```
// Single-line comments start with two slashes.
/* Multiline comments start with slash-star,
   and end with star-slash */

// Statements can be terminated by ;
doStuff();

// ... but they don't have to be, as semicolons are automatically inserted
// wherever there's a newline, except in certain cases.
doStuff()

// Because those cases can cause unexpected results, we'll keep on using
// semicolons in this guide.

///////////////////////////////////
// 1. Numbers, Strings and Operators

// JavaScript has one number type (which is a 64-bit IEEE 754 double).
// Doubles have a 52-bit mantissa, which is enough to store integers
// up to about 9âM-^\M-^U10Â¹âM-^Aµ precisely.
3; // = 3
1.5; // = 1.5

// Some basic arithmetic works as you'd expect.
1 + 1; // = 2
0.1 + 0.2; // = 0.30000000000000004
8 - 1; // = 7
10 * 2; // = 20
35 / 5; // = 7

// Including uneven division.
5 / 2; // = 2.5

// And modulo division.
10 % 2; // = 0
30 % 4; // = 2
18.5 % 7; // = 4.5

// Bitwise operations also work; when you perform a bitwise operation your float
// is converted to a signed int *up to* 32 bits.
1 << 2; // = 4

// Precedence is enforced with parentheses.
(1 + 3) * 2; // = 8

// There are three special not-a-real-number values:
Infinity; // result of e.g. 1/0
-Infinity; // result of e.g. -1/0
NaN; // result of e.g. 0/0, stands for 'Not a Number'

// There's also a boolean type.
true;
false;

// Strings are created with ' or ".
'abc';
"Hello, world";

// Negation uses the ! symbol
!true; // = false
!false; // = true

// Equality is ===
1 === 1; // = true
2 === 1; // = false

// Inequality is !==
1 !== 1; // = false
2 !== 1; // = true
```



```
// More comparisons
1 < 10; // = true
1 > 10; // = false
2 <= 2; // = true
2 >= 2; // = true

// Strings are concatenated with +
"Hello " + "world!"; // = "Hello world!"

// ... which works with more than just strings
"1, 2, " + 3; // = "1, 2, 3"
"Hello " + ["world", "!"]; // = "Hello world,!"

// and are compared with < and >
"a" < "b"; // = true

// Type coercion is performed for comparisons with double equals...
"5" == 5; // = true
null == undefined; // = true

// ...unless you use ===
"5" === 5; // = false
null === undefined; // = false

// ...which can result in some weird behaviour...
13 + !0; // 14
"13" + !0; // '13true'

// You can access characters in a string with `charAt`
"This is a string".charAt(0);  // = 'T'

// ...or use `substring` to get larger pieces.
"Hello world".substring(0, 5); // = "Hello"

// `length` is a property, so don't use ().
"Hello".length; // = 5

// There's also `null` and `undefined`.
null;      // used to indicate a deliberate non-value
undefined; // used to indicate a value is not currently present (although
           // `undefined` is actually a value itself)

// false, null, undefined, NaN, 0 and "" are falsy; everything else is truthy.
// Note that 0 is falsy and "0" is truthy, even though 0 == "0".

///////////////////////////////////
// 2. Variables, Arrays and Objects

// Variables are declared with the `var` keyword. JavaScript is dynamically
// typed, so you don't need to specify type. Assignment uses a single `=`
// character.
var someVar = 5;

// If you leave the var keyword off, you won't get an error...
someOtherVar = 10;

// ...but your variable will be created in the global scope, not in the scope
// you defined it in.

// Variables declared without being assigned to are set to undefined.
var someThirdVar; // = undefined

// If you want to declare a couple of variables, then you could use a comma
// separator
var someFourthVar = 2, someFifthVar = 4;

// There's shorthand for performing math operations on variables:
someVar += 5; // equivalent to someVar = someVar + 5; someVar is 10 now
```

```
someVar *= 10; // now someVar is 100

// and an even-shorter-hand for adding or subtracting 1
someVar++; // now someVar is 101
someVar--; // back to 100

// Arrays are ordered lists of values, of any type.
var myArray = ["Hello", 45, true];

// Their members can be accessed using the square-brackets subscript syntax.
