```
// Си-подобные комментарии. Однострочные комментарии начинаются с двух символов слэш,
/* а многострочные комментарии начинаются с последовательности слэш-звёздочка
   и заканчиваются символами звёздочка-слэш */
// Инструкции могут заканчиваться точкой с запятой ;
doStuff();// ... но она необязательна
// 1. Числа, Строки и Операторы
// В JavaScript только один тип числа (64-bit IEEE 754 double). // Он имеет 52-битную мантиссу, которой достаточно для хранения целых чисел с точностью вплоть до 9✕10¹⁵.
3; // = 3
1.5; // = 1.5
// Некоторые простые арифметические операции работают так, как вы ожидаете.
1 + 1; // = 2
0.1 + 0.2; // = 0.30000000000000004 (а некоторые - нет)
8 - 1; // = 7     10 * 2; // = 20     35 / 5; // = 7
// Включая деление с остатком.  5 / 2; // = 2.5
// Побитовые операции также имеются; когда вы проводите такую операцию, ваше число с плавающей запятой переводится в целое со знаком длиной *до* 32 разрядов.
1 << 2; // = 4
// Есть три специальных значения, которые не являются реальными числами:
Infinity; // "бесконечность", например, результат деления на 0
-Infinity; // "минус бесконечность", результат деления отрицательного числа на 0
NaN; // "не число", например, результат деления 0/0
// Существует также логический тип.
true;  false;
// Строки создаются при помощи двойных или одинарных кавычек. 'абв'; "Привет, мир!";
// Для логического отрицания используется восклицательный знак.
!true; // = false
!false; // = true
// Строгое равенство ===
1 === 1; // = true    2 === 1; // = false
// Строгое неравенство !==
1 !== 1; // = false    2 !== 1; // = true
// Другие операторы сравнения
1 < 10; // = true  1 > 10; // = false 2 <= 2; // = true
2 >= 2; // = true
// Строки объединяются при помощи оператора +
"привет, " + "мир!"; // = "Привет, мир!"
// и сравниваются при помощи < и >
"a" < "b"; // = true
// Проверка равенства с приведением типов осуществляется двойным символом равно
"5" == 5; // = true   null == undefined; // = true
// ...если только не использовать ===
"5" === 5; // = false  null === undefined; // = false
// ...приведение типов может привести к странному поведению… 13 + !0; // 14   "13" + !0; // '13true'
// Вы можете получить доступ к любому символу строки, используя метод charAt
"Это строка".charAt(0);  // = 'Э'
// ...или используйте метод substring, чтобы получить более крупные части
"Привет, мир".substring(0, 6); // = "Привет"
// length - это свойство, для его получения не нужно использовать () "Привет".length; // = 6
// Также есть null и undefined
null; // Намеренное отсутствие значения
undefined; // используется для обозначения переменных, не имеющих присвоенного значения (хотя непосредственно undefined является значением)
// false, null, undefined, NaN, 0 и "" — это ложь; всё остальное - истина.
// Следует отметить, что 0 — это ложь, а "0" — истина, несмотря на то, что 0 == "0".
// 2. Переменные, Массивы и Объекты
// Переменные объявляются при помощи ключевого слова var. JavaScript — язык с динамической типизацией, поэтому не нужно явно указывать тип. Для присваивания значения переменной используется символ =      var someVar = 5;
// если вы пропустите слово var, вы не получите сообщение об ошибке, …    someOtherVar = 10;
// ...но ваша переменная будет создана в глобальном контексте, а не в текущем,где вы ее объявили.
// Переменным, объявленным без присвоения, устанавливается значение undefined.
var someThirdVar; // = undefined
// У математических операций есть сокращённые формы:
someVar += 5; // как someVar = someVar + 5; someVar теперь имеет значение 10
someVar *= 10; // теперь someVar имеет значение 100
// Ещё более краткая запись увеличения и уменьшения на единицу:
someVar++; // теперь someVar имеет значение 101
someVar--; // вернулись к 100
// Массивы — это нумерованные списки, содержащие значения любого типа.
var myArray = ["Привет", 45, true];
// Их элементы могут быть доступны при помощи синтаксиса с квадратными скобками.
