

# Модуль 1 «Основы компьютерной геометрии»

## Лекция 3 «Введение в JavaScript (отличия от C/C++)»

к.ф.-м.н., доц. каф. ФН-11, Захаров Андрей Алексеевич,  
ауд.:930а(УЛК)  
моб.: 8-910-461-70-04,  
email: azaharov@bmstu.ru

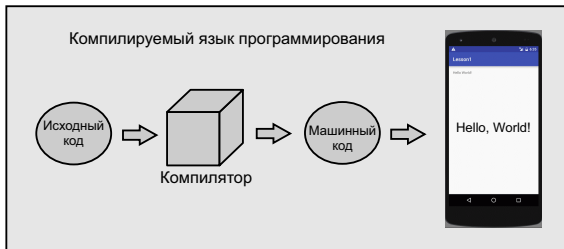


МГТУ им. Н.Э. Баумана

7 октября 2024 г.

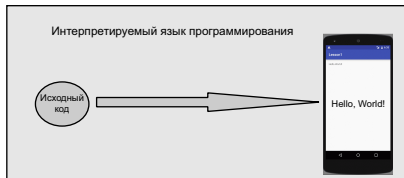
*Компьютерные программы* — это наборы инструкций, с помощью которых можно заставить компьютер выполнять определённые действия. Каждый компьютерный язык программирования располагает собственным набором инструкций и правилами их записи, которыми должны руководствоваться программисты. Компьютер не способен понимать эти инструкции в непосредственном виде. Чтобы компьютер мог понимать язык программирования, требуется предварительный процесс преобразования инструкций, в ходе которого они транслируются (переводятся) из формы, удобной для чтения (и записи) человеком, в машинный язык. В зависимости от того, когда именно осуществляется процесс трансляции инструкций, языки программирования делятся на два основных типа: компилируемые и интерпретируемые.

# Компилируемые языки программирования



*Компилируемые языки программирования* — это языки, которые требуют, чтобы написанный программистом код был обработан специальной программой, так называемым *компилятором*, который анализирует данный код, а затем преобразует его в машинный язык. После этого компьютер может выполнить скомпилированную программу. В качестве примера компилируемых языков программирования можно привести C, C++, Fortran, Java, Objective-C и COBOL.

# Интерпретируемые языки программирования

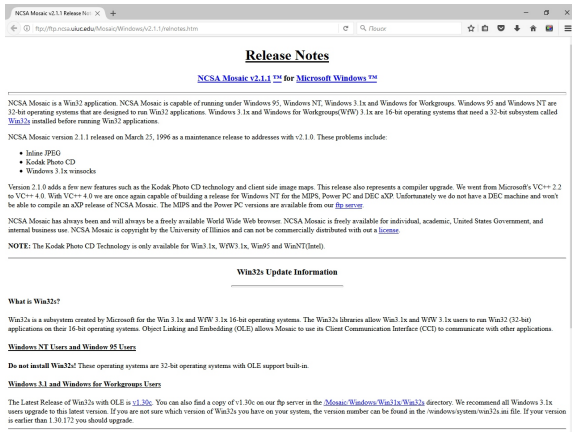


*Интерпретируемые языки программирования* — это языки, которые также компилируются компьютером в машинный язык, но процесс компиляции осуществляется в браузере пользователя непосредственно во время выполнения программы.

Плюсы: изменения могут быть внесены в программу в любой момент.

Минусы: компиляция кода во время его выполнения может замедлять выполнение программ. Отчасти именно из-за такого снижения производительности интерпретируемые языки получили репутацию не очень серьезных языков программирования. Однако с появлением улучшенных компиляторов, осуществляющих компиляцию кода на стадии его выполнения (так называемые *JIT-компиляторы* («just-in-time») или *оперативные компиляторы*), и процессоров с повышенным быстродействием такая точка зрения очень быстро устарела.

Примерами интерпретируемых языков программирования могут служить PHP, Perl, Haskell, Ruby, JavaScript.



Ещё совсем недавно Всемирная паутина казалась очень скучным местом. Составленные из одного лишь HTML-текста веб-страницы отображали информацию и ничего больше. Можно было лишь перейти по ссылке и ждать загрузки новой веб-страницы — вот что считалось настоящей интерактивностью.

Проект JavaScript стартовал в компании Netscape в 1995 году. Брендан Эйх или Айк (Brendan Eich), создатель языка JavaScript, заимствовал все лучшее из ряда других языков программирования. Потребовалось 15 лет, чтобы этот язык завоевал популярность у ведущих разработчиков. Язык JavaScript первоначально назывался Mocha. С выпуском первой бета-версии браузера Netscape Navigator он был переименован в LiveScript, но в версию браузера Netscape 2, выпущенную в 1995 г., он был встроен уже как язык JavaScript.

Название JavaScript, которое было дано языку компанией Netscape и используется большинством тех, кто с ним работает, тоже нельзя признать удачным. Несмотря на определённое сходство, в действительности Java и JavaScript — это два совершенно разных языка.

Компания Netscape представила язык JavaScript в организацию Ecma International, занимающуюся стандартизацией информационных и коммуникационных технологий, где он был утвержден в качестве стандарта ECMAScript в 1997 г. Брендан Эйх высказал по поводу названия стандартизированного языка нашумевшее замечание, в котором охарактеризовал ECMAScript как «неудачное название торговой марки, звучащее подобно названию кожного заболевания».

Версия ECMAScript 5.1 (ES5) была опубликована в июне 2011 года, версия ECMAScript 6 (ES6) опубликована в июне 2015 года (ECMAScript 2015), версия ECMAScript 7 — ECMAScript 2016.

Сразу же после своего дебюта язык JavaScript быстро приобрёл популярность в качестве средства, позволяющего делать веб-страницы более динамичными. Одним из первых результатов встраивания JavaScript в браузеры стало появление так называемого динамического HTML (DHTML) — языка разметки веб-страниц, обеспечившего реализацию не только всевозможных визуальных эффектов, но и более полезных вещей, например раскрывающихся меню и средств валидации (проверки корректности данных) веб-страниц.

Сейчас JavaScript стал наиболее широко используемым языком программирования в мире, и практически на каждом из эксплуатируемых в настоящее время персональных компьютеров установлен по крайней мере один браузер, способный выполнять JavaScript-код.

Язык JavaScript достаточно гибок для того, чтобы быть пригодным для использования непрограммистами, и одновременно располагает достаточно мощными возможностями, которых профессиональным программистам вполне хватает для того, чтобы поддерживать функциональность практически любого современного сайта в Интернете, начиная с одностраничных сайтов и кончая такими «монстрами», как сайты Google, Amazon, Facebook и многие другие.

JavaScript часто описывают как *динамический сценарный язык*, т.е. язык для написания сценариев (жарг. *скриптовый язык*).

Чаще всего программы на JavaScript выполняются в браузерах или Node.js. Существуют три способа выполнения JavaScript в браузере:

- ▶ поместить код непосредственно в атрибут события HTML-элемента:

```
<button id="bigButton" onclick="alert('Привет,  
мир!');">Щелкните здесь</button>
```

В данном случае, когда пользователь щелкает на кнопке, созданной этим HTML-элементом, на экране появляется всплывающее окно, в котором отображается текст "Привет, мир!";

- ▶ поместить код между открывающим и закрывающим тегами script:

```
<script type="text/javascript">
```

Сюда вставляется код JavaScript

```
</script>
```

- ▶ поместить код в отдельный документ, который включается в HTML:

```
<script type="text/javascript" src="myScript.js"></script>
```

Преимущества использования внешних файлов JavaScript:

- ▶ улучшается удобочитаемость HTML-файлов;
- ▶ упрощается сопровождение кода, поскольку вносить изменения или исправлять ошибки приходится только в одном месте.



Прикрепляя внешний файл JavaScript к веб-странице, следует указать URL-адрес для атрибута `src` элемента `script`. URL или *унифицированный указатель ресурса*, — это путь к файлу, расположенному во Всемирной паутине, то есть его адрес. Есть три вида адресов: *абсолютный*, *относительно корневого каталога* и *относительно документа*. Все они указывают браузеру, где он может найти конкретный файл (например, другую веб-страницу, изображение или файл JavaScript).

*Абсолютный адрес* содержит всю информацию, необходимую браузеру, чтобы найти файл, где бы он ни находился. Абсолютный путь содержит `http://`, имя сервера, имя каталога и имя файла. Например, `http://www.autospeedguides.com/scripts/site.js`. Этот тип используется при указании файла, находящегося на сервере, отличном от того, на котором расположена веб-страница.

Путь относительно корневого каталога указывает, где находится файл по отношению к корневому каталогу — наивысшему каталогу в иерархии сайта. Путь относительно корневого каталога не имеет `http://` или доменного имени. Он начинается с символа `/` (слеша), обозначающего корневой каталог сайта — каталог, в котором находится домашняя страница. Например, адрес `/scripts/site.js` сообщает, что файл `site.js` расположен в каталоге с именем `scripts`, который находится в каталоге высшего уровня. Простой способ создать путь относительно корневого каталога — это взять абсолютный адрес и отбросить `http://` и имя сервера. Например, путь

`http://www.autospeedguides.com/index.html`, записанный как ссылка относительно корневого каталога, — это `/index.html`.

Этот тип подходит для файлов JavaScript, хранящихся на том же сайте, что и использующие его html-файлы. Поскольку такие адреса всегда начинаются с указания корневого каталога, URL-адрес файла JavaScript, будет одинаковым для всех страниц сайта, даже если они расположены в разных каталогах. Этот способ работает либо при просмотре веб-страницы на сервере в Интернете, либо через веб-сервер, настроенный на собственном компьютере. Он не работает, если просто открыть веб-страницу, используя команду меню Файл ⇒ Открыть.

*Путь относительно документа* показывает путь от веб-страницы к файлу JavaScript. Если на сайте имеется много уровней и каталогов, то требуется использовать различные пути, чтобы указать на один и тот же файл JavaScript. Например, пусть файл JavaScript с именем *site.js* размещен в каталоге *scripts* в корневом каталоге сайта. Тогда путь относительно документа для этого файла будет одним для домашней страницы: `script/site.js`, но другим для страницы, находящейся внутри каталога `about`: `../scripts/site.js`. Символы `../` означают подъем на уровень выше каталога `about`, тогда как `/scripts/site.js` означает переход к каталогу `scripts` и к файлу `site.js`. Этот тип является наилучшим выбором для тестирования на собственном компьютере без обращения к серверу. Пути остаются рабочими и после перемещения на сервер, но их придется менять в случае изменения местоположения веб-страницы.