// Array indices start at zero.
myArray[1]; // = 45

// Arrays are mutable and of variable length.
myArray.push("World");
myArray.length; // = 4

// Add/Modify at specific index
myArray[3] = "Hello";

// Add and remove element from front or back end of an array
myArray.unshift(3); // Add as the first element
someVar = myArray.shift(); // Remove first element and return it
myArray.push(3); // Add as the last element
someVar = myArray.pop(); // Remove last element and return it

// Join all elements of an array with semicolon
var myArray0 = [32,false,"js",12,56,90];
myArray0.join(";"); // = "32;false;js;12;56;90"

// Get subarray of elements from index 1 (include) to 4 (exclude)
myArray0.slice(1,4); // = [false,"js",12]

// Remove 4 elements starting from index 2, and insert there strings
// "hi","wr" and "ld"; return removed subarray
myArray0.splice(2,4,"hi","wr","ld"); // = ["js",12,56,90]
// myArray0 === [32,false,"hi","wr","ld"]

// JavaScript's objects are equivalent to "dictionaries" or "maps" in other
// languages: an unordered collection of key-value pairs.
var myObj = {key1: "Hello", key2: "World"};

// Keys are strings, but quotes aren't required if they're a valid
// JavaScript identifier. Values can be any type.
var myObj = {myKey: "myValue", "my other key": 4};

// Object attributes can also be accessed using the subscript syntax,
myObj["my other key"]; // = 4

// ... or using the dot syntax, provided the key is a valid identifier.
myObj.myKey; // = "myValue"

// Objects are mutable; values can be changed and new keys added.
myObj.myThirdKey = true;

// If you try to access a value that's not yet set, you'll get undefined.
myObj.myFourthKey; // = undefined

////////////////////////////////
// 3. Logic and Control Structures

// The `if` structure works as you'd expect.
var count = 1;
if (count == 3){
    // evaluated if count is 3
} else if (count == 4){
    // evaluated if count is 4
} else {
    // evaluated if it's not either 3 or 4
```



```
}

// As does `while`.
while (true){
    // An infinite loop!
}

// Do-while loops are like while loops, except they always run at least once.
var input;
do {
    input = getInput();
} while (!isValid(input));

// The `for` loop is the same as C and Java:
// initialization; continue condition; iteration.
for (var i = 0; i < 5; i++){
    // will run 5 times
}

// Breaking out of labeled loops is similar to Java
outer:
for (var i = 0; i < 10; i++) {
    for (var j = 0; j < 10; j++) {
        if (i == 5 && j ==5) {
            break outer;
            // breaks out of outer loop instead of only the inner one
        }
    }
}

// The for/in statement allows iteration over properties of an object.
var description = "";
var person = {fname:"Paul", lname:"Ken", age:18};
for (var x in person){
    description += person[x] + " ";
} // description = 'Paul Ken 18 '

// The for/of statement allows iteration over iterable objects (including the bu
ilt-in String,
// Array, e.g. the Array-like arguments or NodeList objects, TypedArray, Map and
 Set,
// and user-defined iterables).
var myPets = "";
var pets = ["cat", "dog", "hamster", "hedgehog"];
for (var pet of pets){
    myPets += pet + " ";
} // myPets = 'cat dog hamster hedgehog '

// && is logical and, || is logical or
if (house.size == "big" && house.colour == "blue"){
    house.contains = "bear";
}
if (colour == "red" || colour == "blue"){
    // colour is either red or blue
}

// && and || "short circuit", which is useful for setting default values.
var name = otherName || "default";

// The `switch` statement checks for equality with `===`.
// Use 'break' after each case
// or the cases after the correct one will be executed too.
grade = 'B';
switch (grade) {
  case 'A':
    console.log("Great job");
    break;
  case 'B':
    console.log("OK job");
```

```
    break;
  case 'C':
    console.log("You can do better");
    break;
  default:
    console.log("Oy vey");
    break;
}


////////////////////////////////
// 4. Functions, Scope and Closures

// JavaScript functions are declared with the `function` keyword.
function myFunction(thing){
    return thing.toUpperCase();
}
myFunction("foo"); // = "FOO"

// Note that the value to be returned must start on the same line as the
// `return` keyword, otherwise you'll always return `undefined` due to
// automatic semicolon insertion. Watch out for this when using Allman style.
function myFunction(){
    return // <- semicolon automatically inserted here
    {thisIsAn: 'object literal'};
}
myFunction(); // = undefined

// JavaScript functions are first class objects, so they can be reassigned to
// different variable names and passed to other functions as arguments - for
// example, when supplying an event handler:
function myFunction(){
    // this code will be called in 5 seconds' time
}
setTimeout(myFunction, 5000);
// Note: setTimeout isn't part of the JS language, but is provided by browsers
// and Node.js.

