычислении *условия* в `false`.
Но мы можем выйти из цикла в любой момент с помощью специальной директивы `break`.
Например, следующий код подсчитывает сумму вводимых чисел до тех пор, пока посетитель их вводит, а затем – выдаёт:
```
let sum = 0;

while (true) {

  let value = +prompt("Введите число", '');

  if (!value) break; // (*)

  sum += value;

}
alert( 'Сумма: ' + sum );
```
Директива `break` в строке `(*)` 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 или конце цикла, а посередине.

Переход к следующей итерации: continue

Директива `continue` – «облегчённая версия» `break`. При её выполнении цикл не прерывается, а переходит к следующей итерации (если условие все ещё равно `true`). Её используют, если понятно, что на текущем повторе цикла делать больше нечего.

Например, цикл ниже использует `continue`, чтобы выводить только нечётные значения:

```
for (let i = 0; i < 10; i++) {

  // если true, пропустить оставшуюся часть тела цикла
  if (i % 2 == 0) continue;

  alert(i); // 1, затем 3, 5, 7, 9
}
```

Для чётных значений `i`, директива `continue` прекращает выполнение тела цикла и передаёт управление на следующую итерацию `for` (со следующим числом). Таким образом `alert` вызывается только для нечётных значений.

Директива `continue` позволяет избегать вложенности

Цикл, который обрабатывает только нечётные значения, мог бы выглядеть так:

```
for (let 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 „?“

Обратите внимание, что эти синтаксические конструкции не являются выражениями и не могут быть использованы с тернарным оператором `?`. В частности, использование таких директив, как `break/continue`, вызовет ошибку.

Например, если мы возьмём этот код:

```
if (i > 5) {
  alert(i);
} else {
  continue;
}
```

…и перепишем его, используя вопросительный знак:

```
(i > 5) ? alert(i) : continue; // continue здесь приведёт к ошибке
```

…то будет синтаксическая ошибка.

Это ещё один повод не использовать оператор вопросительного знака `?` вместо `if`.

Метки для break/continue

Бывает, нужно выйти одновременно из нескольких уровней цикла сразу.

Например, в коде ниже мы проходимся циклами по `i` и `j`, запрашивая с помощью `prompt` координаты `(i, j)` с `(0,0)` до `(2,2)`:

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

  for (let j = 0; j < 3; j++) {
```

```
    let input = prompt(`Значение на координатах (${i},${j})`, '');

    // Что если мы захотим перейти к Готово (ниже) прямо отсюда?

  }
}

alert('Готово!');
```

Нам нужен способ остановить выполнение если пользователь отменит ввод.

Обычный `break` после `input` лишь прервёт внутренний цикл, но этого недостаточно. Достичь желаемого поведения можно с помощью меток.

Метка имеет вид идентификатора с двоеточием перед циклом:

```
labelName: for (...) {
  ...
}
```

Вызов `break <labelName>` в цикле ниже ищет ближайший внешний цикл с такой меткой и переходит в его конец.

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

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

    let input = prompt(`Значение на координатах (${i},${j})`, '');

    // если пустая строка или Отмена, то выйти из обоих циклов
    if (!input) break outer; // (*)

    // сделать что-нибудь со значениями...
  }
}
alert('Готово!');
```

В примере выше это означает, что вызовом `break outer` будет разорван внешний цикл до метки с именем `outer`, и управление перейдёт со строки, помеченной `(*)`, к `alert('Готово!')`.

Можно размещать метку на отдельной строке:

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

Директива `continue` также может быть использована с меткой. В этом случае управление перейдёт на следующую итерацию цикла с меткой.

Метки не позволяют «прыгнуть» куда угодно

Метки не дают возможности передавать управление в произвольное место кода.

Например, нет возможности сделать следующее:

```
break label; // не прыгает к метке ниже

label: for (...)
```

Вызов `break/continue` возможен только внутри цикла, и метка должна находиться где-то выше этой директивы.

Итого

Мы рассмотрели 3 вида циклов:

- `while` – Проверяет условие перед каждой итерацией.
- `do..while` – Проверяет условие после каждой итерации.
- `for (;;)` – Проверяет условие перед каждой итерацией, есть возможность задать 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` выйти за пределы текущего цикла, повлиять на выполнение внешнего.
Заметим, что метки не позволяют прыгнуть в произвольное место кода, в JavaScript нет такой возможности.

Конструкция "switch"

Конструкция `switch` заменяет собой сразу несколько `if`.
Она представляет собой более наглядный способ сравнить выражение сразу с несколькими вариантами.

Синтаксис

Конструкция `switch` имеет один или более блок `case` и необязательный блок `default`.
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`.

Пример работы

Пример использования `switch` (сработавший код выделен):

```
let a = 2 + 2;

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

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`

```
let a = 2 + 2;

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

В примере выше последовательно выполнятся три `alert`:

```
alert( 'В точку!' );
alert( 'Перебор' );
alert( "Нет таких значений" );
```

Любое выражение может быть аргументом для `switch/case`

И `switch` и `case` допускают любое выражение в качестве аргумента.

Например:

```
let a = "1";
let b = 0;

switch (+a) {
  case b + 1:
    alert("Выполнится, т.к. значением +a будет 1, что в точности равно b+1");
    break;

  default:
    alert("Это не выполнится");
}
```

В этом примере выражение `+a` вычисляется в `1`, что совпадает с выражением `b + 1` в `case`, и следовательно, код в этом блоке будет выполнен.

Группировка «case»

Несколько вариантов `case`, использующих один код, можно группировать.

Для примера, выполним один и тот же код для `case 3` и `case 5`, сгруппировав их:

```
let a = 2 + 2;

switch (a) {
  case 4:
    alert('Правильно!');
    break;
  case 3: // (*) группируем оба case
  case 5:
    alert('Неправильно!');
    alert("Может вам посетить урок математики?");
    break;

  default:
    alert('Результат выглядит странновато. Честно.');
```

```
}
```
Теперь оба варианта `3` и `5` выводят одно сообщение.
Возможность группировать `case` – это побочный эффект того, как `switch/case` работает без `break`. Здесь выполнение `case 3` начинается со строки `(*)` и продолжается в `case 5`, потому что отсутствует `break`.

Тип имеет значение

Нужно отметить, что проверка на равенство всегда строгая. Значения должны быть одного типа, чтобы выполнялось равенство.
Для примера, давайте рассмотрим следующий код:

```
let arg = prompt("Введите число?");
switch (arg) {
  case '0':
  case '1':
    alert( 'Один или ноль' );
    break;
  case '2':
    alert( 'Два' );
    break;
  case 3:
    alert( 'Никогда не выполнится!' );
    break;
  default:
    alert( 'Неизвестное значение' );
}
```

1. Для `'0'` и `'1'` выполнится первый `alert`.
2. Для `'2'` – второй `alert`.
3. Но для `3`, результат выполнения `prompt` будет строка `"3"`, которая не соответствует строгому равенству `===` с числом `3`. Таким образом, мы имеем «мёртвый код» в `case 3`! Выполнится вариант `default`.

Функции

Зачастую нам надо повторять одно и то же действие во многих частях программы.
Например, необходимо красиво вывести сообщение 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 функции

Для создания функций мы можем использовать *объявление функции*.
Пример объявления функции:

```
function showMessage() {
  alert( 'Всем привет!' );
}
```

Вначале идёт ключевое слово `function`, после него *имя функции*, затем список *параметров* в круглых скобках через запятую (в вышеприведённом примере он пустой) и, наконец, код функции, также называемый «телом функции», внутри фигурных скобок.

```
function имя(параметры) {
  ...тело...
}
```

Наша новая функция может быть вызвана по её имени: `showMessage()`.
Например:

```
function showMessage() {
  alert( 'Всем привет!' );
}
```

```
showMessage();
showMessage();
```
Вызов `showMessage()` выполняет код функции. Здесь мы увидим сообщение дважды.
Этот пример явно демонстрирует одно из главных предназначений 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

Переменные, объявленные внутри функции, видны только внутри этой функции.
Например:
```
function showMessage() {
  let message = "Привет, я JavaScript!"; // локальная переменная
  alert( message );
}
showMessage(); // Привет, я JavaScript!
alert( message ); // <-- будет ошибка, т.к. переменная видна только внутри функции
```

Внешние переменные

У функции есть доступ к внешним переменным, например:
```
let userName = 'Вася';

function showMessage() {
  let message = 'Привет, ' + userName;
  alert(message);
}
showMessage(); // Привет, Вася
```
Функция обладает полным доступом к внешним переменным и может изменять их значение.
Например:
```
let userName = 'Вася';
function showMessage() {
  userName = "Петя"; // (1) изменяем значение внешней переменной
  let message = 'Привет, ' + userName;
  alert(message);
}
alert( userName ); // Вася перед вызовом функции
showMessage();
alert( userName ); // Петя, значение внешней переменной было изменено функцией
```
Внешняя переменная используется, только если внутри функции нет такой локальной.
Если одноимённая переменная объявляется внутри функции, тогда она перекрывает внешнюю. Например, в коде ниже функция использует локальную переменную `userName`.
Внешняя будет проигнорирована:
```
let userName = 'Вася';
function showMessage() {
  let userName = "Петя"; // объявляем локальную переменную
  let message = 'Привет, ' + userName; // Петя
  alert(message);
}
// функция создаст и будет использовать свою собственную локальную переменную userName
showMessage();
alert( userName ); // Вася, не изменилась, функция не трогала внешнюю переменную
```

Глобальные переменные

Переменные, объявленные снаружи всех функций, такие как внешняя переменная `userName` в вышеприведённом коде – называются *глобальными*.
Глобальные переменные видимы для любой функции (если только их не перекрывают одноимённые локальные переменные).
Желательно сводить использование глобальных переменных к минимуму. В современном коде обычно мало или совсем нет глобальных переменных. Хотя они иногда полезны для хранения важнейших «общепроектовых» данных.

Параметры

Мы можем передать внутрь функции любую информацию, используя параметры (также называемые *аргументы функции*).
В нижеприведённом примере функции передаются два параметра: `from` и `text`

