

# Chapter 11. The JavaScript Standard Library

Some datatypes, such as numbers and strings ([Chapter 3](#)), objects ([Chapter 6](#)), and arrays ([Chapter 7](#)) are so fundamental to JavaScript that we can consider them to be part of the language itself. This chapter covers other important but less fundamental APIs that can be thought of as defining the “standard library” for JavaScript: these are useful classes and functions that are built in to JavaScript and available to all JavaScript programs in both web browsers and in Node.<sup>1</sup>

The sections of this chapter are independent of one another, and you can read them in any order. They cover:

- The Set and Map classes for representing sets of values and mappings from one set of values to another set of values.
- Array-like objects known as TypedArrays that represent arrays of binary data, along with a related class for extracting values from non-array binary data.
- Regular expressions and the RegExp class, which define textual patterns and are useful for text processing. This section also covers regular expression syntax in detail.
- The Date class for representing and manipulating dates and times.
- The Error class and its various subclasses, instances of which are thrown when errors occur in JavaScript programs.
- The JSON object, whose methods support serialization and deserialization of JavaScript data structures composed of objects, arrays, strings, numbers, and booleans.
- The Intl object and the classes it defines that can help you localize your JavaScript programs.
- The Console object, whose methods output strings in ways that are particularly useful for debugging programs and logging the behavior of those programs.
- The URL class, which simplifies the task of parsing and manipulating URLs. This section also covers global functions for encoding and decoding URLs and their component parts.
- `setTimeout()` and related functions for specifying code to be executed after a specified interval of time has elapsed.

Some of the sections in this chapter—notably, the sections on typed arrays and regular expressions—are quite long because there is significant background information you need to understand before you can use those types effectively.

Many of the other sections, however, are short: they simply introduce a new API and show some examples of its use.

## 11.1 Sets and Maps

JavaScript's Object type is a versatile data structure that can be used to map strings (the object's property names) to arbitrary values. And when the value being mapped to is something fixed like `true`, then the object is effectively a set of strings.

Objects are actually used as maps and sets fairly routinely in JavaScript programming, but this is limited by the restriction to strings and complicated by the fact that objects normally inherit properties with names like "toString", which are not typically intended to be part of the map or set.

For this reason, ES6 introduces true Set and Map classes, which we'll cover in the sub-sections that follow.

### 11.1.1 The Set Class

A set is a collection of values, like an array is. Unlike arrays, however, sets are not ordered or indexed, and they do not allow duplicates: a value is either a member of a set or it is not a member; it is not possible to ask how many times a value appears in a set.

Create a Set object with the `Set()` constructor:

```
let s = new Set();           // A new, empty set
let t = new Set([1, s]);    // A new set with two members
```

The argument to the `Set()` constructor need not be an array: any iterable object (including other Set objects) is allowed:

```
let t = new Set(s);          // A new set that copies the elements of s.
let unique = new Set("Mississippi"); // 4 elements: "M", "i", "s", and "p"
```

The `size` property of a set is like the `length` property of an array: it tells you how many values the set contains:

```
unique.size      // => 4
```

Sets don't need to be initialized when you create them. You can add and remove elements at any time with `add()`, `delete()`, and `clear()`. Remember that sets cannot contain duplicates, so adding a value to a set when it already contains that value has no effect:

```
let s = new Set(); // Start empty
s.size           // => 0
s.add(1);        // Add a number
s.size           // => 1; now the set has one member
s.add(1);        // Add the same number again
s.size           // => 1; the size does not change
s.add(true);     // Add another value; note that it is fine to mix types
s.size           // => 2
s.add([1,2,3]); // Add an array value
s.size           // => 3; the array was added, not its elements
s.delete(1)      // => true: successfully deleted element 1
s.size           // => 2: the size is back down to 2
s.delete("test") // => false: "test" was not a member, deletion failed
s.delete(true)   // => true: delete succeeded
```

```
s.delete([1,2,3])    // => false: the array in the set is different
s.size              // => 1: there is still that one array in the set
s.clear();          // Remove everything from the set
s.size              // => 0
```

There are a few important points to note about this code:

- The `add()` method takes a single argument; if you pass an array, it adds the array itself to the set, not the individual array elements. `add()` always returns the set it is invoked on, however, so if you want to add multiple values to a set, you can use chained method calls like  
`s.add('a').add('b').add('c');`.
- The `delete()` method also only deletes a single set element at a time. Unlike `add()`, however, `delete()` returns a boolean value. If the value you specify was actually a member of the set, then `delete()` removes it and returns `true`. Otherwise, it does nothing and returns `false`.
- Finally, it is very important to understand that set membership is based on strict equality checks, like the `==` operator performs. A set can contain both the number `1` and the string `"1"`, because it considers them to be distinct values. When the values are objects (or arrays or functions), they are also compared as if with `==`. This is why we were unable to delete the array element from the set in this code. We added an array to the set and then tried to remove that array by passing a *different* array (albeit with the same elements) to the `delete()` method. In order for this to work, we would have had to pass a reference to exactly the same array.

#### Note

Python programmers take note: this is a significant difference between JavaScript and Python sets. Python sets compare members for equality, not identity, but the trade-off is that Python sets only allow immutable members, like tuples, and do not allow lists and dicts to be added to sets.

In practice, the most important thing we do with sets is not to add and remove elements from them, but to check to see whether a specified value is a member of the set. We do this with the `has()` method:

```
let oneDigitPrimes = new Set([2,3,5,7]);
oneDigitPrimes.has(2)    // => true: 2 is a one-digit prime number
oneDigitPrimes.has(3)    // => true: so is 3
oneDigitPrimes.has(4)    // => false: 4 is not a prime
oneDigitPrimes.has("5") // => false: "5" is not even a number
```

The most important thing to understand about sets is that they are optimized for membership testing, and no matter how many members the set has, the `has()` method will be very fast. The `includes()` method of an array also performs membership testing, but the time it takes is proportional to the size of the array, and using an array as a set can be much, much slower than using a real Set object.

The Set class is iterable, which means that you can use a `for/of` loop to enumerate all of the elements of a set:

```
let sum = 0;
for(let p of oneDigitPrimes) { // Loop through the one-digit primes
    sum += p;                // and add them up
}
sum                         // => 17: 2 + 3 + 5 + 7
```

Because Set objects are iterable, you can convert them to arrays and argument lists with the ... spread operator:

```
[...oneDigitPrimes]           // => [2,3,5,7]: the set converted to an Array
Math.max(...oneDigitPrimes) // => 7: set elements passed as function arguments
```

Sets are often described as “unordered collections.” This isn’t exactly true for the JavaScript Set class, however. A JavaScript set is unindexed: you can’t ask for the first or third element of a set the way you can with an array. But the JavaScript Set class always remembers the order that elements were inserted in, and it always uses this order when you iterate a set: the first element inserted will be the first one iterated (assuming you haven’t deleted it first), and the most recently inserted element will be the last one iterated.<sup>2</sup>

In addition to being iterable, the Set class also implements a `forEach()` method that is similar to the array method of the same name:

```
let product = 1;
oneDigitPrimes.forEach(n => { product *= n; });
product // => 210: 2 * 3 * 5 * 7
```

The `forEach()` of an array passes array indexes as the second argument to the function you specify. Sets don’t have indexes, so the Set class’s version of this method simply passes the element value as both the first and second argument.

## 11.1.2 The Map Class

A Map object represents a set of values known as *keys*, where each key has another value associated with (or “mapped to”) it. In a sense, a map is like an array, but instead of using a set of sequential integers as the keys, maps allow us to use arbitrary values as “indexes.” Like arrays, maps are fast: looking up the value associated with a key will be fast (though not as fast as indexing an array) no matter how large the map is.

Create a new map with the `Map()` constructor:

```
let m = new Map(); // Create a new, empty map
let n = new Map([ // A new map initialized with string keys mapped to numbers
  ["one", 1],
  ["two", 2]
]);
```

The optional argument to the `Map()` constructor should be an iterable object that yields two element `[key, value]` arrays. In practice, this means that if you want to initialize a map when you create it, you’ll typically write out the desired keys and associated values as an array of arrays. But you can also use the `Map()` constructor to copy other maps or to copy the property names and values from an existing object:

```
let copy = new Map(n); // A new map with the same keys and values as map n
let o = { x: 1, y: 2}; // An object with two properties
let p = new Map(Object.entries(o)); // Same as new map([["x", 1], ["y", 2]])
```

Once you have created a Map object, you can query the value associated with a given key with `get()` and can add a new key/value pair with `set()`. Remember, though, that a map is a set of keys, each of which has an associated value. This is not quite the same as a set of key/value pairs. If you call `set()` with a key that already exists in the map, you will change the value associated with that key, not add a new key/value mapping. In addition to `get()` and `set()`, the Map class also defines methods that are like Set methods: use `has()` to check whether a map

includes the specified key; use `delete()` to remove a key (and its associated value) from the map; use `clear()` to remove all key/value pairs from the map; and use the `size` property to find out how many keys a map contains.

```
let m = new Map();      // Start with an empty map
m.size                // => 0: empty maps have no keys
m.set("one", 1);       // Map the key "one" to the value 1
m.set("two", 2);       // And the key "two" to the value 2.
m.size                // => 2: the map now has two keys
m.get("two")           // => 2: return the value associated with key "two"
m.get("three")         // => undefined: this key is not in the set
m.set("one", true);    // Change the value associated with an existing key
m.size                // => 2: the size doesn't change
m.has("one")           // => true: the map has a key "one"
m.has(true)            // => false: the map does not have a key true
m.delete("one")        // => true: the key existed and deletion succeeded
m.size                // => 1
m.delete("three")     // => false: failed to delete a nonexistent key
m.clear();             // Remove all keys and values from the map
```

Like the `add()` method of `Set`, the `set()` method of `Map` can be chained, which allows maps to be initialized without using arrays of arrays:

```
let m = new Map().set("one", 1).set("two", 2).set("three", 3);
m.size                // => 3
m.get("two")           // => 2
```

As with `Set`, any JavaScript value can be used as a key or a value in a `Map`. This includes `null`, `undefined`, and `Nan`, as well as reference types like objects and arrays. And as with the `Set` class, `Map` compares keys by identity, not by equality, so if you use an object or array as a key, it will be considered different from every other object and array, even those with exactly the same properties or elements:

```
let m = new Map();      // Start with an empty map.
m.set({}, 1);          // Map one empty object to the number 1.
m.set({}, 2);          // Map a different empty object to the number 2.
m.size                // => 2: there are two keys in this map
m.get({})              // => undefined: but this empty object is not a key
m.set(m, undefined);   // Map the map itself to the value undefined.
m.has(m)               // => true: m is a key in itself
m.get(m)               // => undefined: same value we'd get if m wasn't a key
```

`Map` objects are iterable, and each iterated value is a two-element array where the first element is a key and the second element is the value associated with that key. If you use the spread operator with a `Map` object, you'll get an array of arrays like the ones that we passed to the `Map()` constructor. And when iterating a map with a `for/of` loop, it is idiomatic to use destructuring assignment to assign the key and value to separate variables:

```
let m = new Map([[{"x": 1}, {"y": 2}]];
[...m]    // => [{"x": 1}, {"y": 2}]

for(let [key, value] of m) {
  // On the first iteration, key will be "x" and value will be 1
  // On the second iteration, key will be "y" and value will be 2
}
```

Like the `Set` class, the `Map` class iterates in insertion order. The first key/value pair iterated will be the one least recently added to the map, and the last pair iterated will be the one most recently added.

If you want to iterate just the keys or just the associated values of a map, use the `keys()` and `values()` methods: these return iterable objects that iterate keys and

values, in insertion order. (The `entries()` method returns an iterable object that iterates key/value pairs, but this is exactly the same as iterating the map directly.)

```
[...m.keys()]      // => ["x", "y"]: just the keys
[...m.values()]    // => [1, 2]: just the values
[...m.entries()]   // => [[{"x": 1}, {"y": 2}]]: same as [...m]
```

Map objects can also be iterated using the `forEach()` method that was first implemented by the `Array` class.

```
m.forEach((value, key) => { // note value, key NOT key, value
    // On the first invocation, value will be 1 and key will be "x"
    // On the second invocation, value will be 2 and key will be "y"
});
```

It may seem strange that the `value` parameter comes before the `key` parameter in the code above, since with `for/of` iteration, the `key` comes first. As noted at the start of this section, you can think of a map as a generalized array in which integer array indexes are replaced with arbitrary key values. The `forEach()` method of arrays passes the array element first and the array index second, so, by analogy, the `forEach()` method of a map passes the map value first and the map key second.

### 11.1.3 WeakMap and WeakSet

The `WeakMap` class is a variant (but not an actual subclass) of the `Map` class that does not prevent its key values from being garbage collected. Garbage collection is the process by which the JavaScript interpreter reclaims the memory of objects that are no longer “reachable” and cannot be used by the program. A regular map holds “strong” references to its key values, and they remain reachable through the map, even if all other references to them are gone. The `WeakMap`, by contrast, keeps “weak” references to its key values so that they are not reachable through the `WeakMap`, and their presence in the map does not prevent their memory from being reclaimed.

The `WeakMap()` constructor is just like the `Map()` constructor, but there are some significant differences between `WeakMap` and `Map`:

- `WeakMap` keys must be objects or arrays; primitive values are not subject to garbage collection and cannot be used as keys.
- `WeakMap` implements only the `get()`, `set()`, `has()`, and `delete()` methods. In particular, `WeakMap` is not iterable and does not define `keys()`, `values()`, or `forEach()`. If `WeakMap` was iterable, then its keys would be reachable and it wouldn’t be weak.
- Similarly, `WeakMap` does not implement the `size` property because the size of a `WeakMap` could change at any time as objects are garbage collected.

The intended use of `WeakMap` is to allow you to associate values with objects without causing memory leaks. Suppose, for example, that you are writing a function that takes an object argument and needs to perform some time-consuming computation on that object. For efficiency, you’d like to cache the computed value for later reuse. If you use a `Map` object to implement the cache, you will prevent any of the objects from ever being reclaimed, but by using a `WeakMap`, you avoid this problem. (You can often achieve a similar result using a private `Symbol` property to cache the computed value directly on the object. See [§6.10.3](#).)

`WeakSet` implements a set of objects that does not prevent those objects from being garbage collected. The `WeakSet()` constructor works like the `Set()`

constructor, but WeakSet objects differ from Set objects in the same ways that WeakMap objects differ from Map objects:

- WeakSet does not allow primitive values as members.
- WeakSet implements only the `add()`, `has()`, and `delete()` methods and is not iterable.
- WeakSet does not have a `size` property.

WeakSet is not frequently used: its use cases are like those for WeakMap. If you want to mark (or “brand”) an object as having some special property or type, for example, you could add it to a WeakSet. Then, elsewhere, when you want to check for that property or type, you can test for membership in that WeakSet. Doing this with a regular set would prevent all marked objects from being garbage collected, but this is not a concern when using WeakSet.