// Индексы массивов начинаются с нуля.
myArray[1]; // = 45
```

```
// Массивы можно изменять, как и их длину,
myArray.push("Мир");  myArray.length; // = 4
// добавлять и редактировать определённый элемент
myArray[3] = "Привет";
// Объекты в JavaScript похожи на словари или
ассоциативные массивы в других языках:
неупорядоченный набор пар ключ-значение.
var myObj = {key1: "Привет", key2: "Мир"};
// Ключи — это строки, но кавычки необязательны,
если строка удовлетворяет ограничениям для имён
переменных. Значения могут быть любых типов.
var myObj = {myKey: "myValue", "my other key": 4};
// Атрибуты объектов можно также получить,
используя квадратные скобки
myObj["my other key"]; // = 4
// или через точку, при условии, что ключ является
допустимым идентификатором.
myObj.myKey; // = "myValue"
// Объекты изменяемы; можно изменять значения и
добавлять новые ключи. myObj.myThirdKey = true;
// Если вы попытаетесь получить доступ к
несуществующему значению, вы получите undefined.
myObj.myFourthKey; // = undefined
// 3. Логика и управляющие конструкции.
// Синтаксис для этого раздела почти такой же, как в
Java. Условная конструкция работает, как и следовало
ожидать,
var count = 1;
if (count == 3) { // выполняется, если count равен 3
} else if (count == 4) { //выполняется, если count = 4
} else { // выполняется, если не 3 и не 4
}   // ...как и цикл while.
while (true){  // Бесконечный цикл!  }
// Цикл do-while такой же, как while, но он всегда
выполняется хотя бы раз.
var input
do {
   input = getInput();
} while (!isValid(input))
// цикл for такой же, как в C и Java: инициализация;
условие; шаг.
for (var i = 0; i < 5; i++) { // выполнится 5 раз }
// && — это логическое И, || — это логическое ИЛИ
if (house.size == "big" && house.color == "blue") {
   house.contains = "bear"; }
if (color == "red" || color == "blue") {
   // цвет красный или синий }
// && и || используют сокращённое вычисление, что
полезно для задания значений по умолчанию.
var name = otherName || "default";
// Оператор switch выполняет проверку на равенство
при помощи ===
// используйте break, чтобы прервать выполнение
после каждого case, или выполнение пойдёт далее
даже после правильного варианта.
grade = 4;
switch (grade) {
  case 5:
    console.log("Отлично");
    break;
  case 4:
    console.log("Хорошо");
    break;
  case 3:
    console.log("Можете и лучше");
    break;
  default:
    console.log("Ой-вей!");
    break;
}
// 4. Функции, Область видимости и Замыкания
// Функции в  JavaScript объявляются при помощи
ключевого слова function.
function myFunction(thing) {
   return thing.toUpperCase();
}
myFunction("foo"); // = "FOO"
// Обратите внимание, что значение, которое будет
возвращено, должно начинаться на той же строке, что
и ключевое слово return, в противном случае вы
будете
всегда возвращать undefined по причине
автоматической вставки точки с запятой. Следите за
этим при использовании стиля форматирования
Allman.
function myFunction()
{
   return // <- здесь точка с запятой вставится
автоматически
   {  thisIsAn: 'object literal'}
}
myFunction(); // = undefined
// В JavaScript функции — это объекты первого класса,
поэтому они могут быть присвоены различным
именам переменных и передаваться другим функциям
в качестве аргументов, например, когда назначается
обработчик события:
function myFunction() {  // этот код будет вызван через
5 секунд }
setTimeout(myFunction, 5000);
// Примечание: setTimeout не является частью языка,
но реализован в браузерах и Node.js
// Функции не обязательно должны иметь имя при
объявлении — вы можете написать анонимное
определение функции непосредственно в аргументе
```

```
другой функции.