# Атрибуты событий элементов HTML

В HTML предусмотрено несколько (около 70) специальных атрибутов, предназначенных для запуска JavaScript при наступлении определённых событий в браузере или при выполнении пользователем определённых действий. Наиболее часто используемые из них приведены в табл.:

Атрибут	Условие запуска сценария
onload	Окончание загрузки страницы
onfocus	Получение элементом фокуса ввода
onblur	Потеря элементом фокуса ввода (например, при активизации текстового поля)
onchange	Изменение значения элемента
onselect	Выделение текста
onsubmit	Отправка формы
onkeydown	Нажатие клавиши
onkeypress	Нажатие и последующее отпускание клавиши
onkeyup	Отпускание клавиши
onclick	Щелчок мышью на элементе
ondrag	Перетаскивание элемента
ondrop	Вставка перетаскиваемого элемента
onmouseover	Перемещение указателя мыши над элементом

# Зарезервированные слова

Некоторые слова нельзя использовать в качестве имен переменных. Ниже приведен перечень зарезервированных слов, которые не могут выступать в качестве названий переменных, функций, методов, меток циклов и объектов.

abstract	else	instanceof	static	with
boolean	enum	int	super	yield
break	export	interface	switch	
byte	extends	let	synchronized	
case	false	long	this	
catch	final	native	throw	
char	finally	new	throws	
class	float	null	transient	
const	for	package	true	
continue	function	private	try	
debugger	goto	protected	typeof	
default	if	prototype	var	
delete	implements	public	void	
do	import	return	volatile	
double	in	short	while	

# Объявление переменных

Для объявления переменных используется ключевое слово `let`:

```
let a = 12; // доступна глобально
function myFunction() {
  console.log(a);
  let b = 13; // доступна в пределах функции
  if (true) {
    let c = 14; // доступна только в инструкции if
    console.log(b);
  }
  console.log(c);
}
myFunction();
```

Результаты выполнения примера:

12

13

Reference Error Exception

Переменные имеют область видимости в пределах блока (а также во всех вложенных в него блоках). Если переменные объявлены вне функции, то они являются глобальными.

Константа содержит значение, но в отличие от переменной оно не может быть изменено после инициализации. Константы объявляются с использованием ключевого слова `const`:

```
// допустимые константы  
const ROOM_TEMP_C = 21.5, MAX_TEMP_C = 30;
```

```
// синтаксическая ошибка: отсутствует значение для инициализации  
const NAME;
```

Константы являются постоянными переменными с областью видимости в пределах блока, то есть, к ним применяются те же правила, что и к переменным, объявленным с помощью ключевого слова `let`.



JavaScript относится к категории так называемых слабо типизированных языков. Это означает, что не требуется сообщать JavaScript или даже знать самому, будет ли создаваемая переменная содержать текст, число или какой-либо другой тип данных. JavaScript автоматически выполняет всю работу по определению типов данных, хранящихся в переменных. JavaScript распознает семь основных типов данных. Для сравнения, в языке программирования C++ насчитывается по крайней мере 12 различных типов данных.

Перечислим основные типы данных языка JavaScript:

- ▶ `undefined` — если значение не определено. При создании переменной JavaScript, если не присвоить ей какое-либо значение, то все равно она всегда будет иметь значение по умолчанию `"undefined"`.
- ▶ `number` — целые числа или числа с плавающей точкой;
- ▶ `string` — строки;
- ▶ `boolean` — логический тип данных. Может содержать значения `true` (истина) или `false` (ложь);
- ▶ `function` — ссылка на функцию;
- ▶ `symbol` — символы;
- ▶ `object` — массивы, объекты, а также переменная со значением `null` (ссылка на пустой объект). Данный тип не относится к примитивным (элементарным) типам данных.

В JavaScript числа хранятся в виде 64-разрядных значений с плавающей точкой в соответствии с международным стандартом IEEE 754 (числа с плавающей точкой двойной точности). Этот формат использует для хранения числа 64 бита, где число (дробная часть) занимает биты с 0 по 51, экспонента — биты с 52 по 62, а знак числа — последний бит. В переводе на обычный язык это означает, что их значения могут меняться в пределах от  $\pm 5 \times 10^{-324}$  до  $\pm 1.7976931348623157 \times 10^{308}$ . Любое число может быть записано как с десятичной точкой, так и без неё. В отличие от большинства других языков программирования, в JavaScript отсутствуют отдельные типы данных для целых чисел (положительных и отрицательных чисел без дробной части) и чисел с плавающей точкой.

JavaScript распознает четыре типа числовых литералов: десятичный, двоичный, восьмеричный и шестнадцатеричный. Кроме того, есть специальные значения для бесконечности, отрицательной бесконечности и «не числа»:

```
let count = 10;           // целочисленный литерал двойной точности
const blue = 0x0000ff;    // шестнадцатеричное
const umask = 0o0022;     // восьмеричное
const a = 0b00001111;     // двоичное
const roomTemp = 21.5;    // десятичное число
const c = 3.0e6;          // экспоненциальное
const e = -1.6E-19;       // экспоненциальное
const inf = Infinity;
const ninf = -Infinity;
const nan = NaN;
```

NaN — это сокращение от Not a Number (не число). Этот результат возникает при попытке выполнить математические операции над строками или в тех случаях, когда вычисление приводит к ошибке или не может быть выполнено. Например, невозможно вычислить квадратный корень отрицательного числа. Результатом такой попытки будет NaN.

## Строковый тип данных

Строковая переменная создаётся путём заключения значения в одинарных кавычках, парных кавычках или обратных апострофах:

```
const myString = "Привет, я строка.";
```

Открывающая и закрывающая кавычки должны быть одного типа. Для строк поддерживается оператор конкатенации строк +:

```
const Str = "Строка1" + "Строка2";  
// Переменная Str будет содержать значение "Строка1Строка2"
```

Часто необходимо сформировать строку, состоящую из имени переменной и её значения. Если написать

```
const X = "Строка1";  
const Z = "Значение равно " + X;
```

то переменная Z будет содержать значение "Значение равно Строка1". Для быстрого ввода значения в строку можно применять строковые шаблоны. В строковых шаблонах используются обратные апострофы вместо одинарных или двойных кавычек и знак доллара с фигурными скобками, внутрь которых помещается нужное выражение:

```
let currentTemp = 19.5;  
const message = `The current value is ${currentTemp}`;
```

В любом элементе массива могут храниться данные любого типа, в том числе и другие массивы. Кроме того, элементы массивов могут содержать функции и объекты JavaScript.

Имея возможность хранить в массиве данные любого типа, также возможно хранить разные типы данных в разных элементах одного и того же массива:

```
item[0] = "apple";  
item[1] = 4+8;  
item[2] = 3;  
item[3] = item[2] * item[1];
```

Для создания массива JavaScript используется следующая конструкция:

```
const dogNames = ["Shaggy", "Tennessee", "Dr. Spock"];
```

Создание пустого массива с последующим добавлением в него элементов осуществляется следующим образом:

```
const peopleList = [];
```

```
peopleList[0] = "Chris Minnick";  
peopleList[1] = "Eva Holland";  
peopleList[2] = "Abraham Lincoln";
```

Не обязательно строго соблюдать последовательность заполнения массива элементами. В этом примере вполне допустимой была бы следующая инструкция:

```
peopleList[99] = "Tina Turner";
```

Такой способ добавления элемента в массив дополнительно сопровождается созданием пустых элементов для всех индексов между `peopleList[2]` и `peopleList[99]`.

Оператор	Описание	Пример
<code>==</code>	Равно	<code>3 == "3" // true</code>
<code>!=</code>	Не равно	<code>3 != "3" // false</code>
<code>===</code>	Строго равно	<code>3 === "3" // false</code>
<code>!==</code>	Строго не равно	<code>3 !== "3" // true</code>
<code>&gt;</code>	Больше	<code>7 &gt; 1 //true</code>
<code>&gt;=</code>	Больше или равно	<code>7 &gt;= 7 // true</code>
<code>&lt;</code>	Меньше	<code>7 &lt; 10 //true</code>
<code>&lt;=</code>	Меньше или равно	<code>2 &lt;= 2 // true</code>

В чем отличие оператора `==` (равно) от оператора `===` (строго равно)? Дело все в том, что если используется оператор `==`, интерпретатор пытается преобразовать разные типы данных к одному и лишь затем сравнивает их. Оператор `===`, встретив данные разных типов, сразу возвращает `false` (ложь).

# Написание функций

Функция — это фрагмент кода JavaScript, который можно вызывать из любого места программы. Функция описывается с помощью ключевого слова `function` по следующей схеме:

```
function <Имя функции> ([<Параметры>]) {  
    <Тело функции>  
    [return <Значение>]  
}
```

или

```
const myFunction = new Function([<Параметры>]) {  
    <Тело функции>  
    [return <Значение>]  
};
```

В определении функции разрешается задавать до 255 параметров. Функция объявляется и определяется в программе или на веб-странице только один раз. Если же вы определите одну и ту же функцию более одного раза, то JavaScript не сообщит об ошибке. В подобных случаях используется та версия функции, которая была определена последней.



Количество аргументов, задаваемых при вызове функции, не обязательно совпадать с количеством параметров, указанных в определении этой функции. Если в определении функции содержатся три параметра, но вы вызываете её, задавая всего лишь два аргумента, то третий параметр будет иметь значение `undefined`.

Если желательно, чтобы значения аргументов по умолчанию были отличными от `undefined`, то установите эти значения, например, следующим образом:

```
function welcome(yourName = "друг") {  
    console.log("Привет," + yourName);  
}  
welcome(); // Выведет "Привет,друг"
```

## Функции с произвольным количеством аргументов

Для написания функции с произвольным количеством аргументов можно использовать оператор расширения (...)

```
function addPrefix(prefix, ...words) {  
    let prefixedWords = [];  
    for(let i=0; i<words.length; i++) {  
        prefixedWords[i] = prefix + words[i];  
    }  
    return prefixedWords;  
}
```

```
addPrefix("con", "verse", "vex"); // ["converse", "convex"]
```

Если в объявлении функции используется оператор расширения, то он *должен быть у последнего аргумента*.

# Написание функций

Ссылку на функцию можно сохранить в какой-либо переменной. Для этого название функции указывается без круглых скобок:

```
function f_Sum(x, y) {  
    return x + y;  
}  
let s;  
s = f_Sum; // Присваиваем ссылку на функцию  
s(5,6); // Вызываем функцию f_Sum() через переменную s
```

В заголовке функции имя не является обязательной частью, поэтому можно создавать функции, не имеющие имени (анонимные функции):

```
let s = function(x, y) { // Присваиваем ссылку на анонимную функцию  
    return x + y;  
};  
s(5,6); // Вызываем анонимную функцию через переменную s
```

Анонимные функции, присвоенные переменной, существуют и могут вызываться лишь после выполнения операции присваивания. Доступ же к именованным функциям возможен в любом месте программы.

