Методическое пособие:

Проектирование и разработка веб-приложений. Разработка на стороне клиента

Оглавление

[1. Введение 3](#_Toc50721558)

[Для кого предназначено данное методическое пособие 3](#_Toc50721559)

[Необходимые начальные знания 3](#_Toc50721560)

[2. Основы JavaScript 4](#_Toc50721561)

[Введение в JavaScript 4](#_Toc50721562)

[Подключение скриптов 5](#_Toc50721563)

[Структура кода 6](#_Toc50721564)

[Инструкции 6](#_Toc50721565)

[Комментарии 6](#_Toc50721566)

[Переменные 7](#_Toc50721567)

[Константы 9](#_Toc50721568)

[Типы данных 9](#_Toc50721569)

[Число 10](#_Toc50721570)

[Строка 10](#_Toc50721571)

[Логический тип 11](#_Toc50721572)

[Значение «null» 11](#_Toc50721573)

[Значение «undefined» 11](#_Toc50721574)

[Преобразование типов 12](#_Toc50721575)

[Строковое преобразование 12](#_Toc50721576)

[Численное преобразование 12](#_Toc50721577)

[Логическое преобразование 13](#_Toc50721578)

[Операторы 13](#_Toc50721579)

[Термины: «унарный», «бинарный», «операнд» 14](#_Toc50721580)

[Сложение строк, бинарный + 14](#_Toc50721581)

[Преобразование к числу, унарный плюс + 14](#_Toc50721582)

[Приоритет операторов 14](#_Toc50721583)

[Присваивание 15](#_Toc50721584)

[Остаток от деления % 15](#_Toc50721585)

[Возведение в степень \*\* 15](#_Toc50721586)

[Инкремент/декремент 16](#_Toc50721587)

[Операторы сравнения 16](#_Toc50721588)

[Сравнение строк 16](#_Toc50721589)

[Сравнение разных типов 17](#_Toc50721590)

[Строгое сравнение 17](#_Toc50721591)

[Взаимодействие: alert, prompt, confirm 17](#_Toc50721592)

[alert 17](#_Toc50721593)

[prompt 17](#_Toc50721594)

[confirm 18](#_Toc50721595)

[Условные операторы: if, '?' 18](#_Toc50721596)

[Практическая работа №4 20](#_Toc50721597)

[Логические операторы 21](#_Toc50721598)

[|| (ИЛИ) 21](#_Toc50721599)

[&& (И) 22](#_Toc50721600)

[! (НЕ) 23](#_Toc50721601)

[Циклы while и for 23](#_Toc50721602)

[Цикл «while» 23](#_Toc50721603)

[Цикл «do…while» 24](#_Toc50721604)

[Цикл «for» 24](#_Toc50721605)

[Прерывание цикла: «break» 24](#_Toc50721606)

[Переход к следующей итерации: continue 25](#_Toc50721607)

[Конструкция "switch" 25](#_Toc50721608)

[Функции 27](#_Toc50721609)

[Объявление функции 27](#_Toc50721610)

[Локальные переменные 28](#_Toc50721611)

[Внешние переменные 28](#_Toc50721612)

[Параметры 29](#_Toc50721613)

[Параметры по умолчанию 30](#_Toc50721614)

[Возврат значения 30](#_Toc50721615)

[Выбор имени функции 31](#_Toc50721616)

[Function Expression 32](#_Toc50721617)

[Функции-«колбэки» 32](#_Toc50721618)

[Функции-стрелки, основы 34](#_Toc50721619)

# 

# 1. Введение

## Для кого предназначено данное методическое пособие

Данное пособие предназначено для знакомства с клиентской веб-разработкой. Представленный материал подойдет для самостоятельного изучения и использования как основы для организации учебных практик и практических занятий в учебных заведениях.

## Необходимые начальные знания

Предполагается, что читатель знаком с основами программирования, основами HTML и CSS.

# 2. Основы JavaScript

## Введение в JavaScript

Изначально JavaScript был создан, чтобы «сделать веб-страницы живыми».

Программы на этом языке называются *скриптами*. Они могут встраиваться в HTML и выполняться автоматически при загрузке веб-страницы.

Скрипты распространяются и выполняются, как простой текст. Им не нужна специальная подготовка или компиляция для запуска.

Сегодня JavaScript может выполняться не только в браузере, но и на сервере или на любом другом устройстве, которое имеет специальную программу, называющуюся «движком» JavaScript.

У браузера есть собственный движок, который иногда называют «виртуальная машина JavaScript».

Современный JavaScript – это «безопасный» язык программирования. Он не предоставляет низкоуровневый доступ к памяти или процессору, потому что изначально был создан для браузеров, не требующих этого.