## 11.2 Typed Arrays and Binary Data

Regular JavaScript arrays can have elements of any type and can grow or shrink dynamically. JavaScript implementations perform lots of optimizations so that typical uses of JavaScript arrays are very fast. Nevertheless, they are still quite different from the array types of lower-level languages like C and Java. *Typed arrays*, which are new in ES6,<sup>3</sup> are much closer to the low-level arrays of those languages. Typed arrays are not technically arrays (`Array.isArray()` returns `false` for them), but they implement all of the array methods described in §7.8 plus a few more of their own. They differ from regular arrays in some very important ways, however:

- The elements of a typed array are all numbers. Unlike regular JavaScript numbers, however, typed arrays allow you to specify the type (signed and unsigned integers and IEEE-754 floating point) and size (8 bits to 64 bits) of the numbers to be stored in the array.
- You must specify the length of a typed array when you create it, and that length can never change.
- The elements of a typed array are always initialized to 0 when the array is created.

### 11.2.1 Typed Array Types

JavaScript does not define a `TypedArray` class. Instead, there are 11 kinds of typed arrays, each with a different element type and constructor:

Constructor	Numeric type
<code>Int8Array()</code>	signed bytes
<code>Uint8Array()</code>	unsigned bytes
<code>Uint8ClampedArray()</code>	unsigned bytes without rollover
<code>Int16Array()</code>	signed 16-bit short integers
<code>Uint16Array()</code>	unsigned 16-bit short integers
<code>Int32Array()</code>	signed 32-bit integers
<code>Uint32Array()</code>	unsigned 32-bit integers
<code>BigInt64Array()</code>	signed 64-bit BigInt values (ES2020)

Constructor	Numeric type
<code>BigUint64Array()</code>	unsigned 64-bit BigInt values (ES2020)
<code>Float32Array()</code>	32-bit floating-point value
<code>Float64Array()</code>	64-bit floating-point value: a regular JavaScript number

The types whose names begin with `Int` hold signed integers, of 1, 2, or 4 bytes (8, 16, or 32 bits). The types whose names begin with `Uint` hold unsigned integers of those same lengths. The “`BigInt`” and “`BigUint`” types hold 64-bit integers, represented in JavaScript as `BigInt` values (see §3.2.5). The types that begin with `Float` hold floating-point numbers. The elements of a `Float64Array` are of the same type as regular JavaScript numbers. The elements of a `Float32Array` have lower precision and a smaller range but require only half the memory. (This type is called `float` in C and Java.)

`Uint8ClampedArray` is a special-case variant on `Uint8Array`. Both of these types hold unsigned bytes and can represent numbers between 0 and 255. With `Uint8Array`, if you store a value larger than 255 or less than zero into an array element, it “wraps around,” and you get some other value. This is how computer memory works at a low level, so this is very fast. `Uint8ClampedArray` does some extra type checking so that, if you store a value greater than 255 or less than 0, it “clamps” to 255 or 0 and does not wrap around. (This clamping behavior is required by the HTML `<canvas>` element’s low-level API for manipulating pixel colors.)

Each of the typed array constructors has a `BYTES_PER_ELEMENT` property with the value 1, 2, 4, or 8, depending on the type.

## 11.2.2 Creating Typed Arrays

The simplest way to create a typed array is to call the appropriate constructor with one numeric argument that specifies the number of elements you want in the array:

```
let bytes = new Uint8Array(1024);      // 1024 bytes
let matrix = new Float64Array(9);       // A 3x3 matrix
let point = new Int16Array(3);          // A point in 3D space
let rgba = new Uint8ClampedArray(4);    // A 4-byte RGBA pixel value
let sudoku = new Int8Array(81);         // A 9x9 sudoku board
```

When you create typed arrays in this way, the array elements are all guaranteed to be initialized to `0`, `0n`, or `0.0`. But if you know the values you want in your typed array, you can also specify those values when you create the array. Each of the typed array constructors has static `from()` and `of()` factory methods that work like `Array.from()` and `Array.of()`:

```
let white = Uint8ClampedArray.of(255, 255, 255, 0); // RGBA opaque white
```

Recall that the `Array.from()` factory method expects an array-like or iterable object as its first argument. The same is true for the typed array variants, except that the iterable or array-like object must also have numeric elements. Strings are iterable, for example, but it would make no sense to pass them to the `from()` factory method of a typed array.

If you are just using the one-argument version of `from()`, you can drop the `.from` and pass your iterable or array-like object directly to the constructor function, which behaves exactly the same. Note that both the constructor and the `from()` factory method allow you to copy existing typed arrays, while possibly changing the type:

```
let ints = Uint32Array.from(white); // The same 4 numbers, but as ints
```

When you create a new typed array from an existing array, iterable, or array-like object, the values may be truncated in order to fit the type constraints of your array. There are no warnings or errors when this happens:

```
// Floats truncated to ints, longer ints truncated to 8 bits
Uint8Array.of(1.23, 2.99, 45000) // => new Uint8Array([1, 2, 200])
```

Finally, there is one more way to create typed arrays that involves the `ArrayBuffer` type. An `ArrayBuffer` is an opaque reference to a chunk of memory. You can create one with the constructor; just pass in the number of bytes of memory you'd like to allocate:

```
let buffer = new ArrayBuffer(1024*1024);
buffer.byteLength // => 1024*1024; one megabyte of memory
```

The `ArrayBuffer` class does not allow you to read or write any of the bytes that you have allocated. But you can create typed arrays that use the buffer's memory and that do allow you to read and write that memory. To do this, call the typed array constructor with an `ArrayBuffer` as the first argument, a byte offset within the array buffer as the second argument, and the array length (in elements, not in bytes) as the third argument. The second and third arguments are optional. If you omit both, then the array will use all of the memory in the array buffer. If you omit only the length argument, then your array will use all of the available memory between the start position and the end of the array. One more thing to bear in mind about this form of the typed array constructor: arrays must be memory aligned, so if you specify a byte offset, the value should be a multiple of the size of your type. The `Int32Array()` constructor requires a multiple of four, for example, and the `Float64Array()` requires a multiple of eight.

Given the `ArrayBuffer` created earlier, you could create typed arrays like these:

```
let asbytes = new Uint8Array(buffer);           // Viewed as bytes
let asints = new Int32Array(buffer);            // Viewed as 32-bit signed ints
let lastK = new Uint8Array(buffer, 1023*1024); // Last kilobyte as bytes
let ints2 = new Int32Array(buffer, 1024, 256); // 2nd kilobyte as 256 integers
```

These four typed arrays offer four different views into the memory represented by the `ArrayBuffer`. It is important to understand that all typed arrays have an underlying `ArrayBuffer`, even if you do not explicitly specify one. If you call a typed array constructor without passing a buffer object, a buffer of the appropriate size will be automatically created. As described later, the `buffer` property of any typed array refers to its underlying `ArrayBuffer` object. The reason to work directly with `ArrayBuffer` objects is that sometimes you may want to have multiple typed array views of a single buffer.

### 11.2.3 Using Typed Arrays

Once you have created a typed array, you can read and write its elements with regular square-bracket notation, just as you would with any other array-like object:

```
// Return the largest prime smaller than n, using the sieve of Eratosthenes
function sieve(n) {
    let a = new Uint8Array(n+1);          // a[x] will be 1 if x is composite
    let max = Math.floor(Math.sqrt(n));   // Don't do factors higher than this
    let p = 2;                          // 2 is the first prime
    while(p <= max) {                  // For primes less than max
        for(let i = 2*p; i <= n; i += p) // Mark multiples of p as composite
            a[i] = 1;
    }
}
```

```

        while(a[++p]) /* empty */;           // The next unmarked index is prime
    }
    while(a[n]) n--;
    return n;                           // Loop backward to find the last prime
}                                     // And return it
}

```

The function here computes the largest prime number smaller than the number you specify. The code is exactly the same as it would be with a regular JavaScript array, but using `Uint8Array()` instead of `Array()` makes the code run more than four times faster and use eight times less memory in my testing.

Typed arrays are not true arrays, but they re-implement most array methods, so you can use them pretty much just like you'd use regular arrays:

```
let ints = new Int16Array(10);           // 10 short integers
ints.fill(3).map(x=>x*x).join("")    // => "9999999999"
```

Remember that typed arrays have fixed lengths, so the `length` property is read-only, and methods that change the length of the array (such as `push()`, `pop()`, `unshift()`, `shift()`, and `splice()`) are not implemented for typed arrays. Methods that alter the contents of an array without changing the length (such as `sort()`, `reverse()`, and `fill()`) are implemented. Methods like `map()` and `slice()` that return new arrays return a typed array of the same type as the one they are called on.

## 11.2.4 Typed Array Methods and Properties

In addition to standard array methods, typed arrays also implement a few methods of their own. The `set()` method sets multiple elements of a typed array at once by copying the elements of a regular or typed array into a typed array:

```
let bytes = new Uint8Array(1024);          // A 1K buffer
let pattern = new Uint8Array([0,1,2,3]);   // An array of 4 bytes
bytes.set(pattern);                     // Copy them to the start of another byte array
bytes.set(pattern, 4);                  // Copy them again at a different offset
bytes.set([0,1,2,3], 8);               // Or just copy values direct from a regular array
bytes.slice(0, 12)                     // => new Uint8Array([0,1,2,3,0,1,2,3,0,1,2,3])
```

The `set()` method takes an array or typed array as its first argument and an element offset as its optional second argument, which defaults to 0 if left unspecified. If you are copying values from one typed array to another, the operation will likely be extremely fast.

Typed arrays also have a `subarray` method that returns a portion of the array on which it is called:

```
let ints = new Int16Array([0,1,2,3,4,5,6,7,8,9]);      // 10 short integers
let last3 = ints.subarray(ints.length-3, ints.length); // Last 3 of them
last3[0]      // => 7: this is the same as ints[7]
```

`subarray()` takes the same arguments as the `slice()` method and seems to work the same way. But there is an important difference. `slice()` returns the specified elements in a new, independent typed array that does not share memory with the original array. `subarray()` does not copy any memory; it just returns a new view of the same underlying values:

```
ints[9] = -1; // Change a value in the original array and...
last3[2]      // => -1: it also changes in the subarray
```

The fact that the `subarray()` method returns a new view of an existing array brings us back to the topic of `ArrayBuffers`. Every typed array has three properties that

relate to the underlying buffer:

```
last3.buffer          // The ArrayBuffer object for a typed array
last3.buffer === ints.buffer // => true: both are views of the same buffer
last3.byteOffset      // => 14: this view starts at byte 14 of the buffer
last3.byteLength      // => 6: this view is 6 bytes (3 16-bit ints) long
last3.buffer.byteLength // => 20: but the underlying buffer has 20 bytes
```

The `buffer` property is the `ArrayBuffer` of the array. `byteOffset` is the starting position of the array's data within the underlying buffer. And `byteLength` is the length of the array's data in bytes. For any typed array, `a`, this invariant should always be true:

```
a.length * a.BYTES_PER_ELEMENT === a.byteLength // => true
```

`ArrayBuffers` are just opaque chunks of bytes. You can access those bytes with typed arrays, but an `ArrayBuffer` is not itself a typed array. Be careful, however: you can use numeric array indexing with `ArrayBuffers` just as you can with any JavaScript object. Doing so does not give you access to the bytes in the buffer, but it can cause confusing bugs:

```
let bytes = new Uint8Array(8);
bytes[0] = 1;           // Set the first byte to 1
bytes.buffer[0]        // => undefined: buffer doesn't have index 0
bytes.buffer[1] = 255; // Try incorrectly to set a byte in the buffer
bytes.buffer[1]        // => 255: this just sets a regular JS property
bytes[1]               // => 0: the line above did not set the byte
```

We saw previously that you can create an `ArrayBuffer` with the `ArrayBuffer()` constructor and then create typed arrays that use that buffer. Another approach is to create an initial typed array, then use the `buffer` of that array to create other views:

```
let bytes = new Uint8Array(1024);           // 1024 bytes
let ints = new Uint32Array(bytes.buffer);    // or 256 integers
let floats = new Float64Array(bytes.buffer); // or 128 doubles
```

## 11.2.5 DataView and Endianness

Typed arrays allow you to view the same sequence of bytes in chunks of 8, 16, 32, or 64 bits. This exposes the “endianness”: the order in which bytes are arranged into longer words. For efficiency, typed arrays use the native endianness of the underlying hardware. On little-endian systems, the bytes of a number are arranged in an `ArrayBuffer` from least significant to most significant. On big-endian platforms, the bytes are arranged from most significant to least significant. You can determine the endianness of the underlying platform with code like this:

```
// If the integer 0x00000001 is arranged in memory as 01 00 00 00, then
// we're on a little-endian platform. On a big-endian platform, we'd get
// bytes 00 00 00 01 instead.
let littleEndian = new Int8Array(new Int32Array([1]).buffer)[0] === 1;
```

Today, the most common CPU architectures are little-endian. Many network protocols, and some binary file formats, require big-endian byte ordering, however. If you're using typed arrays with data that came from the network or from a file, you can't just assume that the platform endianness matches the byte order of the data. In general, when working with external data, you can use `Int8Array` and `Uint8Array` to view the data as an array of individual bytes, but you should not use the other typed arrays with multibyte word sizes. Instead, you can use the `DataView` class, which defines methods for reading and writing values from an `ArrayBuffer` with explicitly specified byte ordering:

```
// Assume we have a typed array of bytes of binary data to process. First,
// we create a DataView object so we can flexibly read and write
// values from those bytes
let view = new DataView(bytes.buffer,
                        bytes.byteOffset,
                        bytes.byteLength);

let int = view.getInt32(0);      // Read big-endian signed int from byte 0
int = view.getInt32(4, false);  // Next int is also big-endian
int = view.getUint32(8, true);  // Next int is little-endian and unsigned
view.setUint32(8, int, false); // Write it back in big-endian format
```

DataView defines 10 get methods for each of the 10 typed array classes (excluding Uint8ClampedArray). They have names like getInt16(), getUint32(), getBigInt64(), and getFloat64(). The first argument is the byte offset within the ArrayBuffer at which the value begins. All of these getter methods, other than getInt8() and getUint8(), accept an optional boolean value as their second argument. If the second argument is omitted or is `false`, big-endian byte ordering is used. If the second argument is `true`, little-endian ordering is used.

DataView also defines 10 corresponding Set methods that write values into the underlying ArrayBuffer. The first argument is the offset at which the value begins. The second argument is the value to write. Each of the methods, except `setInt8()` and `setUint8()`, accepts an optional third argument. If the argument is omitted or is `false`, the value is written in big-endian format with the most significant byte first. If the argument is `true`, the value is written in little-endian format with the least significant byte first.

Typed arrays and the DataView class give you all the tools you need to process binary data and enable you to write JavaScript programs that do things like decompressing ZIP files or extracting metadata from JPEG files.