Значение переменной, которой присвоено значение анонимной функции может быть в любой момент изменено, и ей может быть присвоена другая функция.

Считается хорошей практикой всегда документировать JavaScript-код с помощью какой-либо стандартной системы. Наиболее широко используемой системой документирования JavaScript, которая стала стандартом де-факто, является система JSDoc.

Язык JSDoc — это простой язык разметки, который может внедряться в файлы JavaScript. Текущая, 3-я версия JSDoc основана на системе JavaDoc, применяющейся для документирования кода, написанного на языке Java.

Аннотируя свои JavaScript-файлы с помощью JSDoc, можно использовать генератор документации, например jsdoc-toolkit, для создания HTML-файлов, документирующих код.

Разметка JSDoc окружается специальными тегами блочных (многострочных) комментариев. Единственное различие между разметкой JSDoc и обычными блочными комментариями JavaScript состоит в том, что в первом случае разметка начинается символами `/**` и заканчивается символами `*/`, тогда как в последнем случае вслед за начальной косой чертой указывается лишь один символ звездочка. Дополнительная звездочка не препятствует использованию разметки JSDoc в качестве обычного блочного комментария, но при необходимости делает её частью генерируемой документации.

# Документирование функций с помощью JSDoc

Различные части и разделы программы документируются с использованием тегов JSDoc. Вот список наиболее популярных из них.

Тег JSDoc	Описание
@author	Имя разработчика
@constructor	Маркирует функцию как конструктор
@deprecated	Маркирует метод устаревшим и не рекомендуемым
@exception	Описывает исключение, генерируемое методом; синоним тега @throws
@exports	Указывает член, экспортируемый модулем
@param	Описывает параметр метода
@private	Обозначает, что метод закрытый
@return	Описывает возвращаемое значение; синоним @returns
@returns	Описывает возвращаемое значение; синоним @return
@see	Описывает связь с другим объектом
@this	Задаёт тип объекта, на который указывает ключевое слово this в функции
@throws	Описывает исключение, генерируемое методом
@version	Версия библиотеки

# Документирование функций с помощью JSDoc

Пример документирования функции с помощью JSDoc:

```
/**
 *Сложение элементов массива
 *@param {Array.<number>} numbersToAdd
 *@return {Number} sum
 */
function addNumbers(numbersToAdd)
{
    let sum = 0;
    for (let i = 0; i<numbersToAdd.length; i++)
        sum += numbersToAdd[i];

    return sum;
}
```

Более подробно о системе JSDoc можно прочитать на сайте <http://usejsdoc.org>.

# Стрелочные функции

Выражения стрелочных функций имеют более короткий синтаксис по сравнению с функциональными выражениями. Выражение стрелочных функций не позволяют задавать имя, поэтому стрелочные функции анонимны, если их ни к чему не присвоить:

```
(param1, param2, ..., paramN) => { statements }  
(param1, param2, ..., paramN) => expression  
// эквивалентно: (param1, param2, ..., paramN) => { return expression;  
  
// Круглые скобки не обязательны для единственного параметра:  
(singleParam) => { statements }  
singleParam => { statements }  
  
// Функция без параметров нуждается в круглых скобках:  
() => { statements }  
() => expression  
// Эквивалентно: () => { return expression; }  
  
// Остаточные параметры и параметры по умолчанию поддерживаются  
(param1, param2, ...rest) => { statements }  
(param1 = defaultValue1, ..., paramN = defaultValueN) => { statements }
```

Язык JavaScript предоставляет возможность создания собственных классов и объектов. Все типы данных, кроме объектов, называются элементарными типами данных.

Термин класс описывает обобщенную сущность (например, автомобиль), а экземпляр (или объект) — определённую сущность (какой-то конкретный автомобиль). Класс включает в себя переменные и функции для управления этими переменными. Переменные называют *свойствами*, а функции — *методами*.

Когда создаётся объект класса, вызывается его конструктор и происходит инициализация его свойств.



Создадим новый класс по имени Car

```
class Car{  
    constructor() {  
    }  
}
```

Чтобы создать конкретный автомобиль, мы используем ключевое слово `new`

```
const car1 = new Car();  
const car2 = new Car();
```

Важно понимать, что переменные хранят ссылки на объекты (то есть, адреса в памяти), а не сами объекты.

## Создание класса и объекта

Давайте сделаем класс Car немного поинтереснее. Придадим ему некие данные (марка, модель) и некие функции (переключение передач)

```
class Car{
  constructor(make, model) {
    this.make = make;
    this.model = model;
    this.userGears = ['P', 'N', 'R', 'D'];
    this.userGear = this.userGears[0];
  }
  shift(gear) {
    if(this.userGears.indexOf(gear) < 0)
      throw new Error('Ошибочная передача: ${gear}');
    this.userGear = gear;
  }
}
```

Ключевое слово `this` используется для обращения к экземпляру, метод которого был вызван.

```
const car1 = new Car("Tesla", "Model S");
const car2 = new Car("Mazda", "3i");
car1.shift('D'); // car1.userGear = "D"
car2.shift('R'); // car1.userGear = "R"
```

Статические методы класса не относятся ни к какому конкретному объекту. В статическом методе переменная `this` привязана к самому классу и в этом случае вместо неё рекомендуется использовать имя класса. Статические методы используются для выполнения обобщённых задач, которые связаны с классом, а не с любым конкретным объектом.

```
class Car {
    static getNextVin() {
        return Car.nextVin++; } // можно и this.nextVin++
    constructor(make, model) {
        this.make = make;
        this.model = model;
        this.vin = Car.getNextVin(); }
    static areSimilar(car1, car2) {
        return car1.make===car2.make && car1.model===car2.model; } }
    Car.nextVin = 0;

    const car1 = new Car("Tesla", "S");
    const car2 = new Car("Mazda", "3");
    car1.vin; // 0
    car2.vin; // 1
    Car.areSimilar(car1, car2); // false
```

```
class Vehicle {
  constructor() {
    this.passengers = []; }
  addPassenger(p) {
    this.passengers.push(p); } }

class Car extends Vehicle {
  constructor() {
    super(); // эта специальная функция вызывает конструктор родителя}
  deployAirbags() {
    console.log("БАБАХ!!!"); } }

const v = new Vehicle();
v.addPassenger("Frank"); v.addPassenger("Judy");
v.passengers; // ["Frank", "Judy"]
const c = new Car();
c.addPassenger("Alice"); c.addPassenger("Cameron");
c.passengers; // ["Alice", "Cameron"]
v.deployAirbags(); // ошибка
c.deployAirbags(); // "БАБАХ!!!"
```

## Создание объекта без определения класса

В JavaScript возможно сразу создавать объекты без определения класса с помощью объектного литерала:

```
const person = {eyes: 2, feet: 2, hands: 2, eyeColor: "blue"};
```

Если в момент создания объекта ещё не известно, какие свойства он будет иметь, или впоследствии понадобится, чтобы он имел дополнительные свойства, то можно создать объект с каким-то начальным набором свойств или вообще без таковых, а нужные свойства добавить позднее:

```
const car = {};  
car.model = "BA3-2109";  
car.year = 2007;
```

Для доступа к свойствам можно также использовать *скобочную нотацию*:

```
console.log(car["model"]); // BA3-2109  
console.log(car["year"]); // 2007
```

Скобочная нотация предлагает возможности, которые точечная нотация не в состоянии обеспечить. Что наиболее важно, в квадратных скобках можно использовать переменные в тех случаях, когда при написании программы имя свойства не известно заранее:

```
const carProperty = "model";  
car[carProperty] = "BA3-2109";
```

Элементарные типы, такие как числа, строки и булевы значения, а также массивы и функции JavaScript также могут использоваться как объекты. Т.е. все они имеют свойства, и этим свойствам можно присваивать значения, как и в случае любого другого объекта.

## Класс Number. Работа с числами

Класс `Number` используется для хранения числовых величин, а также для доступа к константам. Экземпляр класса создаётся по следующей схеме:

```
<Экземпляр класса> = new Number (<Начальное значение>);
```

Свойства класса `Number` можно использовать без создания экземпляра класса:

- ▶ `MAX_VALUE` — максимально допустимое в JavaScript число:  
`let x = Number.MAX_VALUE; // 1.7976931348623157e+308`
- ▶ `MIN_VALUE` — минимально допустимое в JavaScript число:  
`let x = Number.MIN_VALUE; // 5e-324`
- ▶ `NaN` — значение NaN:  
`let x = Number.NaN; // NaN`
- ▶ `NEGATIVE_INFINITY` — значение «минус бесконечность»:  
`let x = Number.NEGATIVE_INFINITY; // то же, что и -Infinity`
- ▶ `POSITIVE_INFINITY` — значение «плюс бесконечность»:  
`let x = Number.POSITIVE_INFINITY; // то же, что и Infinity`

JavaScript использует такое двоичное представление чисел с плавающей точкой, что некоторые числа, такие как 0.1, 0.2, 0.3 и другие, имеют погрешность. Они округляются до ближайшего числа в этом формате, что приводит к появлению небольшой ошибки. Рассмотрим пример:

```
console.log(0.1 + 0.2 == 0.3);  
console.log(0.9 - 0.8 == 0.1);  
console.log(0.1 + 0.2);  
console.log(0.9 - 0.8);
```

Результат выполнения кода этого примера:

```
false  
false  
0.30000000000000004  
0.09999999999999998
```



У класса `Number` есть свойство `Number.EPSILON`. Это наименьшее значение, которое может быть добавлено к 1, чтобы получить отличное от него число. `Number.EPSILON` приблизительно равно  $2.2^{-16}$ . Используя его, можно создать свою функцию сравнения чисел с плавающей запятой, игнорирующую минимальные ошибки округления. Например:

```
function epsilonEqual(a, b)
{
    return Math.abs(a - b) < Number.EPSILON;
}
console.log(epsilonEqual(0.1 + 0.2, 0.3));
console.log(epsilonEqual(0.9 - 0.8, 0.1));
```

Результат выполнения этого примера:

```
true
true
```

# Класс Array

Класс Array позволяет создавать массивы как объекты и предоставляет доступ к множеству методов для обработки массивов. Экземпляр класса можно создать следующими способами:

```
<Экземпляр класса> = new Array (<Количество элементов массива>);  
<Экземпляр класса> = new Array (<Элементы массива через запятую>);
```

Если в круглых скобках указано одно число, то это число задаёт количество элементов массива. Если указано несколько элементов через запятую или единственное значение не является числом, то указанные значения записываются в создаваемый массив:

```
const catNames = new Array("Larry", "Fuzzball", "Mr. Furly");
```

Метод Array.of() используется в качестве альтернативы конструктору объекта Array для создания массивов. Создадим массив с одним элементом, содержащим это число:

```
const arr = new Array.of(2);  
console.log(arr[0]); // Результат выполнения: 2
```

Метод Array.of() должен использоваться вместо конструктора объекта Array, когда динамически создаётся новый экземпляр массива, то есть, когда тип его значений и количество элементов неизвестны.