// Another function provided by browsers is setInterval
function myFunction(){
    // this code will be called every 5 seconds
}
setInterval(myFunction, 5000);

// Function objects don't even have to be declared with a name - you can write
// an anonymous function definition directly into the arguments of another.
setTimeout(function(){
    // this code will be called in 5 seconds' time
}, 5000);

// JavaScript has function scope; functions get their own scope but other blocks
// do not.
if (true){
    var i = 5;
}
i; // = 5 - not undefined as you'd expect in a block-scoped language

// This has led to a common pattern of "immediately-executing anonymous
// functions", which prevent temporary variables from leaking into the global
// scope.
(function(){
    var temporary = 5;
    // We can access the global scope by assigning to the "global object", which
    // in a web browser is always `window`. The global object may have a
    // different name in non-browser environments such as Node.js.
    window.permanent = 10;
})();
temporary; // raises ReferenceError
permanent; // = 10
```



```
// One of JavaScript's most powerful features is closures. If a function is
// defined inside another function, the inner function has access to all the
// outer function's variables, even after the outer function exits.
function sayHelloInFiveSeconds(name){
    var prompt = "Hello, " + name + "!";
    // Inner functions are put in the local scope by default, as if they were
    // declared with `var`.
    function inner(){
        alert(prompt);
    }
    setTimeout(inner, 5000);
    // setTimeout is asynchronous, so the sayHelloInFiveSeconds function will
    // exit immediately, and setTimeout will call inner afterwards. However,
    // because inner is "closed over" sayHelloInFiveSeconds, inner still has
    // access to the `prompt` variable when it is finally called.
}
sayHelloInFiveSeconds("Adam"); // will open a popup with "Hello, Adam!" in 5s

////////////////////////////////
// 5. More about Objects; Constructors and Prototypes

// Objects can contain functions.
var myObj = {
    myFunc: function(){
        return "Hello world!";
    }
};
myObj.myFunc(); // = "Hello world!"

// When functions attached to an object are called, they can access the object
// they're attached to using the `this` keyword.
myObj = {
    myString: "Hello world!",
    myFunc: function(){
        return this.myString;
    }
};
myObj.myFunc(); // = "Hello world!"

// What this is set to has to do with how the function is called, not where
// it's defined. So, our function doesn't work if it isn't called in the
// context of the object.
var myFunc = myObj.myFunc;
myFunc(); // = undefined

// Inversely, a function can be assigned to the object and gain access to it
// through `this`, even if it wasn't attached when it was defined.
var myOtherFunc = function(){
    return this.myString.toUpperCase();
};
myObj.myOtherFunc = myOtherFunc;
myObj.myOtherFunc(); // = "HELLO WORLD!"

// We can also specify a context for a function to execute in when we invoke it
// using `call` or `apply`.

var anotherFunc = function(s){
    return this.myString + s;
};
anotherFunc.call(myObj, " And Hello Moon!"); // = "Hello World! And Hello Moon!"

// The `apply` function is nearly identical, but takes an array for an argument
// list.

anotherFunc.apply(myObj, [" And Hello Sun!"]); // = "Hello World! And Hello Sun!
"

// This is useful when working with a function that accepts a sequence of
```

```
// arguments and you want to pass an array.

Math.min(42, 6, 27); // = 6
Math.min([42, 6, 27]); // = NaN (uh-oh!)
Math.min.apply(Math, [42, 6, 27]); // = 6

// But, `call` and `apply` are only temporary. When we want it to stick, we can
// use `bind`.

var boundFunc = anotherFunc.bind(myObj);
boundFunc(" And Hello Saturn!"); // = "Hello World! And Hello Saturn!"

// `bind` can also be used to partially apply (curry) a function.

var product = function(a, b){ return a * b; };
var doubler = product.bind(this, 2);
doubler(8); // = 16

// When you call a function with the `new` keyword, a new object is created, and
// made available to the function via the `this` keyword. Functions designed to
be
// called like that are called constructors.

var MyConstructor = function(){
    this.myNumber = 5;
};
myNewObj = new MyConstructor(); // = {myNumber: 5}
myNewObj.myNumber; // = 5

// Unlike most other popular object-oriented languages, JavaScript has no
// concept of 'instances' created from 'class' blueprints; instead, JavaScript
// combines instantiation and inheritance into a single concept: a 'prototype'.

// Every JavaScript object has a 'prototype'. When you go to access a property
// on an object that doesn't exist on the actual object, the interpreter will
// look at its prototype.

// Some JS implementations let you access an object's prototype on the magic
// property `__proto__`. While this is useful for explaining prototypes it's not
// part of the standard; we'll get to standard ways of using prototypes later.
var myObj = {
    myString: "Hello world!"