```
function showMessage(from, text) { // аргументы: from, text
  alert(from + ': ' + text);
}
showMessage('Аня', 'Привет!'); // Аня: Привет! (*)
showMessage('Аня', "Как дела?"); // Аня: Как дела? (**)
```

Когда функция вызывается в строках `(*)` и `(**)`, переданные значения копируются в локальные переменные `from` и `text`. Затем они используются в теле функции.
Вот ещё один пример: у нас есть переменная `from`, и мы передаём её функции. Обратите внимание: функция изменяет значение `from`, но это изменение не видно снаружи. Функция всегда получает только копию значения:

```
function showMessage(from, text) {
  from = '*' + from + '*'; // немного украсим "from"
  alert( from + ': ' + text );
}
let from = "Аня";
showMessage(from, "Привет"); // *Аня*: Привет
// значение "from" осталось прежним, функция изменила значение локальной переменной
alert( from ); // Аня
```

Параметры по умолчанию

Если параметр не указан, то его значением становится `undefined`.
Например, вышеупомянутая функция `showMessage(from, text)` может быть вызвана с одним аргументом:

```
showMessage("Аня");
```

Это не приведёт к ошибке. Такой вызов выведет `"Аня: undefined"`. В вызове не указан параметр `text`, поэтому предполагается, что `text === undefined`.
Если мы хотим задать параметру `text` значение по умолчанию, мы должны указать его после `=`:

```
function showMessage(from, text = "текст не добавлен") {
  alert( from + ": " + text );
}
showMessage("Аня"); // Аня: текст не добавлен
```

Теперь, если параметр `text` не указан, его значением будет `"текст не добавлен"`
В данном случае `"текст не добавлен"` это строка, но на её месте могло бы быть и более сложное выражение, которое бы вычислялось и присваивалось при отсутствии параметра.
Например:

```
function showMessage(from, text = anotherFunction()) {
  // anotherFunction() выполнится только если не передан text
  // результатом будет значение text
}
```

Вычисление параметров по умолчанию

В JavaScript параметры по умолчанию вычисляются каждый раз, когда функция вызывается без соответствующего параметра.

В примере выше `anotherFunction()` будет вызываться каждый раз, когда `showMessage()` вызывается без параметра `text`.

Использование параметров по умолчанию в ранних версиях JavaScript

Ранние версии JavaScript не поддерживали параметры по умолчанию. Поэтому существуют альтернативные способы, которые могут встречаться в старых скриптах.

Например, явная проверка на `undefined`:

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

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

...Или с помощью оператора `||`:

```
function showMessage(from, text) {
  // Если значение text ложно, тогда присвоить параметру text значение по умолчанию
  text = text || 'текст не добавлен';
  ...
}
```

Возврат значения

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

Простейшим примером может служить функция сложения двух чисел:

```
function sum(a, b) {
  return a + b;
}

let result = sum(1, 2);
alert( result ); // 3
```

Директива `return` может находиться в любом месте тела функции. Как только выполнение доходит до этого места, функция останавливается, и значение возвращается в вызвавший её код (присваивается переменной `result` выше).

Вызовов `return` может быть несколько, например:

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

let age = prompt('Сколько вам лет?', 18);

if ( checkAge(age) ) {
  alert( 'Доступ получен' );
} else {
  alert( 'Доступ закрыт' );
}
```

Возможно использовать `return` и без значения. Это приведёт к немедленному выходу из функции.

Например:

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

```
  alert( "Вам показывается кино" ); // (*)
  // ...
}
```

В коде выше, если `checkAge(age)` вернёт `false`, `showMovie` не выполнит `alert`.

Результат функции с пустым `return` или без него – `undefined`

Если функция не возвращает значения, это всё равно, как если бы она возвращала `undefined`:

```
function doNothing() { /* пусто */ }
  alert( doNothing() === undefined ); // true
```

Пустой `return` аналогичен `return undefined`:

```
function doNothing() {
  return;
}

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

Никогда не добавляйте перевод строки между `return` и его значением

Для длинного выражения в `return` может быть заманчиво разместить его на нескольких отдельных строках, например так:

```
return
  (some + long + expression + or + whatever * f(a) + f(b))
```

Код не выполнится, потому что интерпретатор JavaScript подставит точку с запятой после `return`. Для него это будет выглядеть так

```
return;
  (some + long + expression + or + whatever * f(a) + f(b))
```

Таким образом, это фактически стало пустым `return`.

Если мы хотим, чтобы возвращаемое выражение занимало несколько строк, нужно начать его на той же строке, что и `return`. Или, хотя бы, поставить там открывающую скобку, вот так:

```
return (
  some + long + expression
  + or +
  whatever * f(a) + f(b)
    )
```

И тогда всё сработает, как задумано.

Выбор имени функции

Функция – это действие. Поэтому имя функции обычно является глаголом. Оно должно быть простым, точным и описывать действие функции, чтобы программист, который будет читать код, получил верное представление о том, что делает функция.

Как правило, используются глагольные префиксы, обозначающие общий характер действия, после которых следует уточнение. Обычно в командах разработчиков действуют соглашения, касающиеся значений этих префиксов.

Например, функции, начинающиеся с `"show"` обычно что-то показывают.

Функции, начинающиеся с…

- `"get…"` – возвращают значение,
- `"calc…"` – что-то вычисляют,
- `"create…"` – что-то создают,
- `"check…"` – что-то проверяют и возвращают логическое значение, и т.д.

Примеры таких имён:

```
showMessage(..)     // показывает сообщение
getAge(..)          // возвращает возраст (в каком либо значении)
calcSum(..)         // вычисляет сумму и возвращает результат
createForm(..)      // создаёт форму (и обычно возвращает её)
checkPermission(..) // проверяет доступ, возвращая true/false
```

Благодаря префиксам, при первом взгляде на имя функции становится понятным что делает её код, и какое значение она может возвращать.

Одна функция – одно действие

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

Два независимых действия обычно подразумевают две функции, даже если предполагается, что они будут вызываться вместе (в этом случае мы можем создать третью функцию, которая будет их вызывать).

Несколько примеров, которые нарушают это правило:

- `getAge` – будет плохим выбором, если функция будет выводить `alert` с возрастом (должна только возвращать его).
- `createForm` – будет плохим выбором, если функция будет изменять документ, добавляя форму в него (должна только создавать форму и возвращать её).
- `checkPermission` – будет плохим выбором, если функция будет отображать сообщение с текстом `доступ разрешён/запрещён` (должна только выполнять проверку и возвращать её результат).

В этих примерах использовались общепринятые смыслы префиксов. Конечно, вы в команде можете договориться о других значениях, но обычно они мало отличаются от общепринятых. В любом случае вы и ваша команда должны точно понимать, что значит префикс, что функция с ним может делать, а чего не может.

Сверхкороткие имена функций

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

Например, во фреймворке jQuery есть функция с именем `$`. В библиотеке Lodash основная функция представлена именем `_`.

Это исключения. В основном имена функций должны быть в меру краткими и описательными.

Функции == Комментарии

Функции должны быть короткими и делать только что-то одно. Если это что-то большое, имеет смысл разбить функцию на несколько меньших. Иногда следовать этому правилу непросто, но это определённо хорошее правило.

Небольшие функции не только облегчают тестирование и отладку – само существование таких функций выполняет роль хороших комментариев!

Например, сравним ниже две функции `showPrimes(n)`. Каждая из них выводит простое число до `n`.

Первый вариант использует метку `nextPrime`:

```
function showPrimes(n) {
  nextPrime: for (let i = 2; i < n; i++) {

    for (let j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }
    alert( i ); // простое
  }
}
```

Второй вариант использует дополнительную функцию `isPrime(n)` для проверки на простое:

```
function showPrimes(n) {

  for (let i = 2; i < n; i++) {
    if (!isPrime(i)) continue;

    alert(i);  // простое
  }
}
function isPrime(n) {
  for (let i = 2; i < n; i++) {
    if ( n % i == 0) return false;
  }
```

```
  return true;
}
```
Второй вариант легче для понимания, не правда ли? Вместо куска кода мы видим название действия (`isPrime`). Иногда разработчики называют такой код *самодокументируемым*.
Таким образом, допустимо создавать функции, даже если мы не планируем повторно использовать их. Такие функции структурируют код и делают его более понятным.

Итого

Объявление функции имеет вид:
```
function имя(параметры, через, запятую) {
  /* тело, код функции */
}
```
- Передаваемые значения копируются в параметры функции и становятся локальными переменными.
- Функции имеют доступ к внешним переменным. Но это работает только изнутри наружу. Код вне функции не имеет доступа к её локальным переменным.
- Функция может возвращать значение. Если этого не происходит, тогда результат равен `undefined`.

Для того, чтобы сделать код более чистым и понятным, рекомендуется использовать локальные переменные и параметры функций, не пользоваться внешними переменными.
Функция, которая получает параметры, работает с ними и затем возвращает результат, гораздо понятнее функции, вызываемой без параметров, но изменяющей внешние переменные, что чревато побочными эффектами.
Именование функций:
- Имя функции должно понятно и чётко отражать, что она делает. Увидев её вызов в коде, вы должны тут же понимать, что она делает, и что возвращает.
- Функция – это действие, поэтому её имя обычно является глаголом.
- Есть много общепринятых префиксов, таких как: `create…`, `show…`, `get…`, `check…` и т.д. Пользуйтесь ими как подсказками, поясняющими, что делает функция.

Функции являются основными строительными блоками скриптов. Мы рассмотрели лишь основы функций в JavaScript, но уже сейчас можем создавать и использовать их. Это только начало пути. Мы будем неоднократно возвращаться к функциям и изучать их всё более и более глубоко.

Function Expression

Функция в JavaScript – это не магическая языковая структура, а особого типа значение.
Синтаксис, который мы использовали до этого, называется *Function Declaration* (Объявление Функции):
```
function sayHi() {
  alert( "Привет" );
}
```
Существует ещё один синтаксис создания функций, который называется *Function Expression* (Функциональное Выражение).
Оно выглядит вот так:
```
let sayHi = function() {
  alert( "Привет" );
};
```
В коде выше функция создаётся и явно присваивается переменной, как любое другое значение. По сути без разницы, как мы определили функцию, это просто значение, хранимое в переменной `sayHi`.
Смысл обоих примеров кода одинаков: "создать функцию и поместить её значение в переменную `sayHi`".
Мы можем даже вывести это значение с помощью `alert`:
```
function sayHi() {
  alert( "Привет" );
}
alert( sayHi ); // выведет код функции
```

Обратите внимание, что последняя строка не вызывает функцию `sayHi`, после её имени нет круглых скобок. Существуют языки программирования, в которых любое упоминание имени функции совершает её вызов. JavaScript – не один из них.
В JavaScript функции – это значения, поэтому мы и обращаемся с ними, как со значениями. Код выше выведет строковое представление функции, которое является её исходным кодом.
Конечно, функция – не обычное значение, в том смысле, что мы можем вызвать его при помощи скобок: `sayHi()`.
Но всё же это значение. Поэтому мы можем делать с ним то же самое, что и с любым другим значением.
Мы можем скопировать функцию в другую переменную:

```
function sayHi() {   // (1) создаём
  alert( "Привет" );
}
let func = sayHi;    // (2) копируем
func(); // Привет    // (3) вызываем копию (работает)!
sayHi(); // Привет   //     прежняя тоже работает (почему бы нет)
```

Давайте подробно разберём всё, что тут произошло:

1. Объявление Function Declaration `(1)` создало функцию и присвоило её значение переменной с именем `sayHi`.
2. В строке `(2)` мы скопировали её значение в переменную `func`. Обратите внимание (ещё раз): нет круглых скобок после `sayHi`. Если бы они были, то выражение `func = sayHi()` записало бы *результат вызова* `sayHi()` в переменную `func`, а не саму *функцию* `sayHi`.
3. Теперь функция может быть вызвана с помощью обеих переменных `sayHi()` и `func()`.

Заметим, что мы могли бы использовать и Function Expression для того, чтобы создать `sayHi` в первой строке:

```
let sayHi = function() {
  alert( "Привет" );
};
let func = sayHi;
// ...
```

Результат был бы таким же.

Зачем нужна точка с запятой в конце?

У вас мог возникнуть вопрос: Почему в Function Expression ставится точка с запятой `;` на конце, а в Function Declaration нет

```
function sayHi() {
  // ...
}

let sayHi = function() {
  // ...
};
```

Ответ прост:

- Нет необходимости в `;` в конце блоков кода и синтаксических конструкций, которые их используют, таких как `if { ... }`, `for { }`, `function f { }` и т.д.
- Function Expression использует внутри себя инструкции присваивания `let sayHi = ...;` как значение. Это не блок кода, а выражение с присваиванием. Таким образом, точка с запятой не относится непосредственно к Function Expression, она лишь завершает инструкцию.

Функции-«колбэки»

Рассмотрим ещё примеры функциональных выражений и передачи функции как значения.
Давайте напишем функцию `ask(question, yes, no)` с тремя параметрами:
`question`
Текст вопроса

`yes`
Функция, которая будет вызываться, если ответ будет «Yes»
`no`
Функция, которая будет вызываться, если ответ будет «No»
Наша функция должна задать вопрос `question` и, в зависимости от того, как ответит пользователь, вызвать `yes()` или `no()`:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

function showOk() {
  alert( "Вы согласны." );
}
function showCancel() {
  alert( "Вы отменили выполнение." );
}
// использование: функции showOk, showCancel передаются в качестве аргументов ask
ask("Вы согласны?", showOk, showCancel);
```

На практике подобные функции очень полезны. Основное отличие «реальной» функции `ask` от примера выше будет в том, что она использует более сложные способы взаимодействия с пользователем, чем простой вызов `confirm`. В браузерах такие функции обычно отображают красивые диалоговые окна. Но это уже другая история.
Аргументы функции `ask` ещё называют *функциями-колбэками* или просто *колбэками*.
Ключевая идея в том, что мы передаём функцию и ожидаем, что она вызовется обратно (от англ. «call back» – обратный вызов) когда-нибудь позже, если это будет необходимо. В нашем случае, `showOk` становится *колбэком*' для ответа «yes», а `showCancel` – для ответа «no».
Мы можем переписать этот пример значительно короче, используя Function Expression:

```
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 объявляются прямо внутри вызова `ask(...)`. У них нет имён, поэтому они называются *анонимными*. Такие функции недоступны снаружи `ask` (потому что они не присвоены переменным), но это как раз то, что нам нужно.
Подобный код, появившийся в нашем скрипте выглядит очень естественно, в духе JavaScript.

Функция – это значение, представляющее «действие»

Обычные значения, такие как строки или числа представляют собой *данные*.
Функции, с другой стороны, можно воспринимать как «действия».
Мы можем передавать их из переменной в переменную и запускать, когда захотим.

Function Expression в сравнении с Function Declaration

Давайте разберём ключевые отличия Function Declaration от Function Expression.
Во-первых, синтаксис: как определить, что есть что в коде.
Function Declaration: функция объявляется отдельной конструкцией «function…» в основном потоке кода.