О массивах можно получать некоторую информацию, используя свойства массивов. Чаще всего используется свойство `length`. Это свойство позволяет узнать количество элементов в массиве, включая и те, которым значения ещё не присвоены.

```
const myArray = [];  
myArray[2000] = 10;  
myArray.length; // возвращает 2001
```

## Методы для работы с массивами

`push(<Список элементов>)` — добавляет в массив элементы, указанные в списке элементов. Элементы добавляются в конец массива. Метод возвращает новую длину массива:

```
const Mass = [ "Один", "Два", "Три" ] ;  
console.log(Mass.push ("Четвертый", "Пятый")); // 5  
console.log(Mass); // ["Один", "Два", "Три", "Четвертый", "Пятый"]
```

`unshift(<Список элементов>)` — добавляет в массив элементы, указанные в списке элементов. Элементы добавляются в начало массива:

```
const Mass = [ "Один", "Два", "Три" ];  
Mass.unshift("Четвертый", "Пятый");  
console.log(Mass); // ["Четвертый", "Пятый", "Один", "Два", "Три"]
```

`concat(<Список элементов>)` — возвращает массив, полученный в результате объединения текущего массива и списка элементов. При этом в текущий массив элементы из списка не добавляются:

```
const Mass = [ "Один", "Два", "Три" ] ;  
let Mass2 = []; // Пустой массив  
Mass2 = Mass.concat("Четвертый", "Пятый");  
console.log(Mass.join (", ")); // ["Один", "Два", "Три"]  
console.log(Mass2.join (", "));  
// ["Один", "Два", "Три", "Четвертый", "Пятый"]
```

## Методы для работы с массивами

`join(<Разделитель>)` — возвращает строку, полученную в результате объединения всех элементов массива через разделитель:

```
const Mass = [ "Один", "Два", "Три" ];  
const Str = Mass.join(" - ") ;  
console.log(Str); // Один - Два - Три
```

`shift()` — удаляет первый элемент массива и возвращает его:

```
const Mass = [ "Один", "Два", "Три" ];  
console.log(Mass.shift()); // Один  
console.log(Mass); // ["Два", "Три"]
```

`pop()` — удаляет последний элемент массива и возвращает его:

```
const Mass = [ "Один", "Два", "Три" ] ;  
console.log (Mass.pop()); // Три  
console.log (Mass); // ["Один", "Два"]
```

`reverse` — переворачивает массив. Элементы будут следовать в обратном порядке относительно исходного массива:

```
const Mass = [ "Один", "Два", "Три" ];  
Mass.reverse();  
console.log (Mass); // ["Три", "Два", "Один"]
```

`slice(<начало>, [<конец>])` — возвращает срез массива, начиная от индекса `<начало>` и заканчивая индексом `<конец>`, но не включает элемент с этим индексом. Если второй параметр не указан, то возвращаются все элементы до конца массива:

```
const Mass1 = [ 1, 2, 3, 4, 5 ];  
const Mass2 = Mass1.slice(1, 4);  
console.log (Mass2) ; // [2, 3, 4]  
const Mass3 = Mass1.slice(2) ;  
console.log (Mass3) ; // [3, 4, 5]
```

`sort([Функция сортировки])` — выполняет сортировку массива. Если функция не указана, будет выполнена обычная сортировка (числа сортируются по возрастанию, а символы — по алфавиту):

```
const Mass = [ "Один", "Два", "Три" ];  
Mass.sort();  
console.log(Mass); // ["Два", "Один", "Три"]
```

# Методы для работы с массивами

Если нужно изменить стандартный порядок сортировки, это можно сделать с помощью функции сортировки. Функция принимает две переменные и должна возвращать:

- ▶ 1 — если первый больше второго;
- ▶ -1 — если второй больше первого;
- ▶ 0 — если элементы равны.

Например, изменим стандартную сортировку на свою сортировку без учёта регистра символов

```
function f_sort(Str1, Str2) { // Сортировка без учёта регистра
    const Str1_1 = Str1.toLowerCase(); //Преобразуем к нижнему регистру
    const Str2_1 = Str2.toLowerCase(); //Преобразуем к нижнему регистру
    if (Str1_1>Str2_1) return 1;
    if (Str1_1<Str2_1) return -1;
    return 0;
}
const Mass = [ "единица1", "Единый", "Единица2" ];
Mass.sort(f_sort); // Имя функции указывается без скобок
console.log(Mass); // ["единица1", "Единица2", "Единый"]
```

Для этого две переменные приводим к одному регистру, а затем производим стандартное сравнение. При этом регистр самих элементов массива не изменяется, т.к. создаются их копии.

`splice(<Начало>, <Количество>, [<Список значений>])` — позволяет удалить, заменить или вставить элементы массива. Возвращает массив, состоящий из удаленных элементов:

```
const Mass1 = [ 1, 2, 3, 4, 5 ];
const Mass2 = Mass1.splice(2, 2);
console.log(Mass1); // [1, 2, 5]
console.log(Mass2); // [3, 4]
const Mass3 = Mass1.splice (1, 1, 7, 8, 9);
console.log(Mass1); // [1, 7, 8, 9, 5]
console.log(Mass3); // [2]
const Mass4 = Mass1.splice(1, 0, 2, 3, 4);
console.log(Mass1); // [1, 2, 3, 4, 7, 8, 9, 5]
console.log(Mass4); // []
```

`toString()` и `valueOf()` — преобразуют массив в строку. Элементы указываются через запятую без пробела:

```
const Mass = [ "Один", "Два", "Три" ];
console.log(Mass.toString()); // Один,Два,Три
```



## Методы для работы с массивами

Метод	Возвращаемое значение
<code>every()</code>	<code>true</code> , если каждый элемент массива удовлетворяет условию
<code>filter()</code>	Массив, который содержит все элементы текущего массива удовлетворяющие условию
<code>forEach()</code>	Выполняет заданную функцию для каждого элемента массива
<code>includes()</code>	Проверка вхождения значения в массив
<code>indexOf()</code>	Индекс первого вхождения заданного значения или <code>-1</code>
<code>lastIndexOf()</code>	Индекс последнего вхождения заданного значения или <code>-1</code>
<code>map()</code>	Массив, полученный путём преобразования каждого элемента
<code>reduce()</code>	Сводит два значения массива в одно, применяя к обоим заданную функцию (слева направо)
<code>reduceRight()</code>	Сводит два значения массива в одно, применяя к обоим заданную функцию (справа налево)
<code>some()</code>	<code>true</code> , если один или несколько элементов массива удовлетворяют условию
<code>fill()</code>	Заменяет все элементы массива от <code>startIndex</code> до <code>endIndex</code> (исключая <code>endIndex</code> ) заданным значением
<code>find()</code>	Возвращают элемент/индекс массива, если он удовлетворяет
<code>findIndex()</code>	условиям функции проверки, или <code>undefined</code>
<code>copyWithin()</code>	Копирует последовательность значений массива в другое место в этом же массиве.

Для передачи значений из одного массива в другой можно использовать оператор расширения. Следующий пример, демонстрирует, как сделать значения существующего массива частью другого массива при его создании.

```
let array1 = [2,3,4];  
let array2 = [1, ...array1, 5, 6, 7];  
console.log(array2); // Выведет "1, 2, 3, 4, 5, 6, 7"
```

Копирование значений из одного массива в конец другого массива можно выполнить следующим образом:

```
let array1 = [2,3,4];  
let array2 = [1];  
array2.push(...array1);  
console.log(array2); // Выведет "1, 2, 3, 4"
```

# Многомерные массивы

В любом элементе массива могут храниться данные любого типа, в том числе и другие массивы, например:

```
Mass1[0] = [1, 2, 3, 4];
```

В этом случае получить значение массива можно, указав два индекса:

```
Str = Mass1[0][2]; // Переменной Str будет присвоено значение 3
```

Так как массив является объектом, то операция присваивания сохраняет в переменной ссылку на массив, а не копию его значений. Например, если попробовать сделать так:

```
let Mass1, Mass2;  
Mass1 = [1, 2, 3, 4];  
Mass2 = Mass1; // Присваивается ссылка на массив  
Mass2[0] = "Новое значение";  
console.log(Mass1);  
console.log(Mass2);
```

то изменение Mass2 затронет Mass1, и мы получим следующий результат:

```
["Новое значение", 2, 3, 4]  
["Новое значение", 2, 3, 4]
```

Многомерные массивы также можно создать с помощью объекта Array:

```
const Mass = new Array(new Array("Один", "Два", "Три"),  
                        new Array("Четыре", "Пять", "Шесть"));  
console.log(Mass[0][1]); // "Два"
```

или поэлементно:

```
const Mass = new Array();  
Mass[0] = new Array();  
Mass[1] = new Array();  
Mass[0][0] = "Один";  
Mass[0][1] = "Два";  
Mass[0][2] = "Три";  
Mass[1][0] = "Четыре";  
Mass[1][1] = "Пять";  
Mass[1][2] = "Шесть";  
console.log(Mass[1][2]); // "Шесть"
```

При использовании многомерных массивов метод `slice()` создаёт «поверхностную» копию, а не полную:

```
let Mass1, Mass2;  
Mass1 = [[0, 1], 2, 3, 4];  
Mass2 = Mass1.slice(0);  
Mass2[0][0] = "Новое значение1";  
Mass2[1] = "Новое значение2";
```

В результате массивы будут выглядеть так:

```
Mass1 = [["Новое значение1", 1], 2, 3, 4];  
Mass2 = [["Новое значение1", 1], "Новое значение2", 3, 4];
```

Как видно из примера, изменение вложенного массива в `Mass2` привело к одновременному изменению значения в `Mass1`. Иными словами, оба массива содержат ссылку на один и тот же вложенный массив.

Класс Math содержит математические константы и функции. Его использование не требует создания экземпляра класса.