Возможности JavaScript сильно зависят от окружения, в котором он работает. Например, Node.JS поддерживает функции чтения/записи произвольных файлов, выполнения сетевых запросов и т.д.

В браузере для JavaScript доступно всё, что связано с манипулированием веб-страницами, взаимодействием с пользователем и веб-сервером.

Например, в браузере JavaScript может:

1. Добавлять новый HTML-код на страницу, изменять существующее содержимое, модифицировать стили.
2. Реагировать на действия пользователя, щелчки мыши, перемещения указателя, нажатия клавиш.
3. Отправлять сетевые запросы на удалённые сервера, скачивать и загружать файлы (технологии AJAX и COMET).
4. Получать и устанавливать куки, задавать вопросы посетителю, показывать сообщения.
5. Запоминать данные на стороне клиента («local storage»).

Возможности JavaScript в браузере ограничены ради безопасности пользователя. Цель заключается в предотвращении доступа недобросовестной веб-страницы к личной информации или нанесения ущерба данным пользователя.

Примеры таких ограничений включают в себя:

1. JavaScript на веб-странице не может читать/записывать произвольные файлы на жёстком диске, копировать их или запускать программы. Он не имеет прямого доступа к системным функциям ОС.
2. Современные браузеры позволяют ему работать с файлами, но с ограниченным доступом, и предоставляют его, только если пользователь выполняет определенные действия, такие как «перетаскивание» файла в окно браузера или его выбор с помощью тега <input>.
3. Существуют способы взаимодействия с камерой/микрофоном и другими устройствами, но они требуют явного разрешения пользователя. Таким образом, страница с поддержкой JavaScript не может незаметно включить веб-камеру, наблюдать за происходящим и отправлять информацию в ФСБ.
4. Различные окна/вкладки не знают друг о друге. Иногда одно окно, используя JavaScript, открывает другое окно. Но даже в этом случае JavaScript с одной страницы не имеет доступа к другой, если они пришли с разных сайтов (с другого домена, протокола или порта).
5. JavaScript может легко взаимодействовать с сервером, с которого пришла текущая страница. Но его способность получать данные с других сайтов/доменов ограничена. Хотя это возможно в принципе, для чего требуется явное согласие (выраженное в заголовках HTTP) с удалённой стороной. Опять же, это ограничение безопасности.

Как минимум, три сильные стороны JavaScript:

1. Полная интеграция с HTML/CSS.
2. Простые вещи делаются просто.
3. Поддерживается всеми основными браузерами и включён по умолчанию.

## Подключение скриптов

Программы на JavaScript могут быть вставлены в любое место HTML-документа с помощью тега <script> (Листинг 2.1)

**Листинг 2.1** Включение скрипта в тело страницы

<!DOCTYPE HTML>

<html>

<body>

<p>Перед скриптом...</p>

<script>

*alert*( **'Привет, мир!'** );

</script>

<p>...После скрипта.</p>

</body>

</html>

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

Файл скрипта можно подключить к HTML с помощью атрибута src (Листинг 2.2).

**Листинг 2.2** Подключение скрипта отдельным файлом

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

## Структура 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.

### 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 (Листинг 2.3).

**Листинг 2.3** Однострочный комментарий

*// 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 \*/** (Листинг 2.4).

**Листинг 2.4** Многострочный комментарий

*/\* Пример с двумя сообщениями.*

*Это - многострочный комментарий.*

*\*/*

*alert*(**'Привет'**);

*alert*(**'Мир'**);

## Переменные

*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 (Листинг 2.5).

**Листинг 2.5** Объявление переменной

**let *message***;

Теперь можно поместить в нее данные, используя оператор присваивания = (Листинг 2.6).

**Листинг 2.6** Присваивание переменной

**let *message***;

***message*** = **'Hello'**; *// сохранить строку*

Строка сохраняется в области памяти, связанной с переменной. Мы можем получить к ней доступ, используя имя переменной (Листинг 2.7).

**Листинг 2.7** Использование переменной

**let *message***;

***message*** = **'Hello!'**;

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

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 (Рисунок 2.1).



**Рисунок. 2.1.** Визуализация переменной

Мы можем положить любое значение в коробку.

Мы также можем изменить его столько раз, сколько захотим (Листинг 2.8)

**Листинг 2.8** Изменение значение переменной

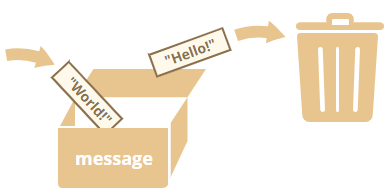
**let *message***;

***message*** = **'Hello!'**;

***message*** = **'World!'**; *// значение изменено*

*alert*(***message***);

При изменении значения старые данные удаляются из переменной (Рисунок 2.2).