## 11.3 Pattern Matching with Regular Expressions

A *regular expression* is an object that describes a textual pattern. The JavaScript RegExp class represents regular expressions, and both String and RegExp define methods that use regular expressions to perform powerful pattern-matching and search-and-replace functions on text. In order to use the RegExp API effectively, however, you must also learn how to describe patterns of text using the regular expression grammar, which is essentially a mini programming language of its own. Fortunately, the JavaScript regular expression grammar is quite similar to the grammar used by many other programming languages, so you may already be familiar with it. (And if you are not, the effort you invest in learning JavaScript regular expressions will probably be useful to you in other programming contexts as well.)

The subsections that follow describe the regular expression grammar first, and then, after explaining how to write regular expressions, they explain how you can use them with methods of the String and RegExp classes.

### 11.3.1 Defining Regular Expressions

In JavaScript, regular expressions are represented by RegExp objects. RegExp objects may be created with the `RegExp()` constructor, of course, but they are more often created using a special literal syntax. Just as string literals are specified as characters within quotation marks, regular expression literals are specified as

characters within a pair of slash (/) characters. Thus, your JavaScript code may contain lines like this:

```
let pattern = /s$/;
```

This line creates a new RegExp object and assigns it to the variable pattern. This particular RegExp object matches any string that ends with the letter “s.” This regular expression could have equivalently been defined with the RegExp() constructor, like this:

```
let pattern = new RegExp("s$");
```

Regular-expression pattern specifications consist of a series of characters. Most characters, including all alphanumeric characters, simply describe characters to be matched literally. Thus, the regular expression /java/ matches any string that contains the substring “java”. Other characters in regular expressions are not matched literally but have special significance. For example, the regular expression /s\$/ contains two characters. The first, “s”, matches itself literally. The second, “\$”, is a special meta-character that matches the end of a string. Thus, this regular expression matches any string that contains the letter “s” as its last character.

As we’ll see, regular expressions can also have one or more flag characters that affect how they work. Flags are specified following the second slash character in RegExp literals, or as a second string argument to the RegExp() constructor. If we wanted to match strings that end with “s” or “S”, for example, we could use the i flag with our regular expression to indicate that we want case-insensitive matching:

```
let pattern = /s$/i;
```

The following sections describe the various characters and meta-characters used in JavaScript regular expressions.

## Literal characters

All alphabetic characters and digits match themselves literally in regular expressions. JavaScript regular expression syntax also supports certain nonalphabetic characters through escape sequences that begin with a backslash (\). For example, the sequence \n matches a literal newline character in a string. [Table 11-1](#) lists these characters.

Table 11-1. Regular-expression literal characters

Character	Matches
Alphanumeric character	Itself
\0	The NUL character (\u0000)
\t	Tab (\u0009)
\n	Newline (\u000A)
\v	Vertical tab (\u000B)
\f	Form feed (\u000C)
\r	Carriage return (\u000D)
\xnn	The Latin character specified by the hexadecimal number nn; for example, \x0A is the same as \n.
\uxxxx	The Unicode character specified by the hexadecimal number xxxx; for example, \u0009 is the same as \t.

Character	Matches
\u{n}	The Unicode character specified by the codepoint <i>n</i> , where <i>n</i> is one to six hexadecimal digits between 0 and 10FFFF. Note that this syntax is only supported in regular expressions that use the <code>u</code> flag.
\cX	The control character ^X; for example, \cJ is equivalent to the newline character \n.

A number of punctuation characters have special meanings in regular expressions. They are:

^ \$ . \* + ? = ! : | \ / ( ) [ ] { }

The meanings of these characters are discussed in the sections that follow. Some of these characters have special meaning only within certain contexts of a regular expression and are treated literally in other contexts. As a general rule, however, if you want to include any of these punctuation characters literally in a regular expression, you must precede them with a \. Other punctuation characters, such as quotation marks and @, do not have special meaning and simply match themselves literally in a regular expression.

If you can't remember exactly which punctuation characters need to be escaped with a backslash, you may safely place a backslash before any punctuation character. On the other hand, note that many letters and numbers have special meaning when preceded by a backslash, so any letters or numbers that you want to match literally should not be escaped with a backslash. To include a backslash character literally in a regular expression, you must escape it with a backslash, of course. For example, the following regular expression matches any string that includes a backslash: /\\\/. (And if you use the `RegExp()` constructor, keep in mind that any backslashes in your regular expression need to be doubled, since strings also use backslashes as an escape character.)

## Character classes

Individual literal characters can be combined into *character classes* by placing them within square brackets. A character class matches any one character that is contained within it. Thus, the regular expression /[abc]/ matches any one of the letters a, b, or c. Negated character classes can also be defined; these match any character except those contained within the brackets. A negated character class is specified by placing a caret (^) as the first character inside the left bracket. The `RegExp` /^[^abc]/ matches any one character other than a, b, or c. Character classes can use a hyphen to indicate a range of characters. To match any one lowercase character from the Latin alphabet, use /[a-z]/, and to match any letter or digit from the Latin alphabet, use /[a-zA-Z0-9]/. (And if you want to include an actual hyphen in your character class, simply make it the last character before the right bracket.)

Because certain character classes are commonly used, the JavaScript regular-expression syntax includes special characters and escape sequences to represent these common classes. For example, \s matches the space character, the tab character, and any other Unicode whitespace character; \S matches any character that is *not* Unicode whitespace. [Table 11-2](#) lists these characters and summarizes character-class syntax. (Note that several of these character-class escape sequences match only ASCII characters and have not been extended to work with Unicode characters. You can, however, explicitly define your own Unicode character classes; for example, /[\u0400-\u04FF]/ matches any one Cyrillic character.)

Table 11-2. Regular expression character classes

Character	Matches
[...]	Any one character between the brackets.
[^...]	Any one character not between the brackets.
.	Any character except newline or another Unicode line terminator. Or, if the RegExp uses the s flag, then a period matches any character, including line terminators.
\w	Any ASCII word character. Equivalent to [a-zA-Z0-9_].
\W	Any character that is not an ASCII word character. Equivalent to [^a-zA-Z0-9_].
\s	Any Unicode whitespace character.
\S	Any character that is not Unicode whitespace.
\d	Any ASCII digit. Equivalent to [0-9].
\D	Any character other than an ASCII digit. Equivalent to [^0-9].
[\b]	A literal backspace (special case).

Note that the special character-class escapes can be used within square brackets. \s matches any whitespace character, and \d matches any digit, so /[\s\d]/ matches any one whitespace character or digit. Note that there is one special case. As you'll see later, the \b escape has a special meaning. When used within a character class, however, it represents the backspace character. Thus, to represent a backspace character literally in a regular expression, use the character class with one element: /[\b]/.

### Unicode Character Classes

In ES2018, if a regular expression uses the u flag, then character classes \p{...} and its negation \P{...} are supported. (As of early 2020, this is implemented by Node, Chrome, Edge, and Safari, but not Firefox.) These character classes are based on properties defined by the Unicode standard, and the set of characters they represent may change as Unicode evolves.

The \d character class matches only ASCII digits. If you want to match one decimal digit from any of the world's writing systems, you can use /\p{Decimal\_Number}/u. And if you want to match any one character that is *not* a decimal digit in any language, you can capitalize the p and write \P{Decimal\_Number}. If you want to match any number-like character, including fractions and roman numerals, you can use \p{Number}. Note that "Decimal\_Number" and "Number" are not specific to JavaScript or to regular expression grammar: it is the name of a category of characters defined by the Unicode standard.

The \w character class only works for ASCII text, but with \p, we can approximate an internationalized version like this:

```
/[\p{Alphabetic}\p{Decimal_Number}\p{Mark}]/u
```

(Though to be fully compatible with the complexity of the world's languages, we really need to add in the categories "Connector\_Punctuation" and "Join\_Control" as well.)

As a final example, the \p syntax also allows us to define regular expressions that match characters from a particular alphabet or script:

```
let greekLetter = /\p{Script=Greek}/u;
let cyrillicLetter = /\p{Script=Cyrillic}/u;
```

## Repetition

With the regular expression syntax you've learned so far, you can describe a two-digit number as `/\d\d/` and a four-digit number as `/\d\d\d\d/`. But you don't have any way to describe, for example, a number that can have any number of digits or a string of three letters followed by an optional digit. These more complex patterns use regular expression syntax that specifies how many times an element of a regular expression may be repeated.

The characters that specify repetition always follow the pattern to which they are being applied. Because certain types of repetition are quite commonly used, there are special characters to represent these cases. For example, `+` matches one or more occurrences of the previous pattern.

[Table 11-3](#) summarizes the repetition syntax.

Table 11-3. Regular expression repetition characters

Character	Meaning
<code>{n,m}</code>	Match the previous item at least $n$ times but no more than $m$ times.
<code>{n,}</code>	Match the previous item $n$ or more times.
<code>{n}</code>	Match exactly $n$ occurrences of the previous item.
<code>?</code>	Match zero or one occurrences of the previous item. That is, the previous item is optional. Equivalent to <code>{0,1}</code> .
<code>+</code>	Match one or more occurrences of the previous item. Equivalent to <code>{1,}</code> .
<code>*</code>	Match zero or more occurrences of the previous item. Equivalent to <code>{0,}</code> .

The following lines show some examples:

```
let r = /\d{2,4}/; // Match between two and four digits
r = /\w{3}\d?/;   // Match exactly three word characters and an optional digit
r = /\s+java\s+/; // Match "java" with one or more spaces before and after
r = /[^()*/]/;    // Match zero or more characters that are not open parens
```

Note that in all of these examples, the repetition specifiers apply to the single character or character class that precedes them. If you want to match repetitions of more complicated expressions, you'll need to define a group with parentheses, which are explained in the following sections.

Be careful when using the `*` and `?` repetition characters. Since these characters may match zero instances of whatever precedes them, they are allowed to match nothing. For example, the regular expression `/a*/` actually matches the string “bbbb” because the string contains zero occurrences of the letter a!

## Non-greedy repetition

The repetition characters listed in [Table 11-3](#) match as many times as possible while still allowing any following parts of the regular expression to match. We say that this repetition is “greedy.” It is also possible to specify that repetition should be done in a non-greedy way. Simply follow the repetition character or characters with a question mark: `??`, `+?`, `*?`, or even `{1,5}?`. For example, the regular expression `/a+/` matches one or more occurrences of the letter a. When applied to the string “aaa”, it matches all three letters. But `/a+?/` matches one or more occurrences of the letter a, matching as few characters as necessary. When applied to the same string, this pattern matches only the first letter a.

Using non-greedy repetition may not always produce the results you expect. Consider the pattern `/a+b/`, which matches one or more a's, followed by the letter b. When applied to the string "aaab", it matches the entire string. Now let's use the non-greedy version: `/a+?b/`. This should match the letter b preceded by the fewest number of a's possible. When applied to the same string "aaab", you might expect it to match only one a and the last letter b. In fact, however, this pattern matches the entire string, just like the greedy version of the pattern. This is because regular expression pattern matching is done by finding the first position in the string at which a match is possible. Since a match is possible starting at the first character of the string, shorter matches starting at subsequent characters are never even considered.

## Alternation, grouping, and references

The regular expression grammar includes special characters for specifying alternatives, grouping subexpressions, and referring to previous subexpressions. The | character separates alternatives. For example, `/ab|cd|ef/` matches the string "ab" or the string "cd" or the string "ef". And `/\d{3}|[a-z]{4}/` matches either three digits or four lowercase letters.

Note that alternatives are considered left to right until a match is found. If the left alternative matches, the right alternative is ignored, even if it would have produced a "better" match. Thus, when the pattern `/a|ab/` is applied to the string "ab", it matches only the first letter.

Parentheses have several purposes in regular expressions. One purpose is to group separate items into a single subexpression so that the items can be treated as a single unit by |, \*, +, ?, and so on. For example, `/java(script)?/` matches "java" followed by the optional "script". And `/(ab|cd)+|ef/` matches either the string "ef" or one or more repetitions of either of the strings "ab" or "cd".

Another purpose of parentheses in regular expressions is to define subpatterns within the complete pattern. When a regular expression is successfully matched against a target string, it is possible to extract the portions of the target string that matched any particular parenthesized subpattern. (You'll see how these matching substrings are obtained later in this section.) For example, suppose you are looking for one or more lowercase letters followed by one or more digits. You might use the pattern `/[a-z]+\\d+/`. But suppose you only really care about the digits at the end of each match. If you put that part of the pattern in parentheses `(/[a-z]+(\\d+)/)`, you can extract the digits from any matches you find, as explained later.

A related use of parenthesized subexpressions is to allow you to refer back to a subexpression later in the same regular expression. This is done by following a \ character by a digit or digits. The digits refer to the position of the parenthesized subexpression within the regular expression. For example, \1 refers back to the first subexpression, and \3 refers to the third. Note that, because subexpressions can be nested within others, it is the position of the left parenthesis that is counted. In the following regular expression, for example, the nested subexpression `([Ss]cript)` is referred to as \2:

```
/([Jj]ava([Ss]cript))\\sis\\s(fun\\w*)/
```

A reference to a previous subexpression of a regular expression does *not* refer to the pattern for that subexpression but rather to the text that matched the pattern. Thus, references can be used to enforce a constraint that separate portions of a string contain exactly the same characters. For example, the following regular expression matches zero or more characters within single or double quotes.

However, it does not require the opening and closing quotes to match (i.e., both single quotes or both double quotes):

```
/[""][^"]*["]/
```

To require the quotes to match, use a reference:

```
/(["])[^"]*\1/
```

The \1 matches whatever the first parenthesized subexpression matched. In this example, it enforces the constraint that the closing quote match the opening quote. This regular expression does not allow single quotes within double-quoted strings or vice versa. (It is not legal to use a reference within a character class, so you cannot write: /(["])[^\1]\*\1./.)

When we cover the RegExp API later, you'll see that this kind of reference to a parenthesized subexpression is a powerful feature of regular-expression search-and-replace operations.

It is also possible to group items in a regular expression without creating a numbered reference to those items. Instead of simply grouping the items within ( and ), begin the group with (?: and end it with ). Consider the following pattern:

```
/([Jj]ava(?:[Ss]cript))\sis\s(fun\bw*)/
```

In this example, the subexpression (?:[Ss]cript) is used simply for grouping, so the ? repetition character can be applied to the group. These modified parentheses do not produce a reference, so in this regular expression, \2 refers to the text matched by (fun\bw\*).

[Table 11-4](#) summarizes the regular expression alternation, grouping, and referencing operators.

Table 11-4. Regular expression alternation, grouping, and reference characters

Character	Meaning
	Alternation: match either the subexpression to the left or the subexpression to the right.
(...)	Grouping: group items into a single unit that can be used with *, +, ?,  , and so on. Also remember the characters that match this group for use with later references.
(?:...)	Grouping only: group items into a single unit, but do not remember the characters that match this group.
\n	Match the same characters that were matched when group number <i>n</i> was first matched. Groups are subexpressions within (possibly nested) parentheses. Group numbers are assigned by counting left parentheses from left to right. Groups formed with (?: are not numbered.