Свойства:

- ▶ E —  $e$ , основание натурального логарифма;
- ▶ LN2 — натуральный логарифм 2;
- ▶ LN10 — натуральный логарифм 10;
- ▶ LOG2E — логарифм по основанию 2 от  $e$ ;
- ▶ LOG10E — десятичный логарифм от  $e$ ;
- ▶ PI — число Пи:

```
console.log(Math.PI); // 3.141592653589793
```

- ▶ SQRT1\_2 — квадратный корень из 0.5;
- ▶ SQRT2 — квадратный корень из 2.

## Методы:

- ▶ `abs()` — абсолютное значение;
- ▶ `sin()`, `cos()`, `tan()` — стандартные тригонометрические функции (синус, косинус, тангенс). Значения углов указываются в радианах;
- ▶ `sign()` возвращает знак числа: 1 если число положительное, -1, если число отрицательное и 0, если число равно нулю:  

```
console.log(Math.sign(11)); // 1  
console.log(Math.sign(-11)); // -1  
console.log(Math.sign(0)); // 0
```
- ▶ `pow(<Число>, <Степень>)` — возведение <Числа> в <Степень>:  

```
const x = 5;  
console.log(Math.pow(x, 2)); // 25 (5 в квадрате)
```
- ▶ `sqrt()` — квадратный корень:  

```
const x = 25;  
console.log(Math.sqrt(x)); // 5 (квадратный корень из 25)
```
- ▶ `hypot()` — квадратный корень из суммы квадратов аргументов:  

```
console.log(Math.hypot(2, 2, 1)); // 3
```
- ▶ `max(<Список чисел через запятую>)` — максимальное значение из списка:  

```
console.log(Math.max(3, 10, 6)); // 10
```
- ▶ `min(<Список чисел через запятую>)` — минимальное значение из списка:

- ▶ `round()` — значение, округленное до ближайшего целого. Если первое число после запятой от 0 до 4, то округление производится к меньшему по модулю целому, а в противном случае — к большему:

```
const x = 2.499;  
const y = 2.5;  
console.log(Math.round(x)); // округлено до 2  
console.log(Math.round(y)); // округлено до 3
```

- ▶ `ceil()` — значение, округленное до ближайшего большего целого:

```
const x = 2.499;  
const y = 2.5;  
console.log(Math.ceil(x)); // округлено до 3  
console.log(Math.ceil(y)); // округлено до 3
```

- ▶ `floor()` — значение, округленное до ближайшего меньшего целого:

```
const x = 2.499;  
const y = 2.5;  
console.log(Math.floor(x)); // округлено до 2  
console.log(Math.floor(y)); // округлено до 2
```

- ▶ `random()` — случайное число в диапазоне `[0;1)`:

```
console.log(Math.random()); // например, 0.9778613566886634
```



## Оператор возведения в степень

В ECMAScript 2016 появился оператор возведения в степень, выполняющий ту же математическую операцию что и метод `Math.pow()`. Оператор возведения в степень имеет форму двух звёздочек (`**`): левый операнд используется как основание, а правый — как степень. Например:

```
let result = 5 ** 2;
```

```
console.log(result);           // 25  
console.log(result === Math.pow(5, 2)); // true
```

Этот пример вычисляет выражение  $5^2$ , результат которого равен 25. Оператор возведения в степень имеет высший приоритет из всех двухместных операторов в JavaScript (унарные операторы имеют более высокий приоритет, чем `**`). Это означает, что он выполняется первым в любом сложном выражении, например:

```
let result = 2 * 5 ** 2;  
console.log(result); // 50
```

Здесь сначала будет найден результат  $5^2$ , затем полученное значение умножается на 2. Конечный результат получится равным 50.

# Оператор возведения в степень

Оператор возведения в степень накладывает некоторые необычные ограничения, отсутствующие в других операторах. Левый операнд не может быть выражением с унарным оператором, кроме ++ и --. Например, следующий пример вызовет синтаксическую ошибку:

```
// синтаксическая ошибка  
let result = -5 ** 2;
```

Выражение -5 в данном примере расценивается как синтаксическая ошибка, потому что возникает неоднозначность в определении порядка выполнения операций. Должен ли унарный оператор - применяться к числу 5 или к результату выражения 5 \*\* 2? Запрет использования унарных выражений слева от оператора возведения в степень устраняет эту неоднозначность. Чтобы ясно обозначить свои намерения, следует заключить в круглые скобки -5 или 5 \*\* 2, как показано ниже:

```
// правильно  
let result1 = -(5 ** 2); // результат равен -25  
// тоже правильно  
let result2 = (-5) ** 2; // результат равен 25
```

Если заключить в круглые скобки выражение, унарный оператор - будет применен ко всему выражению. Если заключить в круглые скобки -5, интерпретатор поймет, что во вторую степень требуется возвести число -5.

# Оператор возведения в степень

Выражения с операторами ++ и -- слева от оператора возведения в степень не требуется заключать в скобки, потому что поведение обоих операторов ясно определено как направленное на их операнды.

Префиксный оператор ++ или -- изменяет свой операнд перед выполнением любой другой операции, а постфиксные версии ничего не изменяют, пока все выражение не будет вычислено. В обоих случаях эти операторы не вызывают ошибок при использовании слева от оператора возведения в степень, например:

```
let num1 = 2,  
    num2 = 2;  
  
console.log(++num1 ** 2); // 9  
console.log(num1); // 3  
console.log(num2-- ** 2); // 4  
console.log(num2); // 1
```

В этом примере значение `num1` увеличивается перед выполнением оператора возведения в степень, поэтому `num1` получает значение 3, и в результате операции получается 9. Переменная `num2` сохраняет значение 2 перед выполнением оператора возведения в степень и затем уменьшается до 1.

Даты хранятся как количество миллисекунд, прошедших с полуночи 1 января 1970 г. согласно универсальному временному коду (Universal Time Code, UTC). Благодаря такому формату с помощью объектов Date можно точно представлять даты, отстоящие от 1 января 1970 г. на 285 616 лет. В ECMAScript 5 добавлен метод `Date.now()`, который возвращает дату и время его выполнения в миллисекундах. Это позволяет использовать объекты Date для профилирования кода:

```
// получение времени начала  
let start = Date.now();
```

```
// вызов функции  
doSomething();
```

```
// получение времени окончания  
let stop = Date.now(),  
result = stop - start;
```

Метод `Date.now()` реализован в Internet Explorer 9+, Firefox 3+, Safari 3+, Opera 10.5 и Chrome.

JavaScript переменные могут содержать значения двух видов: примитивные и ссылочные.

*Примитивные значения* (primitive values) — это просто атомарные элементы данных, в то время как *ссылочные значения* (reference values) — это объекты, которые могут состоять из нескольких значений.

Примитивные значения относятся к одному из пяти примитивных типов данных: неопределённому (undefined), нулевому (null), логическому (boolean), числовому (number) и строковому (string). Доступ к переменным этих типов осуществляется по значению (by value), то есть вы работаете с фактическим значением, хранящимся в переменной.

Примитивные значения имеют фиксированный размер и хранятся в памяти в стеке.

Ссылочные значения — это объекты, хранящиеся в памяти в куче. При выполнении каких-либо действий над объектом вы на самом деле работаете не с самим объектом, а со *ссылкой* (reference) на него. Говорят, что доступ к таким значениям осуществляется *по ссылке*.

## Копирование значений

Когда одна переменная с примитивным значением присваивается другой, создаётся копия значения, хранящегося в объекте переменных, а затем она записывается по адресу новой переменной, например:

```
let num1 = 5;  
let num2 = num1;
```

Здесь `num1` содержит значение 5. Когда переменная `num2` инициализируется значением `num1`, она также получает значение 5. Она никак не связана с `num1`, потому что содержит копию значения. Затем эти переменные можно использовать по отдельности.

Когда ссылочное значение одной переменной присваивается другой, то после копирования обе переменные указывают на один объект, поэтому изменения одной из них отражаются на другой:

```
let obj1 = {};  
let obj2 = obj1;  
obj1.name = "Nicholas";  
console.log(obj2.name);    // Nicholas
```

В этом примере переменной `obj1` присваивается новый объект, после чего значение переменной копируется в `obj2`. Теперь обе переменные указывают на один объект, и когда для `obj1` задаётся свойство `name`, оно становится доступным и через `obj2`.

## Передача аргументов

Все аргументы функций в JavaScript передаются по значению. Это означает, что значения извне функции копируются в аргументы внутри функции так же, как копируются значения переменных: примитивные — по одному сценарию, ссылочные — по другому. Когда аргумент передаётся по значению, его значение просто копируется в локальную переменную.

```
function addTen(num) {  
    num += 10;  
    return num;  
}  
  
let count = 20;  
let result = addTen(count);  
console.log(count); // 20 - без изменений  
console.log(result); // 30
```

При вызове функции `addTen()` ей передаётся переменная `count` со значением 20, которое копируется в аргумент `num` для использования в функции `addTen()`. В функции к значению `num` прибавляется 10, но это не изменяет значение `count` вне функции. Аргумент `num` и переменная `count` не знают друг о друге, они просто имеют одинаковые значения. Если бы значение `num` было передано по ссылке, переменная `count` также стала бы равной 30 согласно изменению внутри функции.

## Передача аргументов

При передаче аргумента по ссылке в локальной переменной сохраняется расположение его значения в памяти. Это означает, что изменения локальной переменной отражаются вне функции.

```
function setName(obj) {  
    obj.name = "Nicholas";  
}  
  
let person = {};  
setName(person);  
console.log(person.name); // Nicholas
```

Внутри функции переменные `obj` и `person` указывают на один и тот же объект. Поэтому при задании свойства `name` в функции это изменение отражается вне функции. Тем не менее, объекты все равно передаются по значению:

```
function setName(obj) {  
    obj.name = "Nicholas";  
    obj = {};  
    obj.name = "Greg";  
}  
  
let person = {};  
setName(person);  
console.log(person.name); // Nicholas
```



# Объектная модель браузера

Объектная модель браузера — это совокупность объектов, обеспечивающих доступ к содержимому Web-страницы и ряду функций Web-браузера.