```
// Function Declaration
function sum(a, b) {
```

```
  return a + b;
}
```

Function Expression: функция, созданная внутри другого выражения или синтаксической конструкции. В данном случае функция создаётся в правой части «выражения присваивания» =:

```
// Function Expression
let sum = function(a, b) {
  return a + b;
};
```

Более тонкое отличие состоит, в том, *когда* создаётся функция движком JavaScript.

Function Expression создаётся, когда выполнение доходит до него, и затем уже может использоваться.

После того, как поток выполнения достигнет правой части выражения присваивания `let sum = function…` – с этого момента, функция считается созданной и может быть использована (присвоена переменной, вызвана и т.д.).

С Function Declaration всё иначе.

Function Declaration можно использовать во всем скрипте (или блоке кода, если функция объявлена в блоке).

Другими словами, когда движок JavaScript *готовится* выполнять скрипт или блок кода, прежде всего он ищет в нём Function Declaration и создаёт все такие функции. Можно считать этот процесс «стадией инициализации».

И только после того, как все объявления Function Declaration будут обработаны, продолжится выполнение.

В результате, функции, созданные, как Function Declaration могут быть вызваны раньше своих определений.

Например, так будет работать:

```
sayHi("Вася"); // Привет, Вася
function sayHi(name) {
  alert( `Привет, ${name}` );
}
```

Функция `sayHi` была создана, когда движок JavaScript подготавливал скрипт к выполнению, и такая функция видна повсюду в этом скрипте.

…Если бы это было Function Expression, то такой код вызовет ошибку:

```
sayHi("Вася"); // ошибка!
let sayHi = function(name) {  // (*) магии больше нет
  alert( `Привет, ${name}` );
};
```

Функции, объявленные при помощи Function Expression, создаются тогда, когда выполнение доходит до них. Это случится только на строке, помеченной звёздочкой `(*)`. Слишком поздно.

Ещё одна важная особенность Function Declaration заключается в их блочной области видимости.

В строгом режиме, когда Function Declaration находится в блоке `{...}`, функция доступна везде внутри блока. Но не снаружи него.

Для примера давайте представим, что нам нужно создать функцию `welcome()` в зависимости от значения переменной `age`, которое мы получим во время выполнения кода. И затем запланируем использовать её когда-нибудь в будущем.

Такой код, использующий Function Declaration, работать не будет:

```
let age = prompt("Сколько Вам лет?", 18);

// в зависимости от условия объявляем функцию
if (age < 18) {

  function welcome() {
    alert("Привет!");
  }

} else {
```

```
  function welcome() {
    alert("Здравствуйте!");
  }

}

// ...не работает
welcome(); // Error: welcome is not defined
```

Это произошло, так как объявление Function Declaration видимо только внутри блока кода, в котором располагается.

Вот ещё один пример:

```
let age = 16; // присвоим для примера 16

if (age < 18) {
  welcome();                // \   (выполнится)
                            //  |
  function welcome() {      //  |
    alert("Привет!");       //  |  Function Declaration доступно
  }                         //  |  во всём блоке кода, в котором
объявлено
                            //  |
  welcome();                // /   (выполнится)
} else {
  function welcome() {
    alert("Здравствуйте!");
  }
}
// здесь фигурная скобка закрывается,
// поэтому Function Declaration, созданные внутри блока кода выше -
- недоступны отсюда.
welcome(); // Ошибка: welcome is not defined
```

Что можно сделать, чтобы `welcome` была видима снаружи `if`?

Верным подходом будет воспользоваться функцией, объявленной при помощи Function Expression, и присвоить значение `welcome` переменной, объявленной снаружи `if`, что обеспечит нам нужную видимость.

Такой код работает, как ожидалось:

```
let age = prompt("Сколько Вам лет?", 18);
let welcome;
if (age < 18) {
  welcome = function() {
    alert("Привет!");
  };
} else {
  welcome = function() {
    alert("Здравствуйте!");
  };
}
welcome(); // теперь всё в порядке
```

Можно упростить этот код ещё сильнее, используя условный оператор `?`:

```
let age = prompt("Сколько Вам лет?", 18);
let welcome = (age < 18) ?
  function() { alert("Привет!"); } :
  function() { alert("Здравствуйте!"); };
welcome(); // теперь всё в порядке
```

Когда использовать Function Declaration, а когда Function Expression?

Как правило, если нам понадобилась функция, в первую очередь нужно рассматривать синтаксис Function Declaration, который мы использовали до этого. Он даёт нам больше свободы в том, как мы можем организовывать код. Функции, объявленные таким образом, можно вызывать до их объявления.
Также функции вида `function f(…) {…}` чуть более заметны в коде, чем `let f = function(…) {…}`. Function Declaration легче «ловятся глазами».
…Но если Function Declaration нам не подходит по какой-то причине (мы рассмотрели это в примере выше), то можно использовать объявление при помощи Function Expression.

Итого

- Функции – это значения. Они могут быть присвоены, скопированы или объявлены в другом месте кода.
- Если функция объявлена как отдельная инструкция в основном потоке кода, то это Function Declaration.
- Если функция была создана как часть выражения, то считается, что эта функция объявлена при помощи Function Expression.
- Function Declaration обрабатываются перед выполнением блока кода. Они видны во всём блоке.
- Функции, объявленные при помощи Function Expression, создаются, только когда поток выполнения достигает их.

В большинстве случаев, когда нам нужно создать функцию, предпочтительно использовать Function Declaration, т.к. функция будет видима до своего объявления в коде. Это позволяет более гибко организовывать код, и улучшает его читаемость.
Таким образом, мы должны прибегать к объявлению функций при помощи Function Expression в случае, когда синтаксис Function Declaration не подходит для нашей задачи. Мы рассмотрели несколько таких примеров в этой главе, и рассмотрим их ещё больше в будущем.

Функции-стрелки, основы

Существует ещё более простой и краткий синтаксис для создания функций, который часто лучше, чем синтаксис Function Expression.
Он называется «функции-стрелки» или «стрелочные функции» (arrow functions), т.к. выглядит следующим образом:

```
let func = (arg1, arg2, ...argN) => expression
```

…Такой код создаёт функцию `func` с аргументами `arg1..argN` и вычисляет `expression` с правой стороны с их использованием, возвращая результат.
Другими словами, это более короткий вариант такой записи:

```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

Давайте взглянем на конкретный пример:

```
let sum = (a, b) => a + b;
/* Более короткая форма для:
let sum = function(a, b) {
  return a + b;
};
*/
alert( sum(1, 2) ); // 3
```

То есть, `(a, b) => a + b` задаёт функцию с двумя аргументами `a` и `b`, которая при запуске вычисляет выражение справа `a + b` и возвращает его результат.

- Если у нас только один аргумент, то круглые скобки вокруг параметров можно опустить, сделав запись ещё короче

```
// тоже что и
// let double = function(n) { return n * 2 }
let double = n => n * 2;
```

- `alert( double(3) ); // 6`
- Если нет аргументов, указываются пустые круглые скобки:

```
let sayHi = () => alert("Hello!");
```
- `sayHi();`

Функции-стрелки могут быть использованы так же, как и Function Expression.
Например, для динамического создания функции:
```
let age = prompt("Сколько Вам лет?", 18);
let welcome = (age < 18) ?
  () => alert('Привет') :
  () => alert("Здравствуйте!");
welcome(); // теперь всё в порядке
```
Поначалу функции-стрелки могут показаться необычными и трудночитаемыми, но это быстро пройдёт, как только глаза привыкнут к этим конструкциям.
Они очень удобны для простых однострочных действий, когда лень писать много букв.

Многострочные стрелочные функции

В примерах выше аргументы использовались слева от `=>`, а справа вычислялось выражение с их значениями.
Порой нам нужно что-то посложнее, например, выполнить несколько инструкций. Это также возможно, нужно лишь заключить инструкции в фигурные скобки. И использовать `return` внутри них, как в обычной функции.
Например:
```
let sum = (a, b) => {  // фигурная скобка, открывающая тело многострочной функции
  let result = a + b;
  return result; // при фигурных скобках для возврата значения нужно явно вызвать return
};
alert( sum(1, 2) ); // 3
```
Дальше будет ещё информация

Здесь мы рассмотрели функции-стрелки как способ писать меньше букв. Но это далеко не всё!
А пока мы можем использовать их для простых однострочных действий и колбэков.

Итого

Функции-стрелки очень удобны для однострочных действий. Они бывают двух типов:

1. Без фигурных скобок: `(...args) => expression` – правая сторона выражение: функция выполняет его и возвращает результат.
2. С фигурными скобками: `(...args) => { body }` – скобки позволяют нам писать многострочные инструкции внутри функции, но при этом необходимо указывать директиву `return`, чтобы вернуть какое-либо значение.

Перепишите с использованием функции-стрелки

Замените код Function Expression стрелочной функцией:
```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}
ask(
  "Вы согласны?",
  function() { alert("Вы согласились."); },
  function() { alert("Вы отменили выполнение."); }
);
```

Особенности JavaScript

Давайте кратко повторим изученный материал и отметим наиболее «тонкие» моменты.

Структура кода

Инструкции разделяются точкой с запятой:
```
alert('Привет'); alert('Мир');
```

Как правило, перевод строки также интерпретируется как разделитель, так тоже будет работать:

```
alert('Привет')
alert('Мир')
```

Это так называемая «автоматическая вставка точки с запятой». Впрочем, она не всегда срабатывает, например:

```
alert("После этого сообщения ждите ошибку")
[1, 2].forEach(alert)
```

Большинство руководств по стилю кода рекомендуют ставить точку с запятой после каждой инструкции.

Точка с запятой не требуется после блоков кода {…} и синтаксических конструкций с ними, таких как, например, циклы:

```
function f() {
  // после объявления функции необязательно ставить точку с запятой
}
for(;;) {
  // после цикла точка с запятой также необязательна
}
```

…Впрочем, если даже мы и поставим «лишнюю» точку с запятой, ошибки не будет. Она просто будет проигнорирована.

Строгий режим

Чтобы по максимуму использовать возможности современного JavaScript, все скрипты рекомендуется начинать с добавления директивы `"use strict"`.

```
'use strict';
...
```

Эту директиву следует размещать в первой строке скрипта или в начале тела функции.

Без `"use strict"` код также запустится, но некоторые возможности будут работать в «режиме совместимости» со старыми версиями языка JavaScript. Нам же предпочтительнее современное поведение.

Некоторые конструкции языка (например, классы, которые нам ещё предстоит изучить) включают строгий режим по умолчанию.

Переменные

Можно объявить при помощи:

- `let`
- `const` (константа, т.е. изменению не подлежит)
- `var` (устаревший способ, подробности позже)

Имя переменной может включать:

- Буквы и цифры, однако цифра не может быть первым символом.
- Символы `$` и `_` используются наряду с буквами.
- Иероглифы и символы нелатинского алфавита также допустимы, но обычно не используются.