**Рисунок. 2.2.** Визуализация изменения значения переменной

Мы также можем объявить две переменные и скопировать данные из одной в другую (Листинг 2.9).

**Листинг 2.9** Копирование значение одной переменной 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!*

В JavaScript есть два ограничения, касающиеся имен переменных:

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

## Константы

Чтобы объявить константную, то есть, неизменяемую переменную, используйте const вместо let (Листинг 2.10).

**Листинг 2.10** Объявление констант

**const *myBirthday*** = **'18.04.1982'**;

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

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 подчеркивания (Листинг 2.11).

**Листинг 2.11** Именование констант

**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*

## Типы данных

Переменная в JavaScript может содержать любые данные. В один момент там может быть строка, а в другой – число (Листинг 2.12).

**Листинг 2.12** Изменение типа переменной

*// 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.

### Число

*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.

NaN означает вычислительную ошибку. Это результат неправильной или неопределённой математической операции.

### Строка

Строка (string) в JavaScript должна быть заключена в кавычки (Листинг 2.13).

**Листинг 2.13** Инициализация строковой переменной

**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 «расширенный функционал». Они позволяют нам встраивать выражения в строку, заключая их в ${…}.Пример в листинге 2.14.:

**Листинг 2.14** Использование обратных кавычек

**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 такого функционала встраивания!

### 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» (Листинг 2.15).

**Листинг 2.15** Логический тип данных

**let *nameFieldChecked*** = **true**; *// да, поле отмечено*

**let *ageFieldChecked*** = **false**; *// нет, поле не отмечено*

Булевые значения также могут быть результатом сравнений (Листинг 2.16)

**Листинг 2.16** Использование логического типа

**let *isGreater*** = 4 > 1;

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

### Значение «null»

Специальное значение null не относится ни к одному из типов, описанных выше.

Оно формирует отдельный тип, который содержит только значение null (Листинг 2.17).

**Листинг 2.17** Тип данных 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».

### 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 (Листинг 2.18).

**Листинг 2.18** Тип данных undefined

**let *x***;

*alert*(***x***); *// выведет "undefined"*

## 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.

### 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 (Листинг 2.19).

**Листинг 2.19** Строковое преобразование

**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 (Листинг 2.20)

**Листинг 2.20** Численное преобразование

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

Мы можем использовать функцию Number(value), чтобы явно преобразовать value к числу (Листинг 2.21)

**Листинг 2.21** Численное преобразование

**let *str*** = **"123"**;

*alert*(**typeof *str***); *// string*

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

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

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. |

### 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:

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

*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.

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

### Сложение строк, бинарный +

Обычно при помощи плюса '+' складывают числа.

Но если бинарный оператор '+' применить к строкам, то он их объединяет в одну (Листинг 2.22)

**Листинг 2.22** Сложение строк

**let *s*** = **"моя"** + **"строка"**;

*alert*(***s***); *// моястрока*

### Преобразование к числу, унарный плюс +

Плюс + существует в двух формах: бинарной, которую мы использовали выше, и унарной.

Унарный, то есть примененный к одному значению, плюс + ничего не делает с числами. Но если операнд не число, унарный плюс преобразует его в число (Листинг 2.23).

**Листинг 2.23** Унарный +

*// 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*

### Приоритет операторов

В том случае, если в выражении есть несколько операторов – порядок их выполнения определяется *приоритетом*, или, другими словами, существует определенный порядок выполнения операторов.

Из школы мы знаем, что умножение в выражении 2 \* 2 + 1 выполняется 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.

Общую таблицу приоритетов операторов можно посмотреть здесь: <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Operator_Precedence>

### Присваивание

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

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

**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.

### 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 (Листинг 2.24).

**Листинг 2.24** Оператор %

*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.

### 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:

1. Инкремент ++ увеличивает на 1;
2. Декремент -- уменьшает на 1.

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

## Операторы сравнения

Многие операторы сравнения известны нам из математики:

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

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

1. true – означает «да», «верно», «истина».
2. false – означает «нет», «неверно», «ложь».

**Листинг 2.25** Возврат значения логического оператора

*alert*( 2 > 1 ); *// true (верно)*

*alert*( 2 == 1 ); *// false (неверно)*

*alert*( 2 != 1 ); *// true (верно)*

### Сравнение строк

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

Другими словами, строки сравниваются посимвольно.

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.

### Сравнение разных типов

При сравнении значений разных типов JavaScript приводит каждое из них к числу (Листинг 2.26).