Объектная модель представлена в виде иерархии объектов. То есть имеется объект верхнего уровня и подчинённые ему объекты. В свою очередь подчинённые объекты имеют свои подчинённые объекты. Часто объект верхнего уровня (и даже подчинённый объект) можно не указывать. Рассмотрим выражение для вызова диалогового окна с сообщением:

```
window.alert("Сообщение");
```

Здесь `window` — это объект самого верхнего уровня, представляющий сам Web-браузер, а `alert()` — это метод объекта `window`. В этом случае указывать объект не обязательно, т. к. объект `window` подразумевается по умолчанию:

```
alert("Сообщение");
```

При печати сообщения в окне Web-браузера часто используется команда:

```
console.log("Сообщение");
```

Поскольку объект `console` является подчинённым объекту `window`, то правильно было бы написать так:

```
window.console.log("Сообщение");
```

Помимо уже упомянутого объекта самого высокого уровня — `window` в объектной модели имеются следующие основные объекты:

- ▶ `event` — предоставляет информацию, связанную с событиями.
- ▶ `frame` — служит для работы с фреймами (коллекция `frames`);
- ▶ `history` — предоставляет доступ к списку истории Web-браузера;
- ▶ `navigator` — содержит информацию о Web-браузере;
- ▶ `location` — содержит URL-адрес текущей Web-страницы;
- ▶ `screen` — служит для доступа к характеристикам экрана компьютера пользователя;
- ▶ `document` — служит для доступа к структуре, содержанию и стилю документа.

Основная область браузера называется *окном* (window). Это та область, в которую загружаются HTML-документы (и связанные с ними ресурсы). Каждая вкладка браузера представляется в JavaScript экземпляром объекта window. Методы объекта window перечислены в табл.:

Метод	Использование
alert()	Отображает окно оповещения, которое содержит текст сообщения и кнопку ОК
blur()	Делает текущее окно неактивным
close()	Закрывает текущее окно
confirm()	Отображает окно подтверждения, которое содержит сообщение и две кнопки — ОК и Cancel (Отмена)
createPopup()	Создает всплывающее окно
focus()	Делает текущее окно активным
moveBy()	Перемещает текущее окно на заданную величину
moveTo()	Перемещает текущее окно в заданную позицию
open()	Открывает новое окно Web-браузера
showModalDialog()	Открывает модальное диалоговое окно
prompt()	Отображает окно запроса, ожидающее пользовательского ввода

Кроме того, имеются четыре метода для работы с таймерами.

Таймеры позволяют однократно или многократно выполнять указанную функцию через определённый интервал времени. Для управления таймерами используются следующие методы объекта `window`:

- ▶ `setTimeout()` — создаёт таймер, однократно выполняющий указанную функцию или выражение спустя заданный интервал времени:

```
<Идентификатор> = setTimeout(<Функция или выражение>, <Интервал>);
```

- ▶ `clearTimeout(<Идентификатор>)` — останавливает таймер, установленный методом `setTimeout()`.
- ▶ `setInterval()` — создаёт таймер, многократно выполняющий указанную функцию или выражение через заданный интервал времени.

```
<Идентификатор> = setInterval(<Функция или выражение>, <Интервал>);
```

- ▶ `clearInterval(<Идентификатор>)` — останавливает таймер, установленный методом `setInterval()`.

Здесь `<Интервал>` — это промежуток времени, по истечении которого выполняется `<Функция или выражение>`. Значение указывается в миллисекундах.

## Использование свойств и методов объекта document

DOM (Document Object Model — объектная модель документа) — это интерфейс, посредством которого JavaScript работает с HTML-документами в окнах браузера.

Объект document предоставляет доступ ко всем элементам Web-страницы. Некоторые методы объекта document:

Метод	Использование
<code>addEventListener()</code>	Назначает обработчик событий в документе
<code>createAttribute()</code>	Создаёт узел атрибута
<code>createComment()</code>	Создает узел комментария
<code>createElement()</code>	Создает узел элемента
<code>createDocumentFragment()</code>	Создает пустой фрагмент документа
<code>getElementById()</code>	Получает элемент по заданному атрибуту id
<code>getElementsByName()</code>	Получает все элементы с указанным именем
<code>getElementsByTagName()</code>	Получает все элементы с указанным именем тега
<code>removeEventListener()</code>	Удаляет из документа обработчик событий, добавленный ранее <code>addEventListener()</code>
<code>write()</code>	Записывает текст, переданный как параметр, в текущее место документа

## Метод `getElementById()`

Метод `getElementById(<Идентификатор>)`, до сих пор чаще всего используемый для выбора элементов, занимает важное место в современной веб-разработке. Это весьма удобное средство, позволяющее находить и обрабатывать любой элемент по его идентификатору — уникальному атрибуту `id`.

Используя этот метод, можно работать с любым элементом в документе, где бы он ни находился, если известен его идентификатор.

Метод возвращает ссылку только на один элемент (даже если элементов с одинаковым идентификатором несколько), т.к. по определению идентификатор должен быть уникальным.

Пример:

```
const str = document.getElementById("check1").value;
```

# Использование свойств и методов объекта Element

Объект Element предоставляет свойства и методы для работы с HTML-элементами в документе.

Некоторые свойства объекта Element:

Свойство	Использование
innerHTML	Получает или устанавливает содержимое элемента
height	Высота элемента
width	Ширина элемента
clientHeight	Высота элемента без учёта рамок, границ, полос прокрутки и т.п.
clientWidth	Ширина элемента без учёта рамок, границ, полос прокрутки и т.п.
clientLeft	Смещение левого края элемента относительно левого края элемента-родителя без учёта рамок, границ, полос прокрутки и т.п.
clientTop	Смещение верхнего края элемента относительно верхнего края элемента-родителя без учёта рамок, границ, полос прокрутки и т.п.
offsetHeight	Высота элемента относительно элемента-родителя
offsetWidth	Ширина элемента относительно элемента-родителя;
offsetLeft	Смещение левого края элемента относительно левого края элемента-родителя

Самым важным свойством элемента, которое можно изменить посредством DOM, является свойство `innerHTML`:

```
...  
<p id="paragraph">Это <i>простой</i> HTML-файл.</p>  
...  
// Получить ссылку на элемент с атрибутом id = paragraph  
const par = document.getElementById('paragraph');  
par.innerHTML; // "Это <i>простой</i> HTML-файл."  
...  
par.innerHTML = "<i>Измененный</i> HTML-файл";
```

Таким образом с помощью свойства `innerHTML` можно также вставлять теги HTML.



## Методы объекта Element:

Метод	Использование
<code>addEventListener()</code>	Назначает обработчик событий для элемента
<code>removeEventListener()</code>	Удаляет указанный слушатель событий
<code>click()</code>	Имитирует щелчок мышью на элементе
<code>blur()</code>	Лишает элемент фокуса ввода
<code>focus()</code>	Переводит фокус ввода на данный элемент
<code>getAttribute()</code>	Получает значение указанного атрибута элемента
<code>setAttribute()</code>	Устанавливает указанное значение для атрибута
<code>removeAttribute()</code>	Удаляет заданный атрибут из элемента
<code>hasAttribute()</code>	Возвращает значение <code>true</code> , если элемент содержит указанный атрибут, иначе — <code>false</code>
<code>appendChild()</code>	Вставляет новый дочерний узел в элемент (в качестве последнего дочернего узла)
<code>insertBefore()</code>	Вставляет новый дочерний узел перед заданным существующим узлом
<code>replaceChild()</code>	Заменяет указанный дочерний узел другим
<code>hasChildNodes()</code>	Возвращает значение <code>true</code> , если элемент содержит дочерние узлы, иначе — <code>false</code>
<code>removeChild()</code>	Удаляет заданный дочерний узел
<code>toString()</code>	Преобразует элемент в строку

Концепция контекста выполнения, или просто *контекста* (context), очень важна в JavaScript. От контекста выполнения переменной или функции зависит, какие другие данные ей доступны и как она должна работать. С каждым контекстом выполнения связан *объект переменных* (variable object), содержащий все переменные и функции контекста. Он недоступен в коде, но используется за кулисами для обработки данных.

Наиболее общим является глобальный контекст выполнения. В веб-браузерах глобальным контекстом считается контекст объекта window, так что все глобальные переменные и функции создаются как свойства и методы объекта window. Когда весь код в контексте выполнен, он уничтожается вместе со всеми определёнными в нём переменными и функциями (глобальный контекст существует вплоть до закрытия веб-страницы или веб-браузера).

У каждого вызова функции имеется свой локальный контекст выполнения. При передаче управления в функцию её контекст помещается в стек контекста, а при выходе из функции он извлекается из стека, при этом управление возвращается в прежний контекст. Так осуществляется контроль над последовательностью операций в JavaScript-программе.

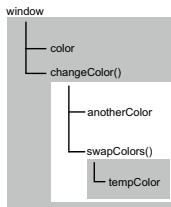
При выполнении кода в контексте создаётся *цепочка областей видимости* (score chain) объектов переменных, которая обеспечивает упорядоченный доступ ко всем переменным и функциям, доступным в контексте выполнения. Первым звеном цепочки областей видимости всегда является объект переменных контекста, код которого выполняется. Каждый последующий объект переменных в цепочке относится к все более внешнему контексту, пока не достигается глобальный контекст. Объект переменных глобального контекста всегда последний в цепочке областей видимости.

При разрешении идентификатора его имя ищется в цепочке областей видимости. Поиск всегда осуществляется в направлении от первого звена к последнему, пока не обнаруживается идентификатор (если найти его не удаётся, обычно возникает ошибка).

Аргументы функций считаются переменными и подчиняются тем же правилам доступа, что и любые другие переменные в контексте выполнения.

## Контекст выполнения и область видимости

```
let color = "blue";
function changeColor(){
  let anotherColor = "red";
  function swapColors(){
    let tempColor = anotherColor;
    anotherColor = color;
    color = tempColor;
    // здесь доступны переменные color, anotherColor и tempColor
  }
  // здесь доступны переменные color и anotherColor, но не tempColor
  swapColors();
}
// здесь доступна только переменная color
changeColor();
```



JavaScript — это язык с автоматической сборкой мусора, то есть за управление памятью при работе сценариев отвечает среда выполнения. В языках вроде C и C++ слежение за использованием памяти относится к важнейшим задачам и является источником многих проблем. JavaScript освобождает разработчиков от забот по управлению памятью, автоматически выделяя сценариям нужную память и возвращая в среду память, которая больше не используется. Сборщик мусора запускается периодически с заданной частотой или в predetermined моменты выполнения кода.

Как правило, объём памяти, доступной в веб-браузерах, гораздо меньше, чем в обычных приложениях. Он ограничивается в основном ради безопасности, чтобы сценарии JavaScript в веб-страницах не могли вызвать сбой операционной системы, израсходовав всю системную память. Ограничения памяти влияют не только на выделение памяти для переменных, но и на стек вызовов и количество инструкций, выполняемых в одном потоке.