### Named Capture Groups

ES2018 standardizes a new feature that can make regular expressions more self-documenting and easier to understand. This new feature is known as “named capture groups” and it allows us to associate a name with each left parenthesis in a regular expression so that we can refer to the matching text by name rather than by number. Equally important: using names allows someone reading the code to more easily understand the purpose of that portion of the regular expression. As of early 2020, this feature is implemented in Node, Chrome, Edge, and Safari, but not yet by Firefox.

To name a group, use `(?<...>` instead of `(` and put the name between the angle brackets. For example, here is a regular expression that might be used to check the formatting of the final line of a US mailing address:

```
/(?<city>\w+) (?<state>[A-Z]{2}) (?<zipcode>\d{5})(?<zip9>-\d{4})?/
```

Notice how much context the group names provide to make the regular expression easier to understand. In [§11.3.2](#), when we discuss the `String replace()` and `match()` methods and the `RegExp exec()` method, you'll see how the `RegExp` API allows you to refer to the text that matches each of these groups by name rather than by position.

If you want to refer back to a named capture group within a regular expression, you can do that by name as well. In the preceding example, we were able to use a regular expression “backreference” to write a `RegExp` that would match a single- or double-quoted string where the open and close quotes had to match. We could rewrite this `RegExp` using a named capturing group and a named backreference like this:

```
/(?<quote>[""])[^"]*\k<quote>/
```

The `\k<quote>` is a named backreference to the named group that captures the open quotation mark.

## Specifying match position

As described earlier, many elements of a regular expression match a single character in a string. For example, `\s` matches a single character of whitespace. Other regular expression elements match the positions between characters instead of actual characters. `\b`, for example, matches an ASCII word boundary—the boundary between a `\w` (ASCII word character) and a `\W` (nonword character), or the boundary between an ASCII word character and the beginning or end of a string.<sup>4</sup> Elements such as `\b` do not specify any characters to be used in a matched string; what they do specify, however, are legal positions at which a match can occur. Sometimes these elements are called *regular expression anchors* because they anchor the pattern to a specific position in the search string. The most commonly used anchor elements are `^`, which ties the pattern to the beginning of the string, and `$`, which anchors the pattern to the end of the string.

For example, to match the word “JavaScript” on a line by itself, you can use the regular expression `/^JavaScript$/`. If you want to search for “Java” as a word by itself (not as a prefix, as it is in “JavaScript”), you can try the pattern `/\sJava\s/`, which requires a space before and after the word. But there are two problems with this solution. First, it does not match “Java” at the beginning or the end of a string, but only if it appears with space on either side. Second, when this pattern does find a match, the matched string it returns has leading and trailing spaces, which is not quite what's needed. So instead of matching actual space characters with `\s`, match (or anchor to) word boundaries with `\b`. The resulting expression is `/\bJava\b/`. The element `\B` anchors the match to a location that is not a word boundary. Thus, the pattern `/\B[Ss]cript/` matches “JavaScript” and “postscript”, but not “script” or “Scripting”.

You can also use arbitrary regular expressions as anchor conditions. If you include an expression within `(?=` and `)` characters, it is a lookahead assertion, and it specifies that the enclosed characters must match, without actually matching them. For example, to match the name of a common programming language, but only if it is followed by a colon, you could use `/[Jj]ava([Ss]cript)?(?=:)/`. This pattern matches the word “JavaScript” in “JavaScript: The Definitive Guide”, but

it does not match “Java” in “Java in a Nutshell” because it is not followed by a colon.

If you instead introduce an assertion with `(?! )`, it is a negative lookahead assertion, which specifies that the following characters must not match. For example, `/Java(?![Script])([A-Z]\w*)/` matches “Java” followed by a capital letter and any number of additional ASCII word characters, as long as “Java” is not followed by “Script”. It matches “JavaBeans” but not “Javanese”, and it matches “JavaScript” but not “JavaScript” or “JavaScripter”. [Table 11-5](#) summarizes regular expression anchors.

Table 11-5. Regular expression anchor characters

Character	Meaning
<code>^</code>	Match the beginning of the string or, with the <code>m</code> flag, the beginning of a line.
<code>\$</code>	Match the end of the string and, with the <code>m</code> flag, the end of a line.
<code>\b</code>	Match a word boundary. That is, match the position between a <code>\w</code> character and a <code>\W</code> character or between a <code>\w</code> character and the beginning or end of a string. (Note, however, that <code>[\b]</code> matches backspace.)
<code>\B</code>	Match a position that is not a word boundary.
<code>(?=p)</code>	A positive lookahead assertion. Require that the following characters match the pattern <code>p</code> , but do not include those characters in the match.
<code>(?!p)</code>	A negative lookahead assertion. Require that the following characters do not match the pattern <code>p</code> .

### Lookbehind Assertions

ES2018 extends regular expression syntax to allow “lookbehind” assertions. These are like lookahead assertions but refer to text before the current match position. As of early 2020, these are implemented in Node, Chrome, and Edge, but not Firefox or Safari.

Specify a positive lookbehind assertion with `(?<= ...)` and a negative lookbehind assertion with `(?<! ...)`. For example, if you were working with US mailing addresses, you could match a 5-digit zip code, but only when it follows a two-letter state abbreviation, like this:

```
/(?<= [A-Z]{2} )\d{5}/
```

And you could match a string of digits that is not preceded by a Unicode currency symbol with a negative lookbehind assertion like this:

```
/(?<![\p{Currency_Symbol}\d.])\d+(\.\d+)?/u
```

### Flags

Every regular expression can have one or more flags associated with it to alter its matching behavior. JavaScript defines six possible flags, each of which is represented by a single letter. Flags are specified after the second / character of a regular expression literal or as a string passed as the second argument to the `RegExp()` constructor. The supported flags and their meanings are:

`g`

The `g` flag indicates that the regular expression is “global”—that is, that we intend to use it to find all matches within a string rather than just finding the

first match. This flag does not alter the way that pattern matching is done, but, as we'll see later, it does alter the behavior of the String `match()` method and the RegExp `exec()` method in important ways.

i

The `i` flag specifies that pattern matching should be case-insensitive.

m

The `m` flag specifies that matching should be done in “multiline” mode. It says that the RegExp will be used with multiline strings and that the `^` and `$` anchors should match both the beginning and end of the string and also the beginning and end of individual lines within the string.

s

Like the `m` flag, the `s` flag is also useful when working with text that includes newlines. Normally, a `“.”` in a regular expression matches any character except a line terminator. When the `s` flag is used, however, `“.”` will match any character, including line terminators. The `s` flag was added to JavaScript in ES2018 and, as of early 2020, is supported in Node, Chrome, Edge, and Safari, but not Firefox.

u

The `u` flag stands for Unicode, and it makes the regular expression match full Unicode codepoints rather than matching 16-bit values. This flag was introduced in ES6, and you should make a habit of using it on all regular expressions unless you have some reason not to. If you do not use this flag, then your RegExps will not work well with text that includes emoji and other characters (including many Chinese characters) that require more than 16 bits. Without the `u` flag, the `“.”` character matches any 1 UTF-16 16-bit value. With the flag, however, `“.”` matches one Unicode codepoint, including those that have more than 16 bits. Setting the `u` flag on a RegExp also allows you to use the new `\u{...}` escape sequence for Unicode character and also enables the `\p{...}` notation for Unicode character classes.

y

The `y` flag indicates that the regular expression is “sticky” and should match at the beginning of a string or at the first character following the previous match. When used with a regular expression that is designed to find a single match, it effectively treats that regular expression as if it begins with `^` to anchor it to the beginning of the string. This flag is more useful with regular expressions that are used repeatedly to find all matches within a string. In this case, it causes special behavior of the String `match()` method and the RegExp `exec()` method to enforce that each subsequent match is anchored to the string position at which the last one ended.

These flags may be specified in any combination and in any order. For example, if you want your regular expression to be Unicode-aware to do case-insensitive matching and you intend to use it to find multiple matches within a string, you would specify the flags `uig`, `gui`, or any other permutation of these three letters.

### 11.3.2 String Methods for Pattern Matching

Until now, we have been describing the grammar used to define regular expressions, but not explaining how those regular expressions can actually be used

in JavaScript code. We are now switching to cover the API for using RegExp objects. This section begins by explaining the string methods that use regular expressions to perform pattern matching and search-and-replace operations. The sections that follow this one continue the discussion of pattern matching with JavaScript regular expressions by discussing the RegExp object and its methods and properties.

## search()

Strings support four methods that use regular expressions. The simplest is `search()`. This method takes a regular expression argument and returns either the character position of the start of the first matching substring or `-1` if there is no match:

```
"JavaScript".search(/script/ui)  // => 4
"Python".search(/script/ui)      // => -1
```

If the argument to `search()` is not a regular expression, it is first converted to one by passing it to the `RegExp` constructor. `search()` does not support global searches; it ignores the `g` flag of its regular expression argument.

## replace()

The `replace()` method performs a search-and-replace operation. It takes a regular expression as its first argument and a replacement string as its second argument. It searches the string on which it is called for matches with the specified pattern. If the regular expression has the `g` flag set, the `replace()` method replaces all matches in the string with the replacement string; otherwise, it replaces only the first match it finds. If the first argument to `replace()` is a string rather than a regular expression, the method searches for that string literally rather than converting it to a regular expression with the `RegExp()` constructor, as `search()` does. As an example, you can use `replace()` as follows to provide uniform capitalization of the word “`JavaScript`” throughout a string of text:

```
// No matter how it is capitalized, replace it with the correct capitalization
text.replace(/javascript/gi, "JavaScript");
```

`replace()` is more powerful than this, however. Recall that parenthesized subexpressions of a regular expression are numbered from left to right and that the regular expression remembers the text that each subexpression matches. If a `$` followed by a digit appears in the replacement string, `replace()` replaces those two characters with the text that matches the specified subexpression. This is a very useful feature. You can use it, for example, to replace quotation marks in a string with other characters:

```
// A quote is a quotation mark, followed by any number of
// nonquotation mark characters (which we capture), followed
// by another quotation mark.
let quote = /"([""]*)"/g;
// Replace the straight quotation marks with guillemets
// leaving the quoted text (stored in $1) unchanged.
'He said "stop"'.replace(quote, '«$1»') // => 'He said «stop»'
```

If your `RegExp` uses named capture groups, then you can refer to the matching text by name rather than by number:

```
let quote = /"(?<quotedText>[""]*)"/g;
'He said "stop"'.replace(quote, '«$<quotedText>»') // => 'He said «stop»'
```

Instead of passing a replacement string as the second argument to `replace()`, you can also pass a function that will be invoked to compute the replacement value. The replacement function is invoked with a number of arguments. First is the entire matched text. Next, if the RegExp has capturing groups, then the substrings that were captured by those groups are passed as arguments. The next argument is the position within the string at which the match was found. After that, the entire string that `replace()` was called on is passed. And finally, if the RegExp contained any named capture groups, the last argument to the replacement function is an object whose property names match the capture group names and whose values are the matching text. As an example, here is code that uses a replacement function to convert decimal integers in a string to hexadecimal:

```
let s = "15 times 15 is 225";
s.replace(/\d+/gu, n => parseInt(n).toString(16)) // => "f times f is e1"
```

## match()

The `match()` method is the most general of the String regular expression methods. It takes a regular expression as its only argument (or converts its argument to a regular expression by passing it to the `RegExp()` constructor) and returns an array that contains the results of the match, or `null` if no match is found. If the regular expression has the `g` flag set, the method returns an array of all matches that appear in the string. For example:

```
"7 plus 8 equals 15".match(/\d+/g) // => ["7", "8", "15"]
```

If the regular expression does not have the `g` flag set, `match()` does not do a global search; it simply searches for the first match. In this nonglobal case, `match()` still returns an array, but the array elements are completely different. Without the `g` flag, the first element of the returned array is the matching string, and any remaining elements are the substrings matching the parenthesized capturing groups of the regular expression. Thus, if `match()` returns an array `a`, `a[0]` contains the complete match, `a[1]` contains the substring that matched the first parenthesized expression, and so on. To draw a parallel with the `replace()` method, `a[1]` is the same string as `$1`, `a[2]` is the same as `$2`, and so on.

For example, consider parsing a URL<sup>5</sup> with the following code:

```
// A very simple URL parsing RegExp
let url = /(\w+):\/\/([\w.]+)/\((\S*)/;
let text = "Visit my blog at http://www.example.com/~david";
let match = text.match(url);
let fullurl, protocol, host, path;
if (match !== null) {
  fullurl = match[0]; // fullurl == "http://www.example.com/~david"
  protocol = match[1]; // protocol == "http"
  host = match[2]; // host == "www.example.com"
  path = match[3]; // path == "~david"
}
```

In this non-global case, the array returned by `match()` also has some object properties in addition to the numbered array elements. The `input` property refers to the string on which `match()` was called. The `index` property is the position within that string at which the match starts. And if the regular expression contains named capture groups, then the returned array also has a `groups` property whose value is an object. The properties of this object match the names of the named groups, and the values are the matching text. We could rewrite the previous URL parsing example, for example, like this:

```
let url = /(?<protocol>\w+):\/\/(?<host>[\w.]+)\/(?<path>\S*)/;
let text = "Visit my blog at http://www.example.com/~david";
```

```
let match = text.match(url);
match[0]           // => "http://www.example.com/~david"
match.input         // => text
match.index         // => 17
match.groups.protocol // => "http"
match.groups.host    // => "www.example.com"
match.groups.path     // => "~david"
```

We've seen that `match()` behaves quite differently depending on whether the `RegExp` has the `g` flag set or not. There are also important but less dramatic differences in behavior when the `y` flag is set. Recall that the `y` flag makes a regular expression "sticky" by constraining where in the string matches can begin. If a `RegExp` has both the `g` and `y` flags set, then `match()` returns an array of matched strings, just as it does when `g` is set without `y`. But the first match must begin at the start of the string, and each subsequent match must begin at the character immediately following the previous match.

If the `y` flag is set without `g`, then `match()` tries to find a single match, and, by default, this match is constrained to the start of the string. You can change this default match start position, however, by setting the `lastIndex` property of the `RegExp` object at the index at which you want to match at. If a match is found, then this `lastIndex` will be automatically updated to the first character after the match, so if you call `match()` again, in this case, it will look for a subsequent match. (`lastIndex` may seem like a strange name for a property that specifies the position at which to begin the *next* match. We will see it again when we cover the `RegExp exec()` method, and its name may make more sense in that context.)

```
let vowel = /[aeiou]/y; // Sticky vowel match
"test".match(vowel)   // => null: "test" does not begin with a vowel
vowel.lastIndex = 1;   // Specify a different match position
"test".match(vowel)[0] // => "e": we found a vowel at position 1
vowel.lastIndex       // => 2: lastIndex was automatically updated
"test".match(vowel)   // => null: no vowel at position 2
vowel.lastIndex       // => 0: lastIndex gets reset after failed match
```

It is worth noting that passing a non-global regular expression to the `match()` method of a string is the same as passing the string to the `exec()` method of the regular expression: the returned array and its properties are the same in both cases.