setTimeout(function() {этот код будет вызван через 5 секунд }, 5000);
// В JavaScript есть область видимости; функции имеют свою область видимости, а другие блоки — нет.
if (true) {   var i = 5; }
i; // = 5, а не undefined, как ожидалось бы в языках с блочной областью видимости. Это привело к общепринятому шаблону "самозапускающихся анонимных функций", которые препятствуют проникновению переменных в глобальную область видимости
(function() {
   var temporary = 5; // Мы можем получить доступ к глобальной области для записи в «глобальный объект», который в веб-браузере всегда window. Глобальный объект может иметь другое имя в таких платформах, как Node.js
   window.permanent = 10;   })();
temporary; // вызовет сообщение об ошибке с типом ReferenceError
permanent; // = 10
// Одной из самых мощных возможностей JavaScript являются замыкания. Если функция определена внутри другой функции, то внутренняя функция имеет доступ к переменным внешней функции даже после того, как контекст выполнения выйдет из внешней функции.
function sayHelloInFiveSeconds(name) {
   var prompt = "Привет, " + name + "!";
   // Внутренние функции по умолчанию помещаются в локальную область видимости, как если бы они были объявлены с помощью var.
   function inner() { alert(prompt); }
   setTimeout(inner, 5000);  setTimeout асинхронна, поэтому функция sayHelloInFiveSeconds сразу выйдет, после чего setTimeout вызовет функцию inner. Однако поскольку функция inner «замкнута» вокруг sayHelloInFiveSeconds, она по-прежнему имеет доступ к переменной prompt на то время, когда она наконец будет вызвана. }
sayHelloInFiveSeconds("Адам"); // Через 5 с откроется окно «Привет, Адам!»
// 5. Подробнее об объектах; Конструкторы и Прототипы
// Объекты могут содержать в себе функции.
var myObj = {
   myFunc: function() { return "Привет, мир!";   }
};
myObj.myFunc(); // = "Привет, мир!"
// Когда вызываются функции, прикреплённые к объекту, они могут получить доступ к этому объекту с помощью ключевого слова this.

myObj = {  myString: "Привет, мир!",
   myFunc: function() { return this.myString;}
};
myObj.myFunc(); // = "Привет, мир!"
// Значение this зависит от того, как функция вызывается, а не от того, где она определена. Таким образом, наша функция не работает, если она вызывается не в контексте объекта.
var myFunc = myObj.myFunc;
myFunc(); // = undefined
// И наоборот, функция может быть присвоена объекту и получать доступ к нему через this, даже если она не была прикреплена к нему при объявлении.
var myOtherFunc = function() {
   return this.myString.toUpperCase(); }
myObj.myOtherFunc = myOtherFunc;
myObj.myOtherFunc(); // = "ПРИВЕТ, МИР!"
// Мы можем также указать контекст для выполнения функции при её вызове, используя call или apply.
var anotherFunc = function(s) {
   return this.myString + s; }
anotherFunc.call(myObj, " И привет, Луна!"); // = "Привет, мир! И привет, Луна!"
// Функция apply почти такая же, но принимает в качестве списка аргументов массив.
anotherFunc.apply(myObj, [" И привет, Солнце!"]); // = "Привет, мир! И привет, Солнце!" Это полезно при работе с функцией, которая принимает последовательность аргументов, а вы хотите передать массив.
Math.min(42, 6, 27); // = 6
Math.min([42, 6, 27]); // = NaN (Ой-ой!)