**Листинг 2.26** Сравнение разных типов

*alert*( **'2'** > 1 ); *// true, строка '2' становится числом 2*

*alert*( **'01'** == 1 ); *// true, строка '01' становится числом 1*

### Строгое сравнение

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

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.

## Взаимодействие: alert, prompt, confirm

### alert

Синтаксис:

*alert(message);*

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

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 отменен пользователем (Листинг 2.27)

**Листинг 2.27** Использование prompt

**let *age*** = *prompt*(**'Сколько тебе лет?'**, 100);

*alert*(**`Тебе** ${***age***} **лет!`**); *// Тебе 100 лет!*

### 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 (Листинг 2.28).

**Листинг 2.28** Использование confirm

**let *isBoss*** = *confirm*(**"Ты здесь главный?"**);

*alert*( ***isBoss*** ); *// true, если нажата OK*

## 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(...) вычисляет условие в скобках и, если результат true, то выполняет блок кода (Листинг 2.29).

**Листинг 2.29** Использование оператора if

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

**if** (***year*** == 2015) *alert*( **'Вы правы!'** );

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

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

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

Оператор if может содержать необязательный блок «else» («иначе»). Выполняется, когда условие ложно (Листинг 2.30).

**Листинг 2.30** Использование блока else

**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*

}

Иногда, нужно проверить несколько вариантов условия. Для этого используется блок else if (Листинг 2.31).

**Листинг 2.31** Использование блока else if

**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!'** );

}

Иногда, нам нужно назначить переменную в зависимости от условия (Листинг 2.32).

**Листинг 2.32** Задание переменной в зависимости от условия

**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 своем роде, имеет три аргумента.

Синтаксис:

*let result = условие ? значение1 : значение2;*

Сначала вычисляется условие: если оно истинно, тогда возвращается значение1, в противном случае – значение2 (Листинг 2.33).

**Листинг 2.33** Тернарный оператор

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

### Практическая работа №4

1. Используя конструкцию if..else, напишите код, который будет спрашивать: «Какое «официальное» название JavaScript?» Если пользователь вводит «ECMAScript», то показать: «Верно!», в противном случае – отобразить: «Не знаете? ECMAScript!»;
2. Используя конструкцию if..else, напишите код, который получает число через prompt, а затем выводит в alert:
3. 1, если значение больше нуля,
4. -1, если значение меньше нуля,
5. 0, если значение равно нулю.

Предполагается, что пользователь вводит только числа.

1. Перепишите if с использованием условного оператора '?':

**let *result***;

**if** (a + b < 4) {

***result*** = **'Мало'**;

} **else** {

***result*** = **'Много'**;

}

## 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.

### || (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 (Листинг 2.34).

**Листинг 2.34** Использование логического ИЛИ

**let *hour*** = 9;

**if** (***hour*** < 10 || ***hour*** > 18) {

*alert*( **'Офис закрыт.'** );

}

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:

1. Вычисляет операнды слева направо.
2. Каждый операнд конвертирует в логическое значение. Если результат true, останавливается и возвращает исходное значение этого операнда.
3. Если все операнды являются ложными (false), возвращает последний из них.
4. Значение возвращается в исходном виде, без преобразования.

Другими словами, цепочка ИЛИ "||" возвращает первое истинное значение или последнее, если такое значение не найдено (Листинг 2.35).

**Листинг 2.35** Расширенные возможности логического ИЛИ

*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)*

### && (И)

Оператор И пишется как два амперсанда &&:

*result = a && b;*

В традиционном программировании И возвращает true, если оба аргумента истинны, а иначе – false (Листинг 2.36).

**Листинг 2.36** Традиционное использование оператора И

**let *hour*** = 12;

**let *minute*** = 30;

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

*alert*( **'The time is 12:30'** );

}

При нескольких подряд операторах И:

*result = value1 && value2 && value3;*

Оператор && выполняет следующие действия:

1. Вычисляет операнды слева направо.
2. Каждый операнд преобразует в логическое значение. Если результат false, останавливается и возвращает исходное значение этого операнда.
3. Если все операнды были истинными, возвращается последний.
4. 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* (Листинг 2.37).

**Листинг 2.37** Расширенное использование оператора И

*// 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*

### ! (НЕ)

Оператор НЕ представлен восклицательным знаком !.

Синтаксис довольно прост:

*result = !value;*

Оператор принимает один аргумент и выполняет следующие действия:

Сначала приводит аргумент к логическому типу true/false.

Затем возвращает противоположное значение.

