1. Arrow functions are most used when you want to pass an unnamed function as an argument to another function.
2. When functions are assigned to the properties of an object, we call them "methods." All JavaScript objects (including arrays) have methods: let a = []; Create an empty array a.push(1,2,3); The push() method adds elements to an array a.reverse();
3. We can define our own methods, too. The "this" keyword refers to the object.

let arr=[0,1,2]

arr.sum=function (){

let sum=0

for(let i=0;i<this.length;i++) sum+=this[i]

return sum

}

console.log(arr.sum()) //=> 3

1. A literal is a data value that appears directly in a program. The following are all literals:

12 // The number twelve

1.2 // The number one point two

"hello world" // A string of text

true // A Boolean value

null // Absence of an object

1. A JavaScript identifier must begin with a letter, an underscore (\_), or a dollar sign ($). Subsequent characters can be letters, digits, underscores, or dollar signs. (Digits are not allowed as the first character so that JavaScript can easily distinguish identifiers from numbers.)
2. let can be used as a variable name if declared with var outside of a class, for example, but not if declared inside a class or with const.
3. programmers can use mathematical symbols and words from non-English languages as constants and variables: const π = 3.14; const sí = true;
4. Unicode escapes may appear in JavaScript string literals, regular expression literals, and identifiers (but not in language keywords). The Unicode escape for the character “é,” for example, is \u00E9; here are three different ways to write a variable name that includes this character:

let café = 1; // Define a variable using a Unicode character.

console.log(caf\u00e9) // => 1;

console.log(caf\u{E9}) // => 1;

1. JavaScript does not treat every line break as a semicolon: it usually treats line breaks as semicolons only if it cannot parse the code without adding an implicit semicolon. More formally (and with three exceptions described a bit later), JavaScript treats a line break as a semicolon if the next nonspace character cannot be interpreted as a continuation of the current statement. Consider the following code:

let a

a

=

3

console.log(a)

JavaScript interprets this code like this: let a; a = 3; console.log(a); JavaScript does treat the first line break as a semicolon because it cannot parse the code let a a without a semicolon. The second a could stand alone as the statement a;, but JavaScript does not treat the second line break as a semicolon because it can continue parsing the longer statement a = 3;.

These statement termination rules lead to some surprising cases. This code looks like two separate statements separated with a newline:

let y = x + f

(a+b).toString()

But the parentheses on the second line of code can be interpreted as a function invocation of f from the first line, and JavaScript interprets the code like this: let y = x + f(a+b).toString();

1. There are three exceptions to the general rule that JavaScript interprets line breaks as semicolons when it cannot parse the second line as a continuation of the statement on the first line. The first exception involves the return, throw, yield, break, and continue statements.

If a line break appears after any of these words (before any other tokens), JavaScript will always interpret that line break as a semicolon. For example, if you write:

return

true;

JavaScript assumes you meant: return; true;

However, you probably meant: return true;

The second exception involves the ++ and −− operators. These operators can be prefix operators that appear before an expression or postfix operators that appear after an expression. If you want to use either of these operators as postfix operators, they must appear on the same line as the expression they apply to.

The third exception involves functions defined using concise “arrow” syntax: the => arrow itself must appear on the same line as the parameter list.

1. JavaScript types can be divided into two categories: primitive types and object types. JavaScript’s primitive types include numbers, strings of text (known as strings), and Boolean truth values (known as Booleans).

The special JavaScript values null and undefined are primitive values, but they are not numbers, strings, or Booleans. Each value is typically considered to be the sole member of its own special type.

Any JavaScript value that is not a number, a string, a Boolean, a symbol, null, or undefined is an object.

Like any JavaScript value that is not a primitive value, functions and classes are a specialized kind of object.

Technically, it is only JavaScript objects that have methods. But numbers, strings, Boolean, and symbol values behave as if they have methods. In JavaScript, null and undefined are the only values that methods cannot be invoked on.