};
var myPrototype = {
    meaningOfLife: 42,
    myFunc: function(){
        return this.myString.toLowerCase();
    }
};

myObj.__proto__ = myPrototype;
myObj.meaningOfLife; // = 42

// This works for functions, too.
myObj.myFunc(); // = "hello world!"

// Of course, if your property isn't on your prototype, the prototype's
// prototype is searched, and so on.
myPrototype.__proto__ = {
    myBoolean: true
};
myObj.myBoolean; // = true

// There's no copying involved here; each object stores a reference to its
// prototype. This means we can alter the prototype and our changes will be
// reflected everywhere.
myPrototype.meaningOfLife = 43;
myObj.meaningOfLife; // = 43
```



```
// The for/in statement allows iteration over properties of an object,
// walking up the prototype chain until it sees a null prototype.
for (var x in myObj){
    console.log(myObj[x]);
}
///prints:
// Hello world!
// 43
// [Function: myFunc]
// true

// To only consider properties attached to the object itself
// and not its prototypes, use the `hasOwnProperty()` check.
for (var x in myObj){
    if (myObj.hasOwnProperty(x)){
        console.log(myObj[x]);
    }
}
///prints:
// Hello world!

// We mentioned that `__proto__` was non-standard, and there's no standard way t
o
// change the prototype of an existing object. However, there are two ways to
// create a new object with a given prototype.

// The first is Object.create, which is a recent addition to JS, and therefore
// not available in all implementations yet.
var myObj = Object.create(myPrototype);
myObj.meaningOfLife; // = 43

// The second way, which works anywhere, has to do with constructors.
// Constructors have a property called prototype. This is *not* the prototype of
// the constructor function itself; instead, it's the prototype that new objects
// are given when they're created with that constructor and the new keyword.
MyConstructor.prototype = {
    myNumber: 5,
    getMyNumber: function(){
        return this.myNumber;
    }
};
var myNewObj2 = new MyConstructor();
myNewObj2.getMyNumber(); // = 5
myNewObj2.myNumber = 6;
myNewObj2.getMyNumber(); // = 6

// Built-in types like strings and numbers also have constructors that create
// equivalent wrapper objects.
var myNumber = 12;
var myNumberObj = new Number(12);
myNumber == myNumberObj; // = true

// Except, they aren't exactly equivalent.
typeof myNumber; // = 'number'
typeof myNumberObj; // = 'object'
myNumber === myNumberObj; // = false
if (0){
    // This code won't execute, because 0 is falsy.
}
if (new Number(0)){
   // This code will execute, because wrapped numbers are objects, and objects
   // are always truthy.
}

// However, the wrapper objects and the regular builtins share a prototype, so
// you can actually add functionality to a string, for instance.
String.prototype.firstCharacter = function(){
    return this.charAt(0);
};
```

```
"abc".firstCharacter(); // = "a"

// This fact is often used in "polyfilling", which is implementing newer
// features of JavaScript in an older subset of JavaScript, so that they can be
// used in older environments such as outdated browsers.

// For instance, we mentioned that Object.create isn't yet available in all
// implementations, but we can still use it with this polyfill:
if (Object.create === undefined){ // don't overwrite it if it exists
    Object.create = function(proto){
        // make a temporary constructor with the right prototype
        var Constructor = function(){};
        Constructor.prototype = proto;
        // then use it to create a new, appropriately-prototyped object
        return new Constructor();
    };
}

// ES6 Additions

// The "let" keyword allows you to define variables in a lexical scope,
// as opposed to a block scope like the var keyword does.
let name = "Billy";

// Variables defined with let can be reassigned new values.
name = "William";

// The "const" keyword allows you to define a variable in a lexical scope
// like with let, but you cannot reassign the value once one has been assigned.

const pi = 3.14;

pi = 4.13; // You cannot do this.

// There is a new syntax for functions in ES6 known as "lambda syntax".
// This allows functions to be defined in a lexical scope like with variables
// defined by const and let.

const isEven = (number) => {
    return number % 2 === 0;
};

isEven(7); // false

// The "equivalent" of this function in the traditional syntax would look like t
his:

function isEven(number) {
    return number % 2 === 0;
};

// I put the word "equivalent" in double quotes because a function defined
// using the lambda syntax cannnot be called before the definition.
// The following is an example of invalid usage:

add(1, 8);

const add = (firstNumber, secondNumber) => {
    return firstNumber + secondNumber;
};
```