## matchAll()

The `matchAll()` method is defined in ES2020, and as of early 2020 is implemented by modern web browsers and Node. `matchAll()` expects a `RegExp` with the `g` flag set. Instead of returning an array of matching substrings like `match()` does, however, it returns an iterator that yields the kind of match objects that `match()` returns when used with a non-global `RegExp`. This makes `matchAll()` the easiest and most general way to loop through all matches within a string.

You might use `matchAll()` to loop through the words in a string of text like this:

```
// One or more Unicode alphabetic characters between word boundaries
const words = /\b\p{Alphabetic}+\b/gu; // \p is not supported in Firefox yet
const text = "This is a naïve test of the matchAll() method.";
for(let word of text.matchAll(words)) {
    console.log(`Found '${word[0]}' at index ${word.index}.`);
}
```

You can set the `lastIndex` property of a `RegExp` object to tell `matchAll()` what index in the string to begin matching at. Unlike the other pattern-matching methods, however, `matchAll()` never modifies the `lastIndex` property of the `RegExp` you call it on, and this makes it much less likely to cause bugs in your code.

## split()

The last of the regular expression methods of the String object is `split()`. This method breaks the string on which it is called into an array of substrings, using the argument as a separator. It can be used with a string argument like this:

```
"123,456,789".split(",") // => ["123", "456", "789"]
```

The `split()` method can also take a regular expression as its argument, and this allows you to specify more general separators. Here we call it with a separator that includes an arbitrary amount of whitespace on either side:

```
"1, 2, 3,\n4, 5".split(/\s*,\s*/) // => ["1", "2", "3", "4", "5"]
```

Surprisingly, if you call `split()` with a RegExp delimiter and the regular expression includes capturing groups, then the text that matches the capturing groups will be included in the returned array. For example:

```
const htmlTag = /<([^\>]+)>/; // < followed by one or more non->, followed by >
"Testing<br/>1,2,3".split(htmlTag) // => ["Testing", "br/", "1,2,3"]
```

### 11.3.3 The RegExp Class

This section documents the `RegExp()` constructor, the properties of `RegExp` instances, and two important pattern-matching methods defined by the `RegExp` class.

The `RegExp()` constructor takes one or two string arguments and creates a new `RegExp` object. The first argument to this constructor is a string that contains the body of the regular expression—the text that would appear within slashes in a regular-expression literal. Note that both string literals and regular expressions use the \ character for escape sequences, so when you pass a regular expression to `RegExp()` as a string literal, you must replace each \ character with \\. The second argument to `RegExp()` is optional. If supplied, it indicates the regular expression flags. It should be g, i, m, s, u, y, or any combination of those letters.

For example:

```
// Find all five-digit numbers in a string. Note the double \\ in this case.
let zipcode = new RegExp("\\d{5}", "g");
```

The `RegExp()` constructor is useful when a regular expression is being dynamically created and thus cannot be represented with the regular expression literal syntax. For example, to search for a string entered by the user, a regular expression must be created at runtime with `RegExp()`.

Instead of passing a string as the first argument to `RegExp()`, you can also pass a `RegExp` object. This allows you to copy a regular expression and change its flags:

```
let exactMatch = /JavaScript/;
let caseInsensitive = new RegExp(exactMatch, "i");
```

## RegExp properties

`RegExp` objects have the following properties:

`source`

This read-only property is the source text of the regular expression: the characters that appear between the slashes in a RegExp literal.

### flags

This read-only property is a string that specifies the set of letters that represent the flags for the RegExp.

### global

A read-only boolean property that is true if the g flag is set.

### ignoreCase

A read-only boolean property that is true if the i flag is set.

### multiline

A read-only boolean property that is true if the m flag is set.

### dotAll

A read-only boolean property that is true if the s flag is set.

### unicode

A read-only boolean property that is true if the u flag is set.

### sticky

A read-only boolean property that is true if the y flag is set.

### lastIndex

This property is a read/write integer. For patterns with the g or y flags, it specifies the character position at which the next search is to begin. It is used by the exec() and test() methods, described in the next two subsections.

## test()

The test() method of the RegExp class is the simplest way to use a regular expression. It takes a single string argument and returns true if the string matches the pattern or false if it does not match.

test() works by simply calling the (much more complicated) exec() method described in the next section and returning true if exec() returns a non-null value. Because of this, if you use test() with a RegExp that uses the g or y flags, then its behavior depends on the value of the lastIndex property of the RegExp object, which can change unexpectedly. See [“The lastIndex Property and RegExp Reuse”](#) for more details.

## exec()

The RegExp exec() method is the most general and powerful way to use regular expressions. It takes a single string argument and looks for a match in that string. If no match is found, it returns null. If a match is found, however, it returns an array just like the array returned by the match() method for non-global searches. Element 0 of the array contains the string that matched the regular expression, and any subsequent array elements contain the substrings that matched any capturing

groups. The returned array also has named properties: the `index` property contains the character position at which the match occurred, and the `input` property specifies the string that was searched, and the `groups` property, if defined, refers to an object that holds the substrings matching the any named capturing groups.

Unlike the String `match()` method, `exec()` returns the same kind of array whether or not the regular expression has the global `g` flag. Recall that `match()` returns an array of matches when passed a global regular expression. `exec()`, by contrast, always returns a single match and provides complete information about that match. When `exec()` is called on a regular expression that has either the global `g` flag or the sticky `y` flag set, it consults the `lastIndex` property of the `RegExp` object to determine where to start looking for a match. (And if the `y` flag is set, it also constrains the match to begin at that position.) For a newly created `RegExp` object, `lastIndex` is 0, and the search begins at the start of the string. But each time `exec()` successfully finds a match, it updates the `lastIndex` property to the index of the character immediately after the matched text. If `exec()` fails to find a match, it resets `lastIndex` to 0. This special behavior allows you to call `exec()` repeatedly in order to loop through all the regular expression matches in a string. (Although, as we've described, in ES2020 and later, the `matchAll()` method of `String` is an easier way to loop through all matches.) For example, the loop in the following code will run twice:

```
let pattern = /Java/g;
let text = "JavaScript > Java";
let match;
while((match = pattern.exec(text)) !== null) {
  console.log(`Matched ${match[0]} at ${match.index}`);
  console.log(`Next search begins at ${pattern.lastIndex}`);
}
```

### The `lastIndex` Property and `RegExp` Reuse

As you have seen already, JavaScript's regular expression API is complicated. The use of the `lastIndex` property with the `g` and `y` flags is a particularly awkward part of this API. When you use these flags, you need to be particularly careful when calling the `match()`, `exec()`, or `test()` methods because the behavior of these methods depends on `lastIndex`, and the value of `lastIndex` depends on what you have previously done with the `RegExp` object. This makes it easy to write buggy code.

Suppose, for example, that we wanted to find the index of all `<p>` tags within a string of HTML text. We might write code like this:

```
let match, positions = [];
while((match = /<p>/g.exec(html)) !== null) { // POSSIBLE INFINITE LOOP
  positions.push(match.index);
}
```

This code does not do what we want it to. If the `html` string contains at least one `<p>` tag, then it will loop forever. The problem is that we use a `RegExp` literal in the `while` loop condition. For each iteration of the loop, we're creating a new `RegExp` object with `lastIndex` set to 0, so `exec()` always begins at the start of the string, and if there is a match, it will keep matching over and over. The solution, of course, is to define the `RegExp` once, and save it to a variable so that we're using the same `RegExp` object for each iteration of the loop.

On the other hand, sometimes reusing a `RegExp` object is the wrong thing to do. Suppose, for example, that we want to loop through all of the words in a dictionary to find words that contain pairs of double letters:

```

let dictionary = [ "apple", "book", "coffee" ];
let doubleLetterWords = [];
let doubleLetter = /(\w)\1/g;

for(let word of dictionary) {
    if (doubleLetter.test(word)) {
        doubleLetterWords.push(word);
    }
}
doubleLetterWords // => ["apple", "coffee"]: "book" is missing!

```

Because we set the `g` flag on the `RegExp`, the `lastIndex` property is changed after successful matches, and the `test()` method (which is based on `exec()`) starts searching for a match at the position specified by `lastIndex`. After matching the “pp” in “apple”, `lastIndex` is 3, and so we start searching the word “book” at position 3 and do not see the “oo” that it contains.

We could fix this problem by removing the `g` flag (which is not actually necessary in this particular example), or by moving the `RegExp` literal into the body of the loop so that it is re-created on each iteration, or by explicitly resetting `lastIndex` to zero before each call to `test()`.

The moral here is that `lastIndex` makes the `RegExp` API error prone. So be extra careful when using the `g` or `y` flags and looping. And in ES2020 and later, use the `String.matchAll()` method instead of `exec()` to sidestep this problem since `matchAll()` does not modify `lastIndex`.

## 11.4 Dates and Times

The `Date` class is JavaScript’s API for working with dates and times. Create a `Date` object with the `Date()` constructor. With no arguments, it returns a `Date` object that represents the current date and time:

```
let now = new Date(); // The current time
```

If you pass one numeric argument, the `Date()` constructor interprets that argument as the number of milliseconds since the 1970 epoch:

```
let epoch = new Date(0); // Midnight, January 1st, 1970, GMT
```

If you specify two or more integer arguments, they are interpreted as the year, month, day-of-month, hour, minute, second, and millisecond in your local time zone, as in the following:

```
let century = new Date(2100,          // Year 2100
                      0,            // January
                      1,            // 1st
                      2, 3, 4, 5); // 02:03:04.005, local time
```

One quirk of the `Date` API is that the first month of a year is number 0, but the first day of a month is number 1. If you omit the time fields, the `Date()` constructor defaults them all to 0, setting the time to midnight.

Note that when invoked with multiple numbers, the `Date()` constructor interprets them using whatever time zone the local computer is set to. If you want to specify a date and time in UTC (Universal Coordinated Time, aka GMT), then you can use the `Date.UTC()`. This static method takes the same arguments as the `Date()` constructor, interprets them in UTC, and returns a millisecond timestamp that you can pass to the `Date()` constructor:

```
// Midnight in England, January 1, 2100
let century = new Date(Date.UTC(2100, 0, 1));
```

If you print a date (with `console.log(century)`, for example), it will, by default, be printed in your local time zone. If you want to display a date in UTC, you should explicitly convert it to a string with `toUTCString()` or `toISOString()`.

Finally, if you pass a string to the `Date()` constructor, it will attempt to parse that string as a date and time specification. The constructor can parse dates specified in the formats produced by the `toString()`, `toUTCString()`, and `toISOString()` methods:

```
let century = new Date("2100-01-01T00:00:00Z"); // An ISO format date
```

Once you have a `Date` object, various get and set methods allow you to query and modify the year, month, day-of-month, hour, minute, second, and millisecond fields of the `Date`. Each of these methods has two forms: one that gets or sets using local time and one that gets or sets using UTC time. To get or set the year of a `Date` object, for example, you would use `getFullYear()`, `getUTCFullYear()`, `setFullYear()`, or `setUTCFullYear()`:

```
let d = new Date(); // Start with the current date
d.setFullYear(d.getFullYear() + 1); // Increment the year
```

To get or set the other fields of a `Date`, replace “`FullYear`” in the method name with “`Month`”, “`Date`”, “`Hours`”, “`Minutes`”, “`Seconds`”, or “`Milliseconds`”. Some of the date set methods allow you to set more than one field at a time.

`setFullYear()` and `setUTCFullYear()` also optionally allow you to set the month and day-of-month as well. And `setHours()` and `setUTCHours()` allow you to specify the minutes, seconds, and milliseconds fields in addition to the hours field.

Note that the methods for querying the day-of-month are `getDate()` and `getUTCDate()`. The more natural-sounding functions `getDay()` and `getUTCDay()` return the day-of-week (0 for Sunday through 6 for Saturday). The day-of-week is read-only, so there is not a corresponding `setDay()` method.

## 11.4.1 Timestamps

JavaScript represents dates internally as integers that specify the number of milliseconds since (or before) midnight on January 1, 1970, UTC time. Integers as large as 8,640,000,000,000,000 are supported, so JavaScript won’t be running out of milliseconds for more than 270,000 years.

For any `Date` object, the `getTime()` method returns this internal value, and the  `setTime()` method sets it. So you can add 30 seconds to a `Date` with code like this, for example:

```
d.setTime(d.getTime() + 30000);
```

These millisecond values are sometimes called *timestamps*, and it is sometimes useful to work with them directly rather than with `Date` objects. The static `Date.now()` method returns the current time as a timestamp and is helpful when you want to measure how long your code takes to run:

```
let startTime = Date.now();
reticulateSplines(); // Do some time-consuming operation
let endTime = Date.now();
console.log(`Spline reticulation took ${endTime - startTime}ms.`);
```

### High-Resolution Timestamps

The timestamps returned by `Date.now()` are measured in milliseconds. A millisecond is actually a relatively long time for a computer, and sometimes you may want to measure elapsed time with higher precision. The `performance.now()` function allows this: it also returns a millisecond-based timestamp, but the return value is not an integer, so it includes fractions of a millisecond. The value returned by `performance.now()` is not an absolute timestamp like the `Date.now()` value is. Instead, it simply indicates how much time has elapsed since a web page was loaded or since the Node process started.

The `performance` object is part of a larger Performance API that is not defined by the ECMAScript standard but is implemented by web browsers and by Node. In order to use the `performance` object in Node, you must import it with:

```
const { performance } = require("perf_hooks");
```

Allowing high-precision timing on the web may allow unscrupulous websites to fingerprint visitors, so browsers (notably Firefox) may reduce the precision of `performance.now()` by default. As a web developer, you should be able to re-enable high-precision timing somehow (such as by setting `privacy.reduceTimerPrecision` to false in Firefox).

## 11.4.2 Date Arithmetic

Date objects can be compared with JavaScript's standard `<`, `<=`, `>`, and `>=` comparison operators. And you can subtract one Date object from another to determine the number of milliseconds between the two dates. (This works because the Date class defines a `valueOf()` method that returns a timestamp.)