1. When a number appears directly in a JavaScript program, it’s called a numeric literal.

In a JavaScript program, a base-10 integer is written as a sequence of digits. For example:

0

3

10000000

In addition to base-10 integer literals, JavaScript recognizes hexadecimal (base-16) values. A hexadecimal literal begins with 0x or 0X, followed by a string of hexadecimal digits. A hexadecimal digit is one of the digits 0 through 9 or the letters a (or A) through f (or F), which represent values 10 through 15. Here are examples of hexadecimal integer literals:

0xff // => 255: (15\*16 + 15)

0xBADCAFE // => 195939070

In ES6 and later, you can also express integers in binary (base 2) or octal (base 8) using the prefixes 0b and 0o (or 0B and 0O) instead of 0x:

0b10101 // => 21: (1\*16 + 0\*8 + 1\*4 + 0\*2 + 1\*1)

0o377 // => 255: (3\*64 + 7\*8 + 7\*1)

1. Floating-point literals may also be represented using exponential notation: a real number followed by the letter e (or E), followed by an optional plus or minus sign, followed by an integer exponent. This notation represents the real number multiplied by 10 to the power of the exponent.

More succinctly, the syntax is: [digits][.digits][(E|e)[(+|-)]digits]

For example:

3.14

2345.6789

.333333333333333333

6.02e23 // 6.02 × 10²³

1.4738223E-32 // 1.4738223 × 10⁻³²

1. You can use underscores within numeric literals to break long literals up into chunks that are easier to read:

let billion = 1\_00\_000\_000; // Underscore as a thousand separator.

let bytes = 0x89\_ab\_cd\_ef; // As a byte’s separator.

let bits = 0b0001\_1101\_0111; // As a nibble separator.

let fraction = 0.123\_456\_789; // Works in the fractional part, too.

1. Math.pow(2,53) // => 9007199254740992: 2 to the power 53

Math.round(.6) // => 1.0: round to the nearest integer

Math.ceil(.6) // => 1.0: round up to an integer

Math.floor(.6) // => 0.0: round down to an integer

Math.abs(-5) // => 5: absolute value

Math.max(x,y,z) // Return the largest argument

Math.min(x,y,z) // Return the smallest argument

Math.random() // x where 0 <= x < 1.0

Math.PI // π: circumference of a circle / diameter

Math.E // e: The base of the natural logarithm

Math.sqrt(3) // => 3\*\*0.5: the square root of 3

Math.pow(3, 1/3) // => 3\*\*(1/3): the cube root of 3

Math.sin(0) // Trigonometry:Math.cos, Math.atan, etc.

Math.log(10) // Natural logarithm of 10

Math.log(100)/Math.LN10 // Base 10 logarithm of 100

Math.log(512)/Math.LN2 // Base 2 logarithm of 512

Math.exp(3) // Math.E cubed

Math.cbrt(27) // => 3: cube root

Math.hypot(3, 4) //=> 5: square root of sum of squares of all arguments

Math.trunc(3.9)//=>3:convert to an integer by truncating fractional part

1. Arithmetic in JavaScript does not raise errors in cases of overflow, underflow, or division by zero. When the result of a numeric operation is larger than the largest representable number (overflow), the result is a special infinity value, Infinity. Similarly, when the absolute value of a negative value becomes larger than the absolute value of the largest representable negative number, the result is negative infinity, - Infinity. The infinite values behave as you would expect: adding, subtracting, multiplying, or dividing them by anything results in an infinite value (possibly with the sign reversed). Underflow occurs when the result of a numeric operation is closer to zero than the smallest representable number. In this case, JavaScript returns 0. If underflow occurs from a negative number, JavaScript returns a special value known as “negative zero.” This value is almost completely indistinguishable from regular zero and JavaScript programmers rarely need to detect it. Division by zero is not an error in JavaScript: it simply returns infinity or negative infinity. There is one exception, however: zero divided by zero does not have a well-defined value, and the result of this operation is the special not-a-number value, NaN. NaN also arises if you attempt to divide infinity by infinity, take the square root of a negative number, or use arithmetic operators with non-numeric operands that cannot be converted to numbers. JavaScript predefines global constants Infinity and NaN to hold the positive infinity and not-a-number value, and these values are also available as properties of the Number object:

Infinity // A positive number too big to represent

Number.POSITIVE\_INFINITY // Same value

1/0 // => Infinity

Number.MAX\_VALUE \* 2 // => Infinity; overflow

-Infinity // A negative number too big to represent

Number.NEGATIVE\_INFINITY // The same value

-1/0 // => -Infinity

-Number.MAX\_VALUE \* 2 // => -Infinity

NaN // The not-a-number value

Number.NaN // The same value, written another way

0/0 // => NaN

Infinity/Infinity // => NaN

Number.MIN\_VALUE/2 // => 0: underflow

-Number.MIN\_VALUE/2 // => -0: negative zero

-1/Infinity // -> -0: also negative 0

The following Number properties are defined in ES6

Number.parseInt() // Same as the global parseInt() function

Number.parseFloat() // Same as the global parseFloat() function

Number.isNaN(x) // Is x the NaN value?