Если данные больше не нужны, то для оптимизации сборки мусора, лучше всего присвоить соответствующей переменной значение `null`. Этот совет относится преимущественно к глобальным переменным и свойствам глобальных объектов. В случае локальных переменных ссылки на данные удаляются автоматически, когда переменные покидают контекст:

```
function createPerson(name){  
    let localPerson = {};  
    localPerson.name = name;  
    return localPerson;  
}  
  
let globalPerson = createPerson("Nicholas");  
// какие-то действия с globalPerson  
globalPerson = null;
```

В этом коде переменная `localPerson` покидает контекст при завершении функции `createPerson()`, так что разрывать её связь со значением не требуется. Что касается `globalPerson`, то это глобальная переменная, поэтому когда она больше не требуется, ей присваивается значение `null`. Разрыв связи переменной со значением не приводит к автоматическому освобождению занятой им памяти. Значение просто выводится из контекста, что делает возможным возвращение памяти при следующей сборке мусора.

*События* — это все то, что происходит в браузере (например, загрузка веб-страницы), а также то, что делает пользователь (например, щелчки мышью, нажатия клавиш, перемещения мыши, изменение размеров окна, и т.п.). В браузере постоянно происходят какие-то события.

HTML DOM позволяет JavaScript распознавать события, происходящие в браузере, и реагировать на них. События можно разделить на отдельные группы в соответствии с тем, с какими HTML-элементами или объектами браузера они связаны.

## События, поддерживаемые всеми HTML-элементами

Событие	Условие возникновения
click	Выполнения щелчка мышью на элементе
dblclick	Выполнение двойного щелчка мышью на элементе
keydown	Нажатие клавиши. Наступает постоянно до отпускания
keypress	Аналогично событию <code>keydown</code> , но возвращает значение кода символа в кодировке Unicode.
keyup	Отпускание клавиши после того, как она была нажата
mousedown	Нажатие кнопки мыши над элементом
mouseenter	Перемещение указателя мыши в область элемента, к которому подключен слушатель событий
mouseleave	Перемещение указателя мыши за пределы элемента, к которому подключен слушатель событий
mousemove	Перемещение указателя мыши над элементом
mouseout	Перемещение указателя мыши за пределы элемента или одного из его дочерних элементов, к которому подключен слушатель событий
mouseover	Перемещение указателя мыши над элементом или одним из его дочерних элементов, к которому подключен слушатель событий
mouseup	Отпускание кнопки мыши над элементом
mousewheel	Вращение колесика мыши



## События, поддерживаемые всеми элементами кроме <body>, <frameset>

Событие	Условие возникновения
blur	Потеря элементом фокуса ввода
change	Пользователь вносит изменение в поле
drag	Событие перетаскивания
error	Элемент не доступен (например, некорректная ссылка, путь)
focus	Приобретение элементом фокуса ввода
load	Браузер загрузил элемент и все его зависимые ресурсы
resize	Изменение размера окна
scroll	Прокрутка документа или элемента
afterprint	Начало вывода документа на печать
beforeprint	Открытие окна предварительного просмотра выводимого на печать документа или завершение подготовки документа к печати
beforeunload	Окно, документ и подключенные к нему файлы подготовлены к выгрузке
pagehide	Браузер покидает страницу в истории просмотра
pageshow	Браузер переходит на страницу в истории сеанса
popstate	Изменение элемента истории просмотра в активном сеансе
unload	Выгрузка документа или включенного в него файла. Наступает после события beforeunload;

# Последовательность событий

События возникают последовательно, например, последовательность событий при нажатии кнопки мыши на элементе страницы будет такой:

```
mousedown  
mouseup  
click
```

При двойном нажатии последовательность будет такой:

```
mousedown  
mouseup  
click  
dblclick
```

Это значит, что событие `dblclick` возникает после события `click`.  
При нажатии клавиши на клавиатуре последовательность будет такой:

```
keydown  
keypress  
keyup
```

Последовательность возникновения событий также может зависеть от используемого браузера.

Обработчик события — это асинхронная функция обратного вызова, обрабатывающая события ввода от пользователя, такие как щелчки мышью или нажатия клавиш на клавиатуре. Возможность определения обработчиков позволяет создавать динамические веб-страницы и изменять их содержимое в соответствии с вводом пользователя.

Первая из систем обработки событий была введена одновременно с выпуском первых версий JavaScript. Она основана на использовании специальных атрибутов обработчиков событий.

Встроенные атрибуты обработчиков событий образуются путём добавления префикса `on` к имени события. Их использование сводится к добавлению атрибута события в HTML-элемент. Когда происходит указанное событие, выполняется JavaScript-код, записанный в виде значения атрибута:

```
<a href="home.html" onclick="alert('Перейти на главную страницу!')";  
Щёлкните здесь для перехода на главную страницу</a>
```

При щёлчке по ссылке на экране отобразится окно сообщения с текстом «Перейти на главную страницу!». Когда вы закроете это окно, выполнится заданный по умолчанию обработчик событий для этого элемента, который осуществит переход по ссылке, указанной в атрибуте `href`.

Одним из наибольших недостатков, связанных с использованием устаревшей техники встраивания обработчиков событий в элементы, является то, что при этом нарушается главный принцип наилучшей практики программирования, а именно отделение кода представления (ответственного за внешний вид объектов) от функционального кода (ответственного за выполнение полезных действий). Смешивание обработчиков событий с тегами HTML затрудняет сопровождение веб-страниц и их отладку, одновременно затрудняя понимание кода. В версии 3 браузера Netscape была введена новая модель событий, позволяющая программистам подключать события к элементам в виде свойств:

```
document.getElementById("incrementButton").onclick = increaseCount;

function increaseCount(){
    ...
}
```

Обратите внимание, что за именами функций, назначаемых обработчику событий, не следуют круглые скобки. Такой синтаксис не означает вызова функции и лишь указывает на то, что эта функция должна выполняться всякий раз, когда происходит данное событие.

Назначить обработчик события в модели DOM Level 2 позволяет метод `addEventListener()`. Метод `addEventListener()` прослушивает события на любом DOM-узле и запускает выполнение заданных действий в зависимости от наступившего события. Когда запускается функция, указанная для обработчика события в качестве выполняемого действия, она автоматически получает единственный аргумент — объект `Event`. По сравнению с использованием атрибутов DOM-событий метод `addEventListener()` обладает следующими преимуществами:

- ▶ одному элементу может быть назначено несколько обработчиков событий;
- ▶ работает на любом узле DOM, а не только на элементах;
- ▶ предоставляет большую степень контроля над обработкой события.

Удалить обработчик события можно с помощью метода `removeEventListener()`.

## Обработка событий с использованием метода `addEventListener()`

Пример использования метода `addEventListener()`:

```
document.getElementById("incrementButton").  
    addEventListener('click',increaseCount,false);  
function increaseCount(e){ // e - ссылка на объект Event  
    window.alert("Элемент " + this.getAttribute("id"));  
}
```

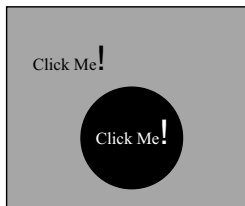
Метод `addEventListener()` реализован с использованием трёх аргументов.

Первый аргумент — это тип события. В отличие от двух других способов обработки событий, метод `addEventListener()` требует указания лишь имени события без префикса `on`, например, `click` вместо `onclick`.

Второй аргумент — это функция, которая должна вызываться при наступлении события. В эту функцию в качестве аргумента передаётся ссылка на объект `event`, а внутри функции через ключевое слово `this` доступна ссылка на текущий элемент.

Третий аргумент — это булево значение (`true` или `false`), которое определяет очередность выполнения обработчиков событий в тех случаях, когда у элемента, с которым связано событие, имеется родительский элемент, также связанный с событием.

## Обработка событий с использованием метода `addEventListener()`



В случае вложенных элементов важно знать, какое из двух событий произойдет первым. Например, если щёлкнуть на внутреннем круге, то какое событие должно произойти первым: связанное с квадратом или с кругом?

Наиболее распространённой стратегией обработки подобных событий является модель *всплытия событий*. События, относящиеся к наиболее глубоко вложенным элементам, происходят первыми, а затем «всплывают» в направлении наружных элементов. Чтобы использовать эту стратегию, следует установить для последнего аргумента метода `addEventListener()` значение `false`, которое является также значением по умолчанию и используется в большинстве случаев.

Другой способ обработки подобных сценариев называется *захватом событий*. В этом случае первыми происходят внешние события, а последними — внутренние.

Объект Event позволяет получить детальную информацию о произошедшем событии и выполнить необходимые действия. Объект event доступен только в обработчиках событий. При наступлении следующего события все предыдущие значения свойств сбрасываются.

Объект event имеет следующие свойства:

- ▶ `clientX` и `clientY` — координаты события (по осям  $X$  и  $Y$ ) в клиентских координатах;
- ▶ `screenX` и `screenY` — координаты события (по осям  $X$  и  $Y$ ) относительно окна;
- ▶ `offsetX` и `offsetY` — координаты события (по осям  $X$  и  $Y$ ) относительно контейнера;
- ▶ `x` и `y` — координаты события по осям  $X$  и  $Y$ . В модели DOM Level 2 этих свойств нет;
- ▶ `button` — число, указывающее нажатую кнопку мыши. Может принимать следующие значения:
  - ▶ 0 — кнопки не были нажаты;
  - ▶ 1 — нажата левая кнопка мыши;
  - ▶ 2 — нажата правая кнопка мыши;
  - ▶ 3 — левая и правая кнопки мыши были нажаты одновременно;
  - ▶ 4 — нажата средняя кнопка.

В модели DOM Level 2 значения другие:

- ▶ 0 — нажата левая кнопка мыши;
- ▶ 1 — нажата средняя кнопка;
- ▶ 2 — нажата правая кнопка мыши;



- ▶ `key` — возвращает значение клавиши, нажатой пользователем.
- ▶ `altKey` — `true`, если в момент события была нажата клавиша `<Alt>`;
- ▶ `ctrlKey` — `true`, если была нажата клавиша `<Ctrl>`;
- ▶ `shiftKey` — `true`, если была нажата клавиша `<Shift>`;
- ▶ `target` — содержит объект, породивший событие.

Во многих случаях может оказаться желательным, чтобы действие по умолчанию, связанное с данным элементом, не было выполнено. Например, если пользователь щелкает на ссылке `<a>`, то браузер обработает это событие, загрузив нужную страницу. Метод `preventDefault()` позволяет предотвратить такое поведение:

```
<a href="file.html" onclick="f_event(event);">  
function f_event(e) {  
    e.preventDefault();  
    window.alert("Перехода по ссылке не будет!");  
}
```

Для управления вызовом вложенных событий существует три возможности.