If you want to add or subtract a specified number of seconds, minutes, or hours from a Date, it is often easiest to simply modify the timestamp as demonstrated in the previous example, when we added 30 seconds to a date. This technique becomes more cumbersome if you want to add days, and it does not work at all for months and years since they have varying numbers of days. To do date arithmetic involving days, months, and years, you can use `setDate()`, `setMonth()`, and `setYear()`. Here, for example, is code that adds three months and two weeks to the current date:

```
let d = new Date();
d.setMonth(d.getMonth() + 3, d.getDate() + 14);
```

Date setting methods work correctly even when they overflow. When we add three months to the current month, we can end up with a value greater than 11 (which represents December). The `setMonth()` handles this by incrementing the year as needed. Similarly, when we set the day of the month to a value larger than the number of days in the month, the month gets incremented appropriately.

## 11.4.3 Formatting and Parsing Date Strings

If you are using the Date class to actually keep track of dates and times (as opposed to just measuring time intervals), then you are likely to need to display dates and times to the users of your code. The Date class defines a number of different methods for converting Date objects to strings. Here are some examples:

```
let d = new Date(2020, 0, 1, 17, 10, 30); // 5:10:30pm on New Year's Day 2020
d.toString() // => "Wed Jan 01 2020 17:10:30 GMT-0800 (Pacific Standard Time)"
d.toUTCString() // => "Thu, 02 Jan 2020 01:10:30 GMT"
d.toLocaleDateString() // => "1/1/2020": 'en-US' locale
d.toLocaleTimeString() // => "5:10:30 PM": 'en-US' locale
d.toISOString() // => "2020-01-02T01:10:30.000Z"
```

This is a full list of the string formatting methods of the Date class:

`toString()`

This method uses the local time zone but does not format the date and time in a locale-aware way.

`toUTCString()`

This method uses the UTC time zone but does not format the date in a locale-aware way.

`toISOString()`

This method prints the date and time in the standard year-month-day hours:minutes:seconds.ms format of the ISO-8601 standard. The letter “T” separates the date portion of the output from the time portion of the output. The time is expressed in UTC, and this is indicated with the letter “Z” as the last letter of the output.

`toLocaleString()`

This method uses the local time zone and a format that is appropriate for the user’s locale.

`toDateString()`

This method formats only the date portion of the Date and omits the time. It uses the local time zone and does not do locale-appropriate formatting.

`toLocaleDateString()`

This method formats only the date. It uses the local time zone and a locale-appropriate date format.

`toTimeString()`

This method formats only the time and omits the date. It uses the local time zone but does not format the time in a locale-aware way.

`toLocaleTimeString()`

This method formats the time in a locale-aware way and uses the local time zone.

None of these date-to-string methods is ideal when formatting dates and times to be displayed to end users. See [\\$11.7.2](#) for a more general-purpose and locale-aware date- and time-formatting technique.

Finally, in addition to these methods that convert a Date object to a string, there is also a static `Date.parse()` method that takes a string as its argument, attempts to parse it as a date and time, and returns a timestamp representing that date.

`Date.parse()` is able to parse the same strings that the `Date()` constructor can and is guaranteed to be able to parse the output of `toISOString()`, `toUTCString()`, and `toString()`.

## 11.5 Error Classes

The JavaScript `throw` and `catch` statements can throw and catch any JavaScript value, including primitive values. There is no exception type that must be used to signal errors. JavaScript does define an `Error` class, however, and it is traditional to use instances of `Error` or a subclass when signaling an error with `throw`. One good reason to use an `Error` object is that, when you create an `Error`, it captures the state of the JavaScript stack, and if the exception is uncaught, the stack trace will be displayed with the error message, which will help you debug the issue. (Note that the stack trace shows where the `Error` object was created, not where the `throw` statement throws it. If you always create the object right before throwing it with `throw new Error()`, this will not cause any confusion.)

Error objects have two properties: `message` and `name`, and a `toString()` method. The value of the `message` property is the value you passed to the `Error()` constructor, converted to a string if necessary. For error objects created with `Error()`, the `name` property is always “`Error`”. The `toString()` method simply returns the value of the `name` property followed by a colon and space and the value of the `message` property.

Although it is not part of the ECMAScript standard, Node and all modern browsers also define a `stack` property on `Error` objects. The value of this property is a multi-line string that contains a stack trace of the JavaScript call stack at the moment that the `Error` object was created. This can be useful information to log when an unexpected error is caught.

In addition to the `Error` class, JavaScript defines a number of subclasses that it uses to signal particular types of errors defined by ECMAScript. These subclasses are `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, and `URIError`. You can use these error classes in your own code if they seem appropriate. Like the base `Error` class, each of these subclasses has a constructor that takes a single `message` argument. And instances of each of these subclasses have a `name` property whose value is the same as the constructor name.

You should feel free to define your own `Error` subclasses that best encapsulate the error conditions of your own program. Note that you are not limited to the `name` and `message` properties. If you create a subclass, you can define new properties to provide error details. If you are writing a parser, for example, you might find it useful to define a `ParseError` class with `line` and `column` properties that specify the exact location of the parsing failure. Or if you are working with HTTP requests, you might want to define an `HTTPError` class that has a `status` property that holds the HTTP status code (such as 404 or 500) of the failed request.

For example:

```
class HTTPError extends Error {
  constructor(status, statusText, url) {
    super(`${status} ${statusText}: ${url}`);
    this.status = status;
    this.statusText = statusText;
    this.url = url;
  }
  get name() { return "HTTPError"; }
}

let error = new HTTPError(404, "Not Found", "http://example.com/");
error.status      // => 404
error.message    // => "404 Not Found: http://example.com/"
error.name       // => "HTTPError"
```

# 11.6 JSON Serialization and Parsing

When a program needs to save data or needs to transmit data across a network connection to another program, it must convert its in-memory data structures into a string of bytes or characters than can be saved or transmitted and then later be parsed to restore the original in-memory data structures. This process of converting data structures into streams of bytes or characters is known as *serialization* (or *marshaling* or even *pickling*).

The easiest way to serialize data in JavaScript uses a serialization format known as JSON. This acronym stands for “JavaScript Object Notation” and, as the name implies, the format uses JavaScript object and array literal syntax to convert data structures consisting of objects and arrays into strings. JSON supports primitive numbers and strings and also the values `true`, `false`, and `null`, as well as arrays and objects built up from those primitive values. JSON does not support other JavaScript types like `Map`, `Set`, `RegExp`, `Date`, or typed arrays. Nevertheless, it has proved to be a remarkably versatile data format and is in common use even with non-JavaScript-based programs.

JavaScript supports JSON serialization and deserialization with the two functions `JSON.stringify()` and `JSON.parse()`, which were covered briefly in §6.8. Given an object or array (nested arbitrarily deeply) that does not contain any nonserializable values like `RegExp` objects or typed arrays, you can serialize the object simply by passing it to `JSON.stringify()`. As the name implies, the return value of this function is a string. And given a string returned by `JSON.stringify()`, you can re-create the original data structure by passing the string to `JSON.parse()`:

```
let o = {s: "", n: 0, a: [true, false, null]};
let s = JSON.stringify(o); // s == '{"s":"","n":0,"a":[true,false,null]}'
let copy = JSON.parse(s); // copy == {s: "", n: 0, a: [true, false, null]}
```

If we leave out the part where serialized data is saved to a file or sent over the network, we can use this pair of functions as a somewhat inefficient way of creating a deep copy of an object:

```
// Make a deep copy of any serializable object or array
function deepcopy(o) {
    return JSON.parse(JSON.stringify(o));
}
```

## JSON Is a Subset of JavaScript

When data is serialized to JSON format, the result is valid JavaScript source code for an expression that evaluates to a copy of the original data structure. If you prefix a JSON string with `var data =` and pass the result to `eval()`, you’ll get a copy of the original data structure assigned to the variable `data`. You should never do this, however, because it is a huge security hole—if an attacker could inject arbitrary JavaScript code into a JSON file, they could make your program run their code. It is faster and safer to just use `JSON.parse()` to decode JSON-formatted data.

JSON is sometimes used as a human-readable configuration file format. If you find yourself hand-editing a JSON file, note that the JSON format is a very strict subset of JavaScript. Comments are not allowed and property names must be enclosed in double quotes even when JavaScript would not require this.

Typically, you pass only a single argument to `JSON.stringify()` and `JSON.parse()`. Both functions accept an optional second argument that allows us to extend the JSON format, and these are described next. `JSON.stringify()` also takes an optional third argument that we'll discuss first. If you would like your JSON-formatted string to be human-readable (if it is being used as a configuration file, for example), then you should pass `null` as the second argument and pass a number or string as the third argument. This third argument tells `JSON.stringify()` that it should format the data on multiple indented lines. If the third argument is a number, then it will use that number of spaces for each indentation level. If the third argument is a string of whitespace (such as '`\t`'), it will use that string for each level of indent.

```
let o = {s: "test", n: 0};
JSON.stringify(o, null, 2) // => '{\n  "s": "test",\n  "n": 0\n}'
```

`JSON.parse()` ignores whitespace, so passing a third argument to `JSON.stringify()` has no impact on our ability to convert the string back into a data structure.

## 11.6.1 JSON Customizations

If `JSON.stringify()` is asked to serialize a value that is not natively supported by the JSON format, it looks to see if that value has a `toJSON()` method, and if so, it calls that method and then stringifies the return value in place of the original value. Date objects implement `toJSON()`: it returns the same string that `toISOString()` method does. This means that if you serialize an object that includes a Date, the date will automatically be converted to a string for you. When you parse the serialized string, the re-created data structure will not be exactly the same as the one you started with because it will have a string where the original object had a Date.

If you need to re-create Date objects (or modify the parsed object in any other way), you can pass a “reviver” function as the second argument to `JSON.parse()`. If specified, this “reviver” function is invoked once for each primitive value (but not the objects or arrays that contain those primitive values) parsed from the input string. The function is invoked with two arguments. The first is a property name—either an object property name or an array index converted to a string. The second argument is the primitive value of that object property or array element. Furthermore, the function is invoked as a method of the object or array that contains the primitive value, so you can refer to that containing object with the `this` keyword.

The return value of the reviver function becomes the new value of the named property. If it returns its second argument, the property will remain unchanged. If it returns `undefined`, then the named property will be deleted from the object or array before `JSON.parse()` returns to the user.

As an example, here is a call to `JSON.parse()` that uses a reviver function to filter some properties and to re-create Date objects:

```
let data = JSON.parse(text, function(key, value) {
  // Remove any values whose property name begins with an underscore
  if (key[0] === "_") return undefined;

  // If the value is a string in ISO 8601 date format convert it to a Date.
  if (typeof value === "string" &&
    /^[\d\d\d\d-\d\d-\d\dT\d\d:\d\d:\d\d.\d\d\dZ$/.test(value)) {
    return new Date(value);
}
```

```
// Otherwise, return the value unchanged
return value;
});
```

In addition to its use of `toJSON()` described earlier, `JSON.stringify()` also allows its output to be customized by passing an array or a function as the optional second argument.

If an array of strings (or numbers—they are converted to strings) is passed instead as the second argument, these are used as the names of object properties (or array elements). Any property whose name is not in the array will be omitted from stringification. Furthermore, the returned string will include properties in the same order that they appear in the array (which can be very useful when writing tests).

If you pass a function, it is a replacer function—effectively the inverse of the optional reviver function you can pass to `JSON.parse()`. If specified, the replacer function is invoked for each value to be stringified. The first argument to the replacer function is the object property name or array index of the value within that object, and the second argument is the value itself. The replacer function is invoked as a method of the object or array that contains the value to be stringified. The return value of the replacer function is stringified in place of the original value. If the replacer returns `undefined` or returns nothing at all, then that value (and its array element or object property) is omitted from the stringification.

```
// Specify what fields to serialize, and what order to serialize them in
let text = JSON.stringify(address, ["city", "state", "country"]);

// Specify a replacer function that omits RegExp-value properties
let json = JSON.stringify(o, (k, v) => v instanceof RegExp ? undefined : v);
```

The two `JSON.stringify()` calls here use the second argument in a benign way, producing serialized output that can be deserialized without requiring a special reviver function. In general, though, if you define a `toJSON()` method for a type, or if you use a replacer function that actually replaces nonserializable values with serializable ones, then you will typically need to use a custom reviver function with `JSON.parse()` to get your original data structure back. If you do this, you should understand that you are defining a custom data format and sacrificing portability and compatibility with a large ecosystem of JSON-compatible tools and languages.

## 11.7 The Internationalization API

The JavaScript internationalization API consists of the three classes `Intl.NumberFormat`, `Intl.DateTimeFormat`, and `Intl.Collator` that allow us to format numbers (including monetary amounts and percentages), dates, and times in locale-appropriate ways and to compare strings in locale-appropriate ways. These classes are not part of the ECMAScript standard but are defined as part of the [ECMA402 standard](#) and are well-supported by web browsers. The `Intl` API is also supported in Node, but at the time of this writing, prebuilt Node binaries do not ship with the localization data required to make them work with locales other than US English. So in order to use these classes with Node, you may need to download a separate data package or use a custom build of Node.

One of the most important parts of internationalization is displaying text that has been translated into the user's language. There are various ways to achieve this, but none of them are within the scope of the `Intl` API described here.

## 11.7.1 Formatting Numbers

Users around the world expect numbers to be formatted in different ways. Decimal points can be periods or commas. Thousands separators can be commas or periods, and they aren't used every three digits in all places. Some currencies are divided into hundredths, some into thousandths, and some have no subdivisions. Finally, although the so-called "Arabic numerals" 0 through 9 are used in many languages, this is not universal, and users in some countries will expect to see numbers written using the digits from their own scripts.

The `Intl.NumberFormat` class defines a `format()` method that takes all of these formatting possibilities into account. The constructor takes two arguments. The first argument specifies the locale that the number should be formatted for and the second is an object that specifies more details about how the number should be formatted. If the first argument is omitted or `undefined`, then the system locale (which we assume to be the user's preferred locale) will be used. If the first argument is a string, it specifies a desired locale, such as "`en-US`" (English as used in the United States), "`fr`" (French), or "`zh-Hans-CN`" (Chinese, using the simplified Han writing system, in China). The first argument can also be an array of locale strings, and in this case, `Intl.NumberFormat` will choose the most specific one that is well supported.

The second argument to the `Intl.NumberFormat()` constructor, if specified, should be an object that defines one or more of the following properties:

### `style`

Specifies the kind of number formatting that is required. The default is "`decimal`". Specify "`percent`" to format a number as a percentage or specify "`currency`" to specify a number as an amount of money.

### `currency`

If `style` is "`currency`", then this property is required to specify the three-letter ISO currency code (such as "`USD`" for US dollars or "`GBP`" for British pounds) of the desired currency.

### `currencyDisplay`

If `style` is "`currency`", then this property specifies how the currency is displayed. The default value "`symbol`" uses a currency symbol if the currency has one. The value "`code`" uses the three-letter ISO code, and the value "`name`" spells out the name of the currency in long form.

### `useGrouping`

Set this property to `false` if you do not want numbers to have thousands separators (or their locale-appropriate equivalents).

### `minimumIntegerDigits`

The minimum number of digits to use to display the integer part of the number. If the number has fewer digits than this, it will be padded on the left with zeros. The default value is 1, but you can use values as high as 21.