Number.isFinite(x) // Is x a number and finite?

Number.isInteger(x) // Is x an integer?

Number.isSafeInteger(x) // Is x an integer -(2\*\*53) < x < 2\*\*53?

Number.MIN\_SAFE\_INTEGER // => -(2\*\*53 - 1)

Number.MAX\_SAFE\_INTEGER // => 2\*\*53 - 1

Number.EPSILON // => 2\*\*-52: smallest difference between numbers.

1. The not-a-number value has one unusual feature in JavaScript: it does not compare equal to any other value, including itself. This means that you can’t write x === NaN to determine whether the value of a variable x is NaN. Instead, you must write x != x or Number.isNaN(x). Those expressions will be true if, and only if, x has the same value as the global constant NaN. The global function isNaN() is similar to Number.isNaN(). It returns true if its argument is NaN, or if that argument is a nonnumeric value that cannot be converted to a number. The related function Number.isFinite() returns true if its argument is a number other than NaN, Infinity, or -Infinity. The global isFinite() function returns true if its argument is, or can be converted to, a finite number. The negative zero value is also somewhat unusual. It compares equal (even using JavaScript’s strict equality test) to positive zero, which means that the two values are almost indistinguishable, except when used as a divisor:

let zero = 0; // Regular zero

let negz = -0; // Negative zero

zero === negz // => true: zero and negative zero are equal

1/zero === 1/negz // => false: Infinity and -Infinity are not equal

1. floating-point representation used by JavaScript (and just about every other modern programming language) is a binary representation, which can exactly represent fractions like 1/2, 1/8, and 1/1024. Unfortunately, the fractions we use most commonly (especially when performing financial calculations) are decimal fractions: 1/10, 1/100, and so on. Binary floating-point representations cannot exactly represent numbers as simple as 0.1. JavaScript numbers have plenty of precision and can approximate 0.1 very closely. But the fact that this number cannot be represented exactly can lead to problems. Consider this code:

let x = .3 - .2; // thirty cents minus 20 cents

let y = .2 - .1; // twenty cents minus 10 cents

x === y // => false: the two values are not the same!

x === .1 // => false: .3-.2 is not equal to .1

y === .1 // => true: .2-.1 is equal to .1

1. BigInt literals are written as a string of digits followed by a lowercase letter n. By default, the are in base 10, but you can use the 0b, 0o, and 0x prefixes for binary, octal, and hexadecimal BigInts:

1234n // A not-so-big BigInt literal

0b111111n // A binary BigInt

0o7777n // An octal BigInt

0x8000000000000000n // => 2n\*\*63n: A 64-bit integer

You can use BigInt() as a function for converting regular JavaScript numbers or strings to BigInt values:

BigInt(Number.MAX\_SAFE\_INTEGER) // => 9007199254740991n

let string = "1" + "0".repeat(100); // 1 followed by 100 zeros.

BigInt(string) // => 10n\*\*100n: one googol

Arithmetic with BigInt values works like arithmetic with regular JavaScript numbers, except that division drops any remainder and rounds down (toward zero):

1000n + 2000n // => 3000n

3000n - 2000n // => 1000n

2000n \* 3000n // => 6000000n

3000n / 997n // => 3n: the quotient is 3

3000n % 997n // => 9n: and the remainder is 9

(2n \*\* 131071n) - 1n // A Mersenne prime with 39457 decimal digits

Although the standard +, -, \*, /, %, and \*\* operators work with BigInt, it is important to understand that you may not mix operands of type BigInt with regular number operands. This may seem confusing at first, but there is a good reason for it. If one numeric type was more general than the other, it would be easy to define arithmetic on mixed operands to simply return a value of the more general type. But neither type is more general than the other: BigInt can represent extraordinarily large values, making it more general than regular numbers. But BigInt can only represent integers, making the regular JavaScript number type more general. There is no way around this problem, so JavaScript sidesteps it by simply not allowing mixed operands to the arithmetic operators. Comparison operators, by contrast, do work with mixed numeric types

1 < 2n // => true

2 > 1n // => true

0 == 0n // => true

0 === 0n // => false: the === checks for type equality as well.

None of the functions of the Math object accept BigInt operands, however.