Переменные типизируются динамически. В них могут храниться любые значения:

```
let x = 5;
x = "Вася";
```

Всего существует 7 типов данных:

- `number` для целых и вещественных чисел,
- `string` для строк,
- `boolean` для логических значений истинности или ложности: `true/false`,
- `null` – тип с единственным значением `null`, т.е. «пустое значение» или «значение не существует»,
- `undefined` – тип с единственным значением `undefined`, т.е. «значение не задано»,
- `object` и `symbol` – сложные структуры данных и уникальные идентификаторы; их мы ещё не изучили.

Оператор `typeof` возвращает тип значения переменной, с двумя исключениями:

```
typeof null == "object" // ошибка в языке
typeof function(){} == "function" // именно для функций
```

Взаимодействие с посетителем

В качестве рабочей среды мы используем браузер, так что простейшими функциями взаимодействия с посетителем являются:

`prompt(question, [default])`

Задаёт вопрос `question` и возвращает то, что ввёл посетитель, либо `null`, если посетитель нажал на кнопку «Отмена».

`confirm(question)`

Задаёт вопрос `question` и предлагает выбрать «ОК» или «Отмена». Выбор возвращается в формате `true/false`.

`alert(message)`

Выводит сообщение `message`.

Все эти функции показывают *модальные окна*, они останавливают выполнение кода и не позволяют посетителю взаимодействовать со страницей, пока не будет дан ответ на вопрос.

Например:

```
let userName = prompt("Введите имя", "Алиса");
let isTeaWanted = confirm("Вы хотите чаю?");
alert( "Посетитель: " + userName ); // Алиса
alert( "Чай: " + isTeaWanted ); // true
```

Операторы

JavaScript поддерживает следующие операторы:

Арифметические

Простые `* + - /`, а также деление по модулю `%` и возведение в степень `**`.

Бинарный плюс `+` объединяет строки. А если одним из операндов является строка, то второй тоже будет конвертирован в строку:

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

Операторы присваивания

Простые `a = b` и составные `a *= 2`.

Битовые операции

Битовые операторы работают с 32-битными целыми числами на самом низком, побитовом уровне. Подробнее об их использовании можно прочитать на ресурсе MDN.

Условный оператор

Единственный оператор с тремя параметрами: `cond ? resultA : resultB`. Если условие `cond` истинно, возвращается `resultA`, иначе – `resultB`.

Логические операторы

Логические И `&&`, ИЛИ `||` используют так называемое «ленивое вычисление» и возвращают значение, на котором оно остановилось (не обязательно `true` или `false`). Логическое НЕТ `!` конвертирует операнд в логический тип и возвращает инвертированное значение.

Сравнение

Проверка на равенство `==` значений разных типов конвертирует их в число (за исключением `null` и `undefined`, которые могут равняться только друг другу), так что примеры ниже равны:

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

Другие операторы сравнения тоже конвертируют значения разных типов в числовой тип.

Оператор строгого равенства `===` не выполняет конвертирования: разные типы для него всегда означают разные значения.

Значения `null` и `undefined` особенные: они равны `==` только друг другу, но не равны ничему ещё.

Операторы сравнения больше/меньше сравнивают строки посимвольно, остальные типы конвертируются в число.

Другие операторы

Существуют и другие операторы, такие как запятая.

Циклы

Мы изучили три вида циклов:

```
// 1
while (condition) {
```

```
  ...
}
// 2
do {
  ...
} while (condition);
// 3
for(let i = 0; i < 10; i++) {
  ...
  }
```

- Переменная, объявленная в цикле `for(let...)`, видна только внутри цикла. Но мы также можем опустить `let` и переиспользовать существующую переменную.
- Директивы `break/continue` позволяют выйти из цикла/текущей итерации. Используйте метки для выхода из вложенных циклов.

Позже мы изучим ещё виды циклов для работы с объектами.

Конструкция «switch»

Конструкция «switch» может заменить несколько проверок `if`. При сравнении она использует оператор строгого равенства `===`.

Например:

```
let age = prompt('Сколько вам лет?', 18);
switch (age) {
  case 18:
    alert("Так не сработает"); // результатом prompt является
строка, а не число
  case "18":
    alert("А так сработает!");
    break;

  default:
    alert("Любое значение, неравное значению выше");
}
```

Функции

Мы рассмотрели три способа создания функции в JavaScript:

Function Declaration: функция в основном потоке кода

```
function sum(a, b) {
  let result = a + b;
  return result;
}
```

1.

Function Expression: функция как часть выражения

```
let sum = function(a, b) {
  let result = a + b;
  return result;
};
```

2.

Стрелочные функции:

```
// выражение в правой части
let sum = (a, b) => a + b;
// многострочный код в фигурных скобках { ... }, здесь нужен
return:
let sum = (a, b) =>
  // ...
  return a + b;
}
// без аргументов
let sayHi = () => alert("Привет");
// с одним аргументом
```

3. `let double = n => n * 2;`

- У функций могут быть локальные переменные: т.е. объявленные в теле функции. Такие переменные видимы только внутри функции.
- У параметров могут быть значения по умолчанию: `function sum(a = 1, b = 2) {...}`.
- Функции всегда что-нибудь возвращают. Если нет оператора `return`, результатом будет `undefined`.