1. Вызов метода `preventDefault` *отменяет* событие. Отменённые события продолжают распространяться, но их свойство `defaultPrevented` устанавливается равным `true`. Встроенные в браузер обработчики событий проверяют значение `defaultPrevented` и, если оно истинно, ничего не предпринимают. Обработчики событий, которые вы пишете, могут проигнорировать значение этого свойства (и обычно так и делают).
2. Вызов метода `stopPropagation` предотвращает дальнейшее распространение события за текущий элемент:

```
<body onclick="f_print(event);">
```

```
function f_print(e) {  
    e.stopPropagation();  
}
```

При этом все обработчики, связанные с текущим элементом, будут вызваны, но никакие из обработчиков, связанных с другими элементами не вызываются.

3. Вызов метода `stopImmediatePropagation` запретит вызов дальнейших обработчиков (даже если они связаны с текущим элементом).

Объект Promise используется для отложенных и асинхронных вычислений:

```
const promise1 = new Promise((resolve, reject) => {  
  setTimeout(() => {  
    resolve("Success!");  
  }, 300);  
});
```

```
promise1.then((value) => {  
  console.log(value);  
  // Expected output: "Success!"  
});
```

Первый аргумент — обработчик(функция) `resolve` вызывается в случае успешного завершения работы, в противном случае вызывается обработчик `reject`. Прикрепление обработчиков осуществляется с помощью метода `then`.

JavaScript предоставляет глобальный метод `fetch(url, [options])`, который позволяет получать ресурсы по сети асинхронно и возвращать объект `Promise`:

```
fetch('http://example.com/movies.txt')
  .then((response) => {
    return response.text();
  })
  .then((data) => {
    console.log(data);
  });
```

Здесь мы забираем обычный текстовый файл по сети и выводим его содержимое в консоль. Самый простой способ использования `fetch()` заключается в вызове этой функции с одним аргументом — строкой, содержащей путь к нужному ресурсу. Чтобы извлечь содержимое тела из HTTP-ответа, мы используем `text()` метод.

## Асинхронные функции и оператор await

Объявление `async function` определяет асинхронную функцию. После вызова асинхронная функция возвращает объект `Promise`.

Асинхронная функция может содержать выражение `await`, которое приостанавливает выполнение и ожидает ответа. Когда ответ получен, возвращается полученное значение и возобновляется выполнение.

```
function resolveAfter2Seconds(x) {  
  return new Promise((resolve) => {  
    setTimeout(() => {  
      resolve(x);  
    }, 2000);  
  });  
}  
  
async function add1(x) {  
  const a = await resolveAfter2Seconds(20);  
  const b = await resolveAfter2Seconds(30);  
  return x + a + b;  
}  
  
add1(10).then((v) => {  
  console.log(v); // prints 60 after 4 seconds.  
});
```

## Асинхронные функции и оператор await

```
async function add2(x) {  
  const a = resolveAfter2Seconds(20);  
  const b = resolveAfter2Seconds(30);  
  return x + (await a) + (await b);  
}
```

```
add2(10).then((v) => {  
  console.log(v); // prints 60 after 2 seconds.  
});
```

Ключевое слово `await` допустимо только в асинхронных функциях. В другом контексте вы получите ошибку `SyntaxError`.  
Типичный запрос с помощью `fetch` состоит из двух операторов `await`:

```
let response = await fetch(url); // завершается с заголовками ответа  
let result = await response.text(); // читать ответ в текстовом формате
```

Или, без `await`:

```
fetch(url, options)  
  .then(response => response.json())  
  .then(result => /* обрабатываем результат */)
```

Для загрузки содержимого графического файла в Java-Script-сценарий, удобнее всего использовать объект класса `Image`, который представляет графическое изображение. Для этого с помощью оператора `new` создаём объекта класса `Image`. Далее присваиваем свойству `src` этого объекта интернет-адрес загружаемого графического файла в виде строки:

```
const image = new Image(); // Создать объект изображения
// Зарегистрировать обработчик, вызываемый после загрузки изображения
image.onload = function(){loadTexture(gl,n,texture,u_Sampler,image)};
// Заставить браузер загрузить изображение
image.src = '../resources/sky.jpg';
```



В некоторых языках (например, в PHP) ошибки времени выполнения возникают из-за деления на ноль или обращения к несуществующему элементу массива. В языке JavaScript в этих случаях программа прервана не будет.

При попытке деления на ноль возвращается значение `Infinity`:

```
console.log(5/0); // Infinity
```

При обращении к несуществующему элементу массива возвращается значение `undefined`:

```
const arr = [1, 2];  
console.log(arr[20]); // undefined
```

В некоторых случаях требуется указать программе, что возникла неисправимая ошибка, и прервать выполнение всей программы. Для этого предназначен оператор `throw`:

```
if (d < 0)
    throw new Error("Переменная не может быть меньше нуля");
```

Сообщение будет выведено в окне браузера.

Браузеры с поддержкой консоли предоставляют объект `console`, у которого есть несколько методов для вывода данных в отладочном режиме. Спецификация этого объекта описана в Console API. Метод `console.log()` выводит данные в консоль из сценария. Он может выводить в консоль сразу несколько значений, разделенных запятыми:

```
console.log('Высота', height);
```

Чтобы разделять разные типы сообщений, которые записываются в консоль, их можно выводить с помощью трёх разных методов. Каждый из них отличается собственным цветом и значком:

1. `console.info()` используется для вывода общей информации.
2. `console.warn()` используется для вывода предупреждений.
3. `console.error()` используется для вывода ошибок.

С помощью метода `console.assert()` можно проверить, истинно ли условие, и выполнить запись в консоль, только если выражение вернуло значение `false`:

```
console.assert(this.value >= 10, 'Пользователь ввел число меньше 10');
```

*Строгий режим* (strict mode) позволяет реализовать более строгую проверку ошибок глобально или локально (в одной функции).

Преимущество строгого режима в том, что он позволяет раньше получать уведомления об ошибках, и вы можете оперативно устранять их.

Включить строгий режим можно с помощью следующей *директивы* (pragma) — строки, которая не назначается никакой переменной:

```
"use strict";
```

Если интерпретатор JavaScript не поддерживает строгий режим, эта директива просто игнорируется как строковый литерал, не назначенный никакой переменной.

Если эта директива применяется глобально, то есть вне функции, строгий режим включается для всего сценария. Строгий режим можно также включить только внутри отдельной функции, например:

```
function doSomething(){  
    "use strict";  
    // другие операции  
}
```

Если у вас нет полного контроля над всеми сценариями на странице, рекомендуется включать строгий режим только внутри конкретных функций, которые были протестированы в нём.

Способ и время создания переменных в строгом и обычном режимах различаются. Так, в строгом режиме нельзя создать глобальную переменную случайно. Например, следующий код в нестрогом режиме создает глобальную переменную:

```
// Переменная не объявлена  
// Обычный режим: создается глобальная переменная  
// Строгий режим: генерируется ошибка ReferenceError  
message = "Hello world!";
```

Хотя переменной `message` не предшествует ключевое слово `let` или `const` и она не определена как свойство глобального объекта, она все же автоматически создаётся как глобальная. В строгом режиме присваивание значения необъявленной переменной приводит при запуске кода к ошибке `ReferenceError`.

Сначала программы на JavaScript были небольшими — они использовались для изолированных задач, добавляя при необходимости немного интерактивности веб-страницам, так что большие скрипты в основном не требовались. Прошло несколько лет, и вот мы уже видим полномасштабные приложения, работающие в браузерах и содержащие массу кода на JavaScript; кроме того, язык стал использоваться и в других контекстах (например, Node.js).

Таким образом, в последние годы появился механизм деления JavaScript-программ на отдельные модули, которые можно импортировать по мере необходимости.

Полезно отделять файлы, которые являются модулями, от обычных JavaScript-файлов. Для этого можно использовать атрибут:

```
<script type="module">
```

# Экспорт функциональности модуля

Чтобы получить доступ к элементам функциональности модуля, нужно поместить перед ними инструкцию `export`, например:

```
export const name = "square";

export function draw(ctx, length, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, length, length);
}
```

Можно экспортировать `let`-, `const`-переменные, функции и классы. Они должны быть в верхней области видимости, например, нельзя использовать `export` внутри функции.

Более удобный способ экспорта всех необходимых элементов, — использовать одну конструкцию `export` в конце файла модуля, где указать переменные, функции, классы, которые нужно экспортировать, через запятую в фигурных скобках. Например:

```
export { name, draw };
```

## Импорт функциональности в скрипт

Чтобы иметь возможность использовать некоторые экспортированные части из модуля, их необходимо импортировать в свой скрипт. Для этого используется инструкция `import`:

```
import { name, draw } from "./modules/square.js";
```

В фигурных скобках указывается разделённый запятыми список элементов, которые нужно импортировать, за которым следует ключевое слово `from` и затем указывается путь к файлу модуля относительно корневого каталога. Если написание пути начинается с символа `(.)`, то это означает "текущую директорию".

После того, как вы импортировали функции в свой скрипт, вы можете использовать их так же, как если бы они были определены в этом же файле:

```
let square1 = draw(ctx, 50, 50, 100, "blue");
```

Инструкции `import` и `export` работают только внутри модулей, внутри обычных скриптов они работать не будут. Соответственно, для скрипта, в который импортируется модуль, тоже нужно добавить атрибут `type="module"` в `<script>`-элемент. Если вы упустите это, то Firefox, например, выдаст ошибку

"import declarations may only appear at top level of a module".



Если попытаться импортировать разные функции из разных модулей с одним и тем же именем, то возникнет ошибка конфликта имен. Чтобы её избежать можно импортировать функции каждого модуля внутри объекта модуля. Для этого используется следующая синтаксическая форма:

```
import * as Module from "./modules/module.js";
```

Эта конструкция берет все экспорты, доступные внутри module.js и делает их доступными в качестве свойств объекта Module, фактически давая ему собственное пространство имен. Теперь, например, можно вызывать функции модуля следующим образом:

```
Module.function1();  
Module.function2();
```

## Другие отличия модулей от обычных скриптов

- ▶ Если вы попытаетесь открыть файл HTML локально, вы столкнётесь с ошибками CORS из-за требований безопасности JavaScript-модулей. Функциональность модулей работает только на сервере.
- ▶ Модули используют строгий режим автоматически.
- ▶ Получить доступ к импортированным частям модуля можно только в скрипте, в который он импортирован, и, например, нельзя получить к нему доступ из консоли JavaScript. Поэтому, возможно, не получится использовать некоторые методы отладки.