1. The JavaScript type for representing text is the string. A string is an immutable ordered sequence of 16-bit values, each of which typically represents a Unicode character. The length of a string is the number of 16-bit values it contains. JavaScript’s strings (and its arrays) use zero-based indexing: the first 16-bit value is at position 0, the second at position 1, and so on. The empty string is the string of length 0. JavaScript does not have a special type that represents a single element of a string. To represent a single 16-bit value, simply use a string that has a length of 1.

CHARACTERS, CODEPOINTS, AND JAVASCRIPT STRINGS:

JavaScript uses the UTF-16 encoding of the Unicode character set, and JavaScript strings are sequences of unsigned 16-bit values. The most commonly used Unicode characters (those from the “basic multilingual plane”) have codepoints that fit in 16 bits and can be represented by one element of a string. Unicode characters whose codepoints do not fit in 16 bits are encoded using the rules of UTF-16 as a sequence (known as a “surrogate pair”) of two 16-bit values. This means that a JavaScript string of length 2 (two 16-bit values) might represent only a single Unicode character:

let euro = "€";

let love = "❤";

euro.length // => 1: this character has one 16-bit element

love.length // => 2: UTF-16 encoding of ❤ is "\ud83d\udc99"

Most string-manipulation methods defined by JavaScript operate on 16-bit values, not characters. They do not treat surrogate pairs specially, they perform no normalization of the string, and don’t even ensure that a string is well-formed UTF-16. In ES6, however, strings are iterable, and if you use the for/of loop or ... operator with a string, it will iterate the actual characters of the string, not the 16-bit values.

1. As of ES5, however, you can break a string literal across multiple lines by ending each line but the last with a backslash (\). Neither the backslash nor the line terminator that follow it are part of the string literal. If you need to include a newline character in a single-quoted or double-quoted string literal, use the character sequence \n (documented in the next section). The ES6 backtick syntax allows strings to be broken across multiple lines, and in this case, the line terminators are part of the string literal:

// A string representing 2 lines written on one line:

'two\nlines'

// A one-line string written on 3 lines:

"one\

long\

line"

// A two-line string written on two lines:

`the newline character at the end of this line is included literally in this string`.

1. let s = "Hello, world"; // Start with some text.

// Obtaining portions of a string

s.substring(1,4) // => "ell": the 2nd, 3rd, and 4th characters.

s.slice(1,4) // => "ell": same thing

s.slice(-3) // => "rld": last 3 characters

s.split(", ") // => ["Hello", "world"]: split at delimiter string

// Searching a string

s.indexOf("l") // => 2: position of first letter l

s.indexOf("l", 3) // => 3: position of first "l" at or after 3

s.indexOf("zz") // => -1: s does not include the substring "zz"

s.lastIndexOf("l") // => 10: position of last letter l

// Boolean searching functions in ES6 and later

s.startsWith("Hell") // => true: the string starts with these

s.endsWith("!") // => false: s does not end with that

s.includes("or") // => true: s includes substring "or"

// Creating modified versions of a string

s.replace("llo", "ya") // => "Heya, world"

s.toLowerCase() // => "hello, world"

s.toUpperCase() // => "HELLO, WORLD"

// Inspecting individual (16-bit) characters of a string

s.charAt(0) // => "H": the first character

s.charAt(s.length-1) // => "d": the last character

s.charCodeAt(0) // => 72: 16-bit number at the specified position

s.codePointAt(0) // => 72: ES6, works for codepoints > 16 bits

// String padding functions in ES2017

"x".padStart(3) // => " x": add spaces on the left to a length of 3

"x".padEnd(3) // => "x ": add spaces on the right to a length of 3

"x".padStart(3, "\*") // => "\*\*x": add stars on the left to a length of 3

"x".padEnd(3, "-") // => "x--":add dashes on the right to a length of 3

// Space trimming functions. trim() is ES5; others ES2019

" test ".trim() // => "test": remove spaces at start and end " test ".

trimStart() // => "test ": remove spaces on left. Also trimLeft

" test ".trimEnd() // => " test": remove spaces at right. Also trimRight

// Miscellaneous string methods

s.concat("!") // => "Hello, world!": just use + operator instead

"<>".repeat(5) // => "<><><><><>": concatenate n copies.

Remember that strings are immutable in JavaScript. Methods like replace() and toUpperCase() return new strings: they do not modify the string on which they are invoked. Strings can also be treated like read-only arrays, and you can access individual characters (16-bit values) from a string using square brackets instead of the charAt() method.