Math.min.apply(Math, [42, 6, 27]); // = 6
Но call и apply — только временные. Когда мы хотим связать функцию с объектом, мы можем использовать bind.
var boundFunc = anotherFunc.bind(myObj);
boundFunc(" И привет, Сатурн!"); // = "Привет, мир! И привет, Сатурн!" Bind также может использоваться для частичного применения (каррирования) функции
var product = function(a, b) { return a * b; }
var doubler = product.bind(this, 2);
doubler(8); // = 16
// Когда вы вызываете функцию с помощью ключевого слова new, создается новый объект, доступный функции при помощи this. Такие функции называют конструкторами.
var MyConstructor = function() { this.myNumber = 5;}
myNewObj = new MyConstructor(); // = {myNumber: 5}
myNewObj.myNumber; // = 5
```

```
// У каждого объекта в JavaScript есть прототип. Когда вы хотите получить доступ к свойству объекта, которое не существует в этом объекте, интерпретатор будет искать это свойство в прототипе.
Некоторые реализации языка позволяют получить доступ к прототипу объекта через «магическое» свойство __proto__. Несмотря на то, что это может быть полезно для понимания прототипов, это не часть стандарта; мы увидим стандартные способы использования прототипов позже.
var myObj = { myString: "Привет, мир!"};
var myPrototype = { meaningOfLife: 42,
  myFunc: function() {
    return this.myString.toLowerCase() } };
myObj.__proto__ = myPrototype;
myObj.meaningOfLife; // = 42
// Для функций это тоже работает.
myObj.myFunc(); // = "Привет, мир!"
// Если интерпретатор не найдёт свойство в прототипе, то продожит поиск в прототипе прототипа и так далее.
myPrototype.__proto__ = { myBoolean: true };
myObj.myBoolean; // = true
// Здесь не участвует копирование; каждый объект хранит ссылку на свой прототип. Это означает, что мы можем изменить прототип, и наши изменения будут отражены везде.
myPrototype.meaningOfLife = 43;
myObj.meaningOfLife; // = 43
// Мы упомянули, что свойство __proto__ нестандартно, и нет никакого стандартного способа, чтобы изменить прототип существующего объекта. Однако есть два способа создать новый объект с заданным прототипом. Первый способ — это Object.create, который появился в JavaScript недавно, а потому доступен ещё не во всех реализациях языка.
var myObj = Object.create(myPrototype);
myObj.meaningOfLife; // = 43
// Второй способ, который работает везде, имеет дело с конструкторами. У конструкторов есть свойство с именем prototype. Это *не* прототип функции-конструктора; напротив, это прототип для новых объектов, которые будут созданы с помощью этого конструктора и ключевого слова new.
MyConstructor.prototype = { myNumber: 5,
  getMyNumber: function() { return this.myNumber; }
};
var myNewObj2 = new MyConstructor();
myNewObj2.getMyNumber(); // = 5
myNewObj2.myNumber = 6
myNewObj2.getMyNumber(); // = 6
У встроенных типов, таких, как строки и числа, также есть конструкторы, которые создают эквивалентные объекты-обёртки.
var myNumber = 12;
var myNumberObj = new Number(12);
myNumber == myNumberObj; // = true

// За исключением того, что они не в точности равны.
typeof myNumber; // = 'number'
typeof myNumberObj; // = 'object'
myNumber === myNumberObj; // = false
if (0) {
  // Этот код не выполнится, потому что 0 - это ложь.
}

// Впрочем, объекты-обёртки и встроенные типы имеют общие прототипы,
// поэтому вы можете расширить функционал строк, например:
String.prototype.firstCharacter = function() {
  return this.charAt(0);
}
"abc".firstCharacter(); // = "a"

// Это часто используется в т.н. полифилах, которые реализуют новые возможности
// JavaScript в старой реализации языка, так что они могут быть использованы в
// старых средах, таких, как устаревшие браузеры.

// Например, мы упомянули, что Object.create доступен не во всех реализациях, но
// мы сможем использовать его с помощью такого полифила:
if (Object.create === undefined) { // не перезаписываем метод, если он существует
  Object.create = function(proto) {
    // создаём временный конструктор с правильным прототипом
    var Constructor = function(){};
    Constructor.prototype = proto;
    // затем используем его для создания нового,
    // правильно прототипированного объекта
    return new Constructor();
  }
}
```