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 (Листинг 2.38)

**Листинг 2.38** Пример использования цикла while

**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.

### 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.

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 ...*

*}*

Цикл ниже выполняет alert(i) для i от 0 до (но не включая) 3 (Листинг 2.39).

**Листинг 2.39** Пример использования цикла for

**for** (**let *i*** = 0; ***i*** < 3; ***i***++) { *// выведет 0, затем 1, затем 2*

*alert*(***i***);

}

То есть, *начало* выполняется один раз, а затем каждая итерация заключается в проверке условия, после которой выполняется *тело* и *шаг*.

### 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 – выдает результат (Листинг 2.40).

**Листинг 2.40** Пример использования команды break

**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).

Ее используют, если понятно, что на текущем повторе цикла делать больше нечего (Листинг 2.41).

**Листинг 2.41** Пример использования команды continue

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

## Конструкция "switch"

Конструкция switch заменяет собой сразу несколько if.

Она представляет собой более наглядный способ сравнить выражение сразу с несколькими вариантами.

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]*

*}*

1. Переменная x проверяется на строгое равенство первому значению value1, затем второму value2 и так далее.
2. Если соответствие установлено – switch начинает выполняться от соответствующей директивы case и далее, до ближайшего break (или до конца switch).
3. Если ни один case не совпал – выполняется (если есть) вариант default.

Пример использования можно увидеть в Листинге 2.42.

**Листинг 2.42** Пример использования конструкции switch

**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 прервет выполнение.

**Если break нет, то выполнение пойдёт ниже по следующим case, при этом остальные проверки игнорируются.**

## 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* (Листинг 2.43).

**Листинг 2.43** 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 вышеприведенном 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() (Листинг 2.44).

**Листинг 2.44** Пример вызова функции

**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 (Листинг 2.45).

**Листинг 2.45** Локальные переменные функции

***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, Листинг 2.46.

**Листинг 2.46** Использование внешних переменных внутри функции

**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.

Внешняя переменная используется, только если внутри функции нет такой локальной.

Если одноименная переменная объявляется внутри функции, тогда она перекрывает внешнюю. Например, в коде Листинга 2.47 функция использует локальную переменную userName. Внешняя будет проигнорирована

**Листинг 2.47** Игнорирование внешней переменной

**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*

Переменные, объявленные снаружи всех функций, такие как внешняя переменная userName в вышеприведенном коде – называются *глобальными*.

*Глобальные переменные* видимы для любой функции (если только их не перекрывают одноименные 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 нижеприведенном примере функции передаются два параметра: from и text (Листинг 2.48).

**Листинг 2.48** Параметры функции

**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 передаем ее функции. Обратите внимание: функция изменяет значение from, но это изменение не видно снаружи. Функция всегда получает только копию значения (Листинг 2.49):

**Листинг 2.49** Параметры функции - локальные переменные

**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.

Если мы хотим задать параметру значение по умолчанию, мы должны указать его после = (Листинг 2.50).

**Листинг 2.50** Параметры функции - значения по умолчанию

**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.

### Возврат значения

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

Простейшим примером может служить функция сложения двух чисел (Листинг 2.51).

**Листинг 2.51** Возврат значения функции

**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 ее код (присваивается переменной result выше).

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

### 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…

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

Примеры таких имен:

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.

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!

### 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 еще один синтаксис создания функций, который называется *Function Expression* (Функциональное Выражение).

Оно выглядит вот так:

**let** *sayHi* = **function**() {

*alert*( **"Привет"** );

};

В коде выше функция создается и явно присваивается переменной, как любое другое значение. По сути без разницы, как мы определили функцию, это просто значение, хранимое в переменной sayHi.

Мы можем скопировать функцию в другую переменную (Листинг 2.52).

**Листинг 2.52** Копирование функции

**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)*

### Функции-«колбэки»

Рассмотрим еще 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() (Листинг 2.53).

**Листинг 2.53** Функции как параметры другой функции

**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.

**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 (Листинг 2.54).

**Листинг 2.54** Описание колбэков в 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.

### 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 (Листинг 2.55).

**Листинг 2.55** Пример стрелочной функции

**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 (Листинг 2.56).

**Листинг 2.56** Пример стрелочной функции без скобок

*// 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 (Листинг 2.57).

**Листинг 2.57** Пример стрелочной функции без входных параметров

**let** *sayHi* = () => *alert*(**"Hello!"**);

*sayHi*();

Поначалу функции-стрелки могут показаться необычными и трудно читаемыми, но это быстро пройдет, как только глаза привыкнут к этим конструкциям.

Они очень удобны для простых однострочных действий, когда лень писать много букв.

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 (Листинг 2.58).

**Листинг 2.58** Пример многострочной стрелочной функции

**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*