### `minimumFractionDigits`, `maximumFractionDigits`

These two properties control the formatting of the fractional part of the number. If a number has fewer fractional digits than the minimum, it will be padded with zeros on the right. If it has more than the maximum, then the

fractional part will be rounded. Legal values for both properties are between 0 and 20. The default minimum is 0 and the default maximum is 3, except when formatting monetary amounts, when the length of the fractional part varies depending on the specified currency.

```
minimumSignificantDigits, maximumSignificantDigits
```

These properties control the number of significant digits used when formatting a number, making them suitable when formatting scientific data, for example. If specified, these properties override the integer and fractional digit properties listed previously. Legal values are between 1 and 21.

Once you have created an `Intl.NumberFormat` object with the desired locale and options, you use it by passing a number to its `format()` method, which returns an appropriately formatted string. For example:

```
let euros = Intl.NumberFormat("es", {style: "currency", currency: "EUR"});
euros.format(10) // => "10,00 €": ten euros, Spanish formatting

let pounds = Intl.NumberFormat("en", {style: "currency", currency: "GBP"});
pounds.format(1000) // => "£1,000.00": One thousand pounds, English formatting
```

A useful feature of `Intl.NumberFormat` (and the other `Intl` classes as well) is that its `format()` method is bound to the `NumberFormat` object to which it belongs. So instead of defining a variable that refers to the formatting object and then invoking the `format()` method on that, you can just assign the `format()` method to a variable and use it as if it were a standalone function, as in this example:

```
let data = [0.05, .75, 1];
let formatData = Intl.NumberFormat(undefined, {
    style: "percent",
    minimumFractionDigits: 1,
    maximumFractionDigits: 1
}).format;

data.map(formatData) // => ["5.0%", "75.0%", "100.0%"]: in en-US locale
```

Some languages, such as Arabic, use their own script for decimal digits:

```
let arabic = Intl.NumberFormat("ar", {useGrouping: false}).format;
arabic(1234567890) // => "١٢٣٤٥٦٧٨٩٠"
```

Other languages, such as Hindi, use a script that has its own set of digits, but tend to use the ASCII digits 0–9 by default. If you want to override the default script used for digits, add `-u-nu-` to the locale and follow it with an abbreviated script name. You can format numbers with Indian-style grouping and Devanagari digits like this, for example:

```
let hindi = Intl.NumberFormat("hi-IN-u-nu-deva").format;
hindi(1234567890) // => "१,२३,४५,६७,८९०"
```

`-u-` in a locale specifies that what comes next is a Unicode extension. `nu` is the extension name for the numbering system, and `deva` is short for Devanagari. The `Intl` API standard defines names for a number of other numbering systems, mostly for the Indic languages of South and Southeast Asia.

## 11.7.2 Formatting Dates and Times

The `Intl.DateTimeFormat` class is a lot like the `Intl.NumberFormat` class. The `Intl.DateTimeFormat()` constructor takes the same two arguments that `Intl.NumberFormat()` does: a locale or array of locales and an object of formatting

options. And the way you use an `Intl.DateTimeFormat` instance is by calling its `format()` method to convert a `Date` object to a string.

As mentioned in §11.4, the `Date` class defines simple `toLocaleDateString()` and `toLocaleTimeString()` methods that produce locale-appropriate output for the user's locale. But these methods don't give you any control over what fields of the date and time are displayed. Maybe you want to omit the year but add a weekday to the date format. Do you want the month to be represented numerically or spelled out by name? The `Intl.DateTimeFormat` class provides fine-grained control over what is output based on the properties in the `options` object that is passed as the second argument to the constructor. Note, however, that `Intl.DateTimeFormat` cannot always display exactly what you ask for. If you specify options to format hours and seconds but omit minutes, you'll find that the formatter displays the minutes anyway. The idea is that you use the `options` object to specify what date and time fields you'd like to present to the user and how you'd like those formatted (by name or by number, for example), then the formatter will look for a locale-appropriate format that most closely matches what you have asked for.

The available options are the following. Only specify properties for date and time fields that you would like to appear in the formatted output.

#### `year`

Use "numeric" for a full, four-digit year or "2-digit" for a two-digit abbreviation.

#### `month`

Use "numeric" for a possibly short number like "1", or "2-digit" for a numeric representation that always has two digits, like "01". Use "long" for a full name like "January", "short" for an abbreviated name like "Jan", and "narrow" for a highly abbreviated name like "J" that is not guaranteed to be unique.

#### `day`

Use "numeric" for a one- or two-digit number or "2-digit" for a two-digit number for the day-of-month.

#### `weekday`

Use "long" for a full name like "Monday", "short" for an abbreviated name like "Mon", and "narrow" for a highly abbreviated name like "M" that is not guaranteed to be unique.

#### `era`

This property specifies whether a date should be formatted with an era, such as CE or BCE. This may be useful if you are formatting dates from very long ago or if you are using a Japanese calendar. Legal values are "long", "short", and "narrow".

#### `hour, minute, second`

These properties specify how you would like time displayed. Use "numeric" for a one- or two-digit field or "2-digit" to force single-digit numbers to be padded on the left with a 0.

#### `timeZone`

This property specifies the desired time zone for which the date should be formatted. If omitted, the local time zone is used. Implementations always recognize “UTC” and may also recognize Internet Assigned Numbers Authority (IANA) time zone names, such as “America/Los\_Angeles”.

#### timeZoneName

This property specifies how the time zone should be displayed in a formatted date or time. Use "long" for a fully spelled-out time zone name and "short" for an abbreviated or numeric time zone.

#### hour12

This boolean property specifies whether or not to use 12-hour time. The default is locale dependent, but you can override it with this property.

#### hourCycle

This property allows you to specify whether midnight is written as 0 hours, 12 hours, or 24 hours. The default is locale dependent, but you can override the default with this property. Note that hour12 takes precedence over this property. Use the value "h11" to specify that midnight is 0 and the hour before midnight is 11pm. Use "h12" to specify that midnight is 12. Use "h23" to specify that midnight is 0 and the hour before midnight is 23. And use "h24" to specify that midnight is 24.

Here are some examples:

```
let d = new Date("2020-01-02T13:14:15Z"); // January 2nd, 2020, 13:14:15 UTC

// With no options, we get a basic numeric date format
Intl.DateTimeFormat("en-US").format(d) // => "1/2/2020"
Intl.DateTimeFormat("fr-FR").format(d) // => "02/01/2020"

// Spelled out weekday and month
let opts = { weekday: "long", month: "long", year: "numeric", day: "numeric" };
Intl.DateTimeFormat("en-US", opts).format(d) // => "Thursday, January 2, 2020"
Intl.DateTimeFormat("es-ES", opts).format(d) // => "jueves, 2 de enero de 2020"

// The time in New York, for a French-speaking Canadian
opts = { hour: "numeric", minute: "2-digit", timeZone: "America/New_York" };
Intl.DateTimeFormat("fr-CA", opts).format(d) // => "8 h 14"
```

Intl.DateTimeFormat can display dates using calendars other than the default Julian calendar based on the Christian era. Although some locales may use a non-Christian calendar by default, you can always explicitly specify the calendar to use by adding -u-ca- to the locale and following that with the name of the calendar. Possible calendar names include “buddhist”, “chinese”, “coptic”, “ethiopic”, “gregory”, “hebrew”, “indian”, “islamic”, “iso8601”, “japanese”, and “persian”. Continuing the preceding example, we can determine the year in various non-Christian calendars:

```
let opts = { year: "numeric", era: "short" };
Intl.DateTimeFormat("en", opts).format(d) // => "2020 AD"
Intl.DateTimeFormat("en-u-ca-iso8601", opts).format(d) // => "2020 AD"
Intl.DateTimeFormat("en-u-ca-hebrew", opts).format(d) // => "5780 AM"
Intl.DateTimeFormat("en-u-ca-buddhist", opts).format(d) // => "2563 BE"
Intl.DateTimeFormat("en-u-ca-islamic", opts).format(d) // => "1441 AH"
Intl.DateTimeFormat("en-u-ca-persian", opts).format(d) // => "1398 AP"
Intl.DateTimeFormat("en-u-ca-indian", opts).format(d) // => "1941 Saka"
Intl.DateTimeFormat("en-u-ca-chinese", opts).format(d) // => "36 78"
Intl.DateTimeFormat("en-u-ca-japanese", opts).format(d) // => "2 Reiwa"
```

## 11.7.3 Comparing Strings

The problem of sorting strings into alphabetical order (or some more general “collation order” for nonalphabetical scripts) is more challenging than English speakers often realize. English uses a relatively small alphabet with no accented letters, and we have the benefit of a character encoding (ASCII, since incorporated into Unicode) whose numerical values perfectly match our standard string sort order. Things are not so simple in other languages. Spanish, for example treats ñ as a distinct letter that comes after n and before o. Lithuanian alphabetizes Y before J, and Welsh treats digraphs like CH and DD as single letters with CH coming after C and DD sorting after D.

If you want to display strings to a user in an order that they will find natural, it is not enough use the `sort()` method on an array of strings. But if you create an `Intl.Collator` object, you can pass the `compare()` method of that object to the `sort()` method to perform locale-appropriate sorting of the strings. `Intl.Collator` objects can be configured so that the `compare()` method performs case-insensitive comparisons or even comparisons that only consider the base letter and ignore accents and other diacritics.

Like `Intl.NumberFormat()` and `Intl.DateTimeFormat()`, the `Intl.Collator()` constructor takes two arguments. The first specifies a locale or an array of locales, and the second is an optional object whose properties specify exactly what kind of string comparison is to be done. The supported properties are these:

### usage

This property specifies how the collator object is to be used. The default value is "sort", but you can also specify "search". The idea is that, when sorting strings, you typically want a collator that differentiates as many strings as possible to produce a reliable ordering. But when comparing two strings, some locales may want a less strict comparison that ignores accents, for example.

### sensitivity

This property specifies whether the collator is sensitive to letter case and accents when comparing strings. The value "base" causes comparisons that ignore case and accents, considering only the base letter for each character. (Note, however, that some languages consider certain accented characters to be distinct base letters.) "accent" considers accents in comparisons but ignores case. "case" considers case and ignores accents. And "variant" performs strict comparisons that consider both case and accents. The default value for this property is "variant" when `usage` is "sort". If `usage` is "search", then the default sensitivity depends on the locale.

### ignorePunctuation

Set this property to `true` to ignore spaces and punctuation when comparing strings. With this property set to `true`, the strings “any one” and “anyone”, for example, will be considered equal.

### numeric

Set this property to `true` if the strings you are comparing are integers or contain integers and you want them to be sorted into numerical order instead of alphabetical order. With this option set, the string “Version 9” will be sorted before “Version 10”, for example.

## caseFirst

This property specifies which letter case should come first. If you specify "upper", then "A" will sort before "a". And if you specify "lower", then "a" will sort before "A". In either case, note that the upper- and lowercase variants of the same letter will be next to one another in sort order, which is different than Unicode lexicographic ordering (the default behavior of the Array `sort()` method) in which all ASCII uppercase letters come before all ASCII lowercase letters. The default for this property is locale dependent, and implementations may ignore this property and not allow you to override the case sort order.

Once you have created an `Intl.Collator` object for the desired locale and options, you can use its `compare()` method to compare two strings. This method returns a number. If the returned value is less than zero, then the first string comes before the second string. If it is greater than zero, then the first string comes after the second string. And if `compare()` returns zero, then the two strings are equal as far as this collator is concerned.

This `compare()` method that takes two strings and returns a number less than, equal to, or greater than zero is exactly what the `Array sort()` method expects for its optional argument. Also, `Intl.Collator` automatically binds the `compare()` method to its instance, so you can pass it directly to `sort()` without having to write a wrapper function and invoke it through the collator object. Here are some examples:

```
// A basic comparator for sorting in the user's locale.
// Never sort human-readable strings without passing something like this:
const collator = new Intl.Collator().compare;
["a", "z", "A", "Z"].sort(collator)      // => ["a", "A", "z", "Z"]

// Filenames often include numbers, so we should sort those specially
const filenameOrder = new Intl.Collator(undefined, { numeric: true }).compare;
["page10", "page9"].sort(filenameOrder)  // => ["page9", "page10"]

// Find all strings that loosely match a target string
const fuzzyMatcher = new Intl.Collator(undefined, {
    sensitivity: "base",
    ignorePunctuation: true
}).compare;
let strings = ["food", "fool", "Føø Bar"];
strings.findIndex(s => fuzzyMatcher(s, "foobar") === 0) // => 2
```

Some locales have more than one possible collation order. In Germany, for example, phone books use a slightly more phonetic sort order than dictionaries do. In Spain, before 1994, "ch" and "ll" were treated as separate letters, so that country now has a modern sort order and a traditional sort order. And in China, collation order can be based on character encodings, the base radical and strokes of each character, or on the Pinyin romanization of characters. These collation variants cannot be selected through the `Intl.Collator` options argument, but they can be selected by adding `-u-co-` to the locale string and adding the name of the desired variant. Use "de-DE-u-co-phonebk" for phone book ordering in Germany, for example, and "zh-TW-u-co-pinyin" for Pinyin ordering in Taiwan.

```
// Before 1994, CH and LL were treated as separate letters in Spain
const modernSpanish = Intl.Collator("es-ES").compare;
const traditionalSpanish = Intl.Collator("es-ES-u-co-trad").compare;
let palabras = ["luz", "llama", "como", "chico"];
palabras.sort(modernSpanish)      // => ["chico", "como", "llama", "luz"]
palabras.sort(traditionalSpanish) // => ["como", "chico", "luz", "llama"]
```

## 11.8 The Console API

You've seen the `console.log()` function used throughout this book: in web browsers, it prints a string in the "Console" tab of the browser's developer tools pane, which can be very helpful when debugging. In Node, `console.log()` is a general-purpose output function and prints its arguments to the process's `stdout` stream, where it typically appears to the user in a terminal window as program output.

The Console API defines a number of useful functions in addition to `console.log()`. The API is not part of any ECMAScript standard, but it is supported by browsers and by Node and has been formally written up and standardized at <https://console.spec.whatwg.org>.

The Console API defines the following functions:

`console.log()`

This is the most well-known of the console functions. It converts its arguments to strings and outputs them to the console. It includes spaces between the arguments and starts a new line after outputting all arguments.

`console.debug()`, `console.info()`, `console.warn()`, `console.error()`

These functions are almost identical to `console.log()`. In Node, `console.error()` sends its output to the `stderr` stream rather than the `stdout` stream, but the other functions are aliases of `console.log()`. In browsers, output messages generated by each of these functions may be prefixed by an icon that indicates its level or severity, and the developer console may also allow developers to filter console messages by level.

`console.assert()`

If the first argument is truthy (i.e., if the assertion passes), then this function does nothing. But if the first argument is `false` or another falsy value, then the remaining arguments are printed as if they had been passed to `console.error()` with an "Assertion failed" prefix. Note that, unlike typical `assert()` functions, `console.assert()` does not throw an exception when an assertion fails.

`console.clear()`

This function clears the console when that is possible. This works in browsers and in Node when Node is displaying its output to a terminal. If Node's output has been redirected to a file or a pipe, however, then calling this function has no effect.

`console.table()`

This function is a remarkably powerful but little-known feature for producing tabular output, and it is particularly useful in Node programs that need to produce output that summarizes data. `console.table()` attempts to display its argument in tabular form (although, if it can't do that, it displays it using regular `console.log()` formatting). This works best when the argument is a relatively short array of objects, and all of the objects in the array have the same (relatively small) set of properties. In this case, each object in the array is formatted as a row of the table, and each property is a column of the table. You can also pass an array of property names as an

optional second argument to specify the desired set of columns. If you pass an object instead of an array of objects, then the output will be a table with one column for property names and one column for property values. Or, if those property values are themselves objects, their property names will become columns in the table.

#### `console.trace()`

This function logs its arguments like `console.log()` does, and, in addition, follows its output with a stack trace. In Node, the output goes to `stderr` instead of `stdout`.

#### `console.count()`

This function takes a string argument and logs that string, followed by the number of times it has been called with that string. This can be useful when debugging an event handler, for example, if you need to keep track of how many times the event handler has been triggered.

#### `console.countReset()`

This function takes a string argument and resets the counter for that string.

#### `console.group()`

This function prints its arguments to the console as if they had been passed to `console.log()`, then sets the internal state of the console so that all subsequent console messages (until the next `console.groupEnd()` call) will be indented relative to the message that it just printed. This allows a group of related messages to be visually grouped with indentation. In web browsers, the developer console typically allows grouped messages to be collapsed and expanded as a group. The arguments to `console.group()` are typically used to provide an explanatory name for the group.

#### `console.groupCollapsed()`

This function works like `console.group()` except that in web browsers, the group will be “collapsed” by default and the messages it contains will be hidden unless the user clicks to expand the group. In Node, this function is a synonym for `console.group()`.

#### `console.groupEnd()`

This function takes no arguments. It produces no output of its own but ends the indentation and grouping caused by the most recent call to `console.group()` or `console.groupCollapsed()`.

#### `console.time()`

This function takes a single string argument, makes a note of the time it was called with that string, and produces no output.

#### `console.timeLog()`

This function takes a string as its first argument. If that string had previously been passed to `console.time()`, then it prints that string followed by the elapsed time since the `console.time()` call. If there are any additional arguments to `console.timeLog()`, they are printed as if they had been passed to `console.log()`.

#### `console.timeEnd()`

This function takes a single string argument. If that argument had previously been passed to `console.time()`, then it prints that argument and the elapsed time. After calling `console.timeEnd()`, it is no longer legal to call `console.timeLog()` without first calling `console.time()` again.

## 11.8.1 Formatted Output with Console

Console functions that print their arguments like `console.log()` have a little-known feature: if the first argument is a string that includes `%s`, `%i`, `%d`, `%f`, `%o`, `%0`, or `%c`, then this first argument is treated as format string,<sup>6</sup> and the values of subsequent arguments are substituted into the string in place of the two-character `%` sequences.

The meanings of the sequences are as follows:

`%s`

The argument is converted to a string.

`%i` and `%d`

The argument is converted to a number and then truncated to an integer.

`%f`

The argument is converted to a number

`%o` and `%0`

The argument is treated as an object, and property names and values are displayed. (In web browsers, this display is typically interactive, and users can expand and collapse properties to explore a nested data structure.) `%o` and `%0` both display object details. The uppercase variant uses an implementation-dependent output format that is judged to be most useful for software developers.

`%c`

In web browsers, the argument is interpreted as a string of CSS styles and used to style any text that follows (until the next `%c` sequence or the end of the string). In Node, the `%c` sequence and its corresponding argument are simply ignored.

Note that it is not often necessary to use a format string with the console functions: it is usually easy to obtain suitable output by simply passing one or more values (including objects) to the function and allowing the implementation to display them in a useful way. As an example, note that, if you pass an Error object to `console.log()`, it is automatically printed along with its stack trace.

## 11.9 URL APIs

Since JavaScript is so commonly used in web browsers and web servers, it is common for JavaScript code to need to manipulate URLs. The URL class parses URLs and also allows modification (adding search parameters or altering paths, for example) of existing URLs. It also properly handles the complicated topic of escaping and unescaping the various components of a URL.

The URL class is not part of any ECMAScript standard, but it works in Node and all internet browsers other than Internet Explorer. It is standardized at <https://url.spec.whatwg.org>.

Create a URL object with the `URL()` constructor, passing an absolute URL string as the argument. Or pass a relative URL as the first argument and the absolute URL that it is relative to as the second argument. Once you have created the URL object, its various properties allow you to query unescaped versions of the various parts of the URL:

```
let url = new URL("https://example.com:8000/path/name?q=term#fragment");
url.href      // => "https://example.com:8000/path/name?q=term#fragment"
url.origin    // => "https://example.com:8000"
url.protocol  // => "https:"
url.host      // => "example.com:8000"
url.hostname  // => "example.com"
url.port      // => "8000"
url.pathname  // => "/path/name"
url.search    // => "?q=term"
url.hash      // => "#fragment"
```

Although it is not commonly used, URLs can include a username or a password, and the URL class can parse these URL components, too:

```
let url = new URL("ftp://admin:1337!@ftp.example.com/");
url.href      // => "ftp://admin:1337!@ftp.example.com/"
url.origin    // => "ftp://ftp.example.com"
url.username  // => "admin"
url.password  // => "1337!"
```

The `origin` property here is a simple combination of the URL protocol and host (including the port if one is specified). As such, it is a read-only property. But each of the other properties demonstrated in the previous example is read/write: you can set any of these properties to set the corresponding part of the URL:

```
let url = new URL("https://example.com"); // Start with our server
url.pathname = "api/search";           // Add a path to an API endpoint
url.search = "q=test";                // Add a query parameter
url.toString() // => "https://example.com/api/search?q=test"
```

One of the important features of the URL class is that it correctly adds punctuation and escapes special characters in URLs when that is needed:

```
let url = new URL("https://example.com");
url.pathname = "path with spaces";
url.search = "q=foo#bar";
url.pathname // => "/path%20with%20spaces"
url.search   // => "?q=foo%23bar"
url.href     // => "https://example.com/path%20with%20spaces?q=foo%23bar"
```

The `href` property in these examples is a special one: reading `href` is equivalent to calling `toString()`: it reassembles all parts of the URL into the canonical string form of the URL. And setting `href` to a new string reruns the URL parser on the new string as if you had called the `URL()` constructor again.

In the previous examples, we've been using the `search` property to refer to the entire query portion of a URL, which consists of the characters from a question mark to the end of the URL or to the first hash character. Sometimes, it is sufficient to just treat this as a single URL property. Often, however, HTTP requests encode the values of multiple form fields or multiple API parameters into the query portion of a URL using the `application/x-www-form-urlencoded` format. In this format, the query portion of the URL is a question mark followed by one or more name/value pairs, which are separated from one another by ampersands. The

same name can appear more than once, resulting in a named search parameter with more than one value.

If you want to encode these kinds of name/value pairs into the query portion of a URL, then the `searchParams` property will be more useful than the `search` property. The `search` property is a read/write string that lets you get and set the entire query portion of the URL. The `searchParams` property is a read-only reference to a `URLSearchParams` object, which has an API for getting, setting, adding, deleting, and sorting the parameters encoded into the query portion of the URL:

```
let url = new URL("https://example.com/search");
url.search // => "": no query yet
url.searchParams.append("q", "term"); // Add a search parameter
url.search // => "?q=term"
url.searchParams.set("q", "x"); // Change the value of this parameter
url.search // => "?q=x"
url.searchParams.get("q") // => "x": query the parameter value
url.searchParams.has("q") // => true: there is a q parameter
url.searchParams.has("p") // => false: there is no p parameter
url.searchParams.append("opts", "1"); // Add another search parameter
url.search // => "?q=x&opts=1"
url.searchParams.append("opts", "&"); // Add another value for same name
url.search // => "?q=x&opts=1&opts=%26": note escape
url.searchParams.get("opts") // => "1": the first value
url.searchParams.getAll("opts") // => ["1", "&"]: all values
url.searchParams.sort(); // Put params in alphabetical order
url.search // => "?opts=1&opts=%26&q=x"
url.searchParams.set("opts", "y"); // Change the opts param
url.search // => "?opts=y&q=x"
// searchParams is iterable
[...url.searchParams] // => [["opts", "y"], ["q", "x"]]
url.searchParams.delete("opts"); // Delete the opts param
url.search // => "?q=x"
url.href // => "https://example.com/search?q=x"
```

The value of the `searchParams` property is a `URLSearchParams` object. If you want to encode URL parameters into a query string, you can create a `URLSearchParams` object, append parameters, then convert it to a string and set it on the `search` property of a URL:

```
let url = new URL("http://example.com");
let params = new URLSearchParams();
params.append("q", "term");
params.append("opts", "exact");
params.toString() // => "q=term&opts=exact"
url.search = params;
url.href // => "http://example.com/?q=term&opts=exact"
```

## 11.9.1 Legacy URL Functions

Prior to the definition of the URL API described previously, there have been multiple attempts to support URL escaping and unescaping in the core JavaScript language. The first attempt was the globally defined `escape()` and `unescape()` functions, which are now deprecated but still widely implemented. They should not be used.

When `escape()` and `unescape()` were deprecated, ECMAScript introduced two pairs of alternative global functions:

`encodeURI()` and `decodeURI()`

`encodeURI()` takes a string as its argument and returns a new string in which non-ASCII characters plus certain ASCII characters (such as space) are escaped. `decodeURI()` reverses the process. Characters that need to be escaped are first converted to their UTF-8 encoding, then each byte of that encoding is replaced with a `%xx` escape sequence, where `xx` is two hexadecimal digits. Because `encodeURI()` is intended for encoding entire URLs, it does not escape URL separator characters such as `/`, `?`, and `#`. But this means that `encodeURI()` cannot work correctly for URLs that have those characters within their various components.

#### `encodeURIComponent()` and `decodeURIComponent()`

This pair of functions works just like `encodeURI()` and `decodeURI()` except that they are intended to escape individual components of a URI, so they also escape characters like `/`, `?`, and `#` that are used to separate those components. These are the most useful of the legacy URL functions, but be aware that `encodeURIComponent()` will escape `/` characters in a path name that you probably do not want escaped. And it will convert spaces in a query parameter to `%20`, even though spaces are supposed to be escaped with a `+` in that portion of a URL.

The fundamental problem with all of these legacy functions is that they seek to apply a single encoding scheme to all parts of a URL when the fact is that different portions of a URL use different encodings. If you want a properly formatted and encoded URL, the solution is simply to use the `URL` class for all URL manipulation you do.

## 11.10 Timers

Since the earliest days of JavaScript, web browsers have defined two functions—`setTimeout()` and `setInterval()`—that allow programs to ask the browser to invoke a function after a specified amount of time has elapsed or to invoke the function repeatedly at a specified interval. These functions have never been standardized as part of the core language, but they work in all browsers and in Node and are a de facto part of the JavaScript standard library.

The first argument to `setTimeout()` is a function, and the second argument is a number that specifies how many milliseconds should elapse before the function is invoked. After the specified amount of time (and maybe a little longer if the system is busy), the function will be invoked with no arguments. Here, for example, are three `setTimeout()` calls that print console messages after one second, two seconds, and three seconds:

```
setTimeout(() => { console.log("Ready..."); }, 1000);
setTimeout(() => { console.log("set..."); }, 2000);
setTimeout(() => { console.log("go!"); }, 3000);
```

Note that `setTimeout()` does not wait for the time to elapse before returning. All three lines of code in this example run almost instantly, but then nothing happens until 1,000 milliseconds elapse.

If you omit the second argument to `setTimeout()`, it defaults to 0. That does not mean, however, that the function you specify is invoked immediately. Instead, the function is registered to be called “as soon as possible.” If a browser is particularly busy handling user input or other events, it may take 10 milliseconds or more before the function is invoked.

`setTimeout()` registers a function to be invoked once. Sometimes, that function will itself call `setTimeout()` to schedule another invocation at a future time. If you want to invoke a function repeatedly, however, it is often simpler to use `setInterval()`. `setInterval()` takes the same two arguments as `setTimeout()` but invokes the function repeatedly every time the specified number of milliseconds (approximately) have elapsed.

Both `setTimeout()` and `setInterval()` return a value. If you save this value in a variable, you can then use it later to cancel the execution of the function by passing it to `clearTimeout()` or `clearInterval()`. The returned value is typically a number in web browsers and is an object in Node. The actual type doesn't matter, and you should treat it as an opaque value. The only thing you can do with this value is pass it to `clearTimeout()` to cancel the execution of a function registered with `setTimeout()` (assuming it hasn't been invoked yet) or to stop the repeating execution of a function registered with `setInterval()`.

Here is an example that demonstrates the use of `setTimeout()`, `setInterval()`, and `clearInterval()` to display a simple digital clock with the Console API:

```
// Once a second: clear the console and print the current time
let clock = setInterval(() => {
    console.clear();
    console.log(new Date().toLocaleTimeString());
}, 1000);

// After 10 seconds: stop the repeating code above.
setTimeout(() => { clearInterval(clock); }, 10000);
```

We'll see `setTimeout()` and `setInterval()` again when we cover asynchronous programming in [Chapter 13](#).

## 11.11 Summary

Learning a programming language is not just about mastering the grammar. It is equally important to study the standard library so that you are familiar with all the tools that are shipped with the language. This chapter has documented JavaScript's standard library, which includes:

- Important data structures, such as Set, Map, and typed arrays.
- The Date and URL classes for working with dates and URLs.
- JavaScript's regular expression grammar and its RegExp class for textual pattern matching.
- JavaScript's internationalization library for formatting dates, time, and numbers and for sorting strings.
- The JSON object for serializing and deserializing simple data structures and the console object for logging messages.

<sup>1</sup> Not everything documented here is defined by the JavaScript language specification: some of the classes and functions documented here were first implemented in web browsers and then adopted by Node, making them de facto members of the JavaScript standard library.

<sup>2</sup> This predictable iteration order is another thing about JavaScript sets that Python programmers may find surprising.

[3](#) Typed arrays were first introduced to client-side JavaScript when web browsers added support for WebGL graphics. What is new in ES6 is that they have been elevated to a core language feature.

[4](#) Except within a character class (square brackets), where \b matches the backspace character.

[5](#) Parsing URLs with regular expressions is not a good idea. See [§11.9](#) for a more robust URL parser.

[6](#) C programmers will recognize many of these character sequences from the `printf()` function.