1. The following values convert to, and therefore work like, false: undefined, null, 0, -0, NaN, "". All other values, including all objects (and arrays) convert to, and work like, true. false, and the six values that convert to it, are sometimes called falsy values, and all other values are called truthy. Any time JavaScript expects a boolean value, a falsy value works like false and a truthy value works like true.
2. null is a language keyword that evaluates to a special value that is usually used to indicate the absence of a value. Using the typeof operator on null returns the string “object”, indicating that null can be thought of as a special object value that indicates “no object”. In practice, however, null is typically regarded as the sole member of its own type, and it can be used to indicate “no value” for numbers and strings as well as objects.

JavaScript also has a second value that indicates absence of value. The undefined value represents a deeper kind of absence. It is the value of variables that have not been initialized and the value you get when you query the value of an object property or array element that does not exist. The undefined value is also the return value of functions that do not explicitly return a value and the value of function parameters for which no argument is passed. undefined is a predefined global constant (not a language keyword like null, though this is not an important distinction in practice) that is initialized to the undefined value. If you apply the typeof operator to the undefined value, it returns “undefined”, indicating that this value is the sole member of a special type. Despite these differences, null and undefined both indicate an absence of value and can often be used interchangeably. The equality operator == considers them to be equal. (Use the strict equality operator === to distinguish them.) Both are falsy values: they behave like false when a boolean value is required. Neither null nor undefined have any properties or methods. In fact, using . or [] to access a property or method of these values causes a TypeError.

1. Symbols were introduced in ES6 to serve as non-string property names. To understand Symbols, you need to know that JavaScript’s fundamental Object type is an unordered collection of properties, where each property has a name and a value. Property names are typically (and until ES6, were exclusively) strings. But in ES6 and later, Symbols can also serve this purpose:

let s = Symbol("propname");

let y = Symbol("propname");

console.log(s, y); // => Symbol(propname) Symbol(propname)

let obj = {};

obj[s] = "symbol s";

obj[y] = "symbol y";

console.log(obj);

// => { [Symbol(propname)]: 'symbol s', [Symbol(propname)]: 'symbol y' }

This function never returns the same value twice, even when called with the same argument. This means that if you call Symbol() to obtain a Symbol value, you can safely use that value as a property name to add a new property to an object and do not need to worry that you might be overwriting an existing property with the same name.

JavaScript defines a global Symbol registry. The Symbol.for() function takes a string argument and returns a Symbol value that is associated with the string you pass. If no Symbol is already associated with that string, then a new one is created and returned; otherwise, the already existing Symbol is returned. That is, the Symbol.for() function is completely different than the Symbol() function: Symbol() never returns the same value twice, but Symbol.for() always returns the same value when called with the same string. The string passed to Symbol.for() appears in the output of toString() for the returned Symbol, and it can also be retrieved by calling Symbol.keyFor() on the returned Symbol.

let s = Symbol.for("shared");

let t = Symbol.for("shared");

s === t // => true

s.toString() // => "Symbol(shared)"

Symbol.keyFor(t) // => "shared"

1. There is a fundamental difference in JavaScript between primitive values (undefined, null, booleans, numbers, and strings) and objects (including arrays and functions).

Primitives are also compared by value: two values are the same only if they have the same value. This sounds circular for numbers, booleans, null, and undefined: there is no other way that they could be compared. Again, however, it is not so obvious for strings. If two distinct string values are compared, JavaScript treats them as equal if, and only if, they have the same length and if the character at each index is the same.

Objects are different than primitives. First, they are mutable—their values can change

Objects are not compared by value: two distinct objects are not equal even if they have the same properties and values. And two distinct arrays are not equal even if they have the same elements in the same order:

let o = {x: 1}, p = {x: 1}; // Two objects with the same properties

o === p // => false: distinct objects are never equal

let a = [], b = []; // Two distinct, empty arrays

a === b // => false: distinct arrays are never equal

Objects are sometimes called reference types to distinguish them from JavaScript’s primitive types. Using this terminology, object values are references, and we say that objects are compared by reference: two object values are the same if and only if they refer to the same underlying object.

let a = []; // The variable a refers to an empty array.

let b = a; // Now b refers to the same array.

b[0] = 1; // Mutate the array referred to by variable b.

a[0] // => 1: the change is also visible through variable a.

a === b // => true: a and b refer to the same object, so they are equal.

1. JavaScript is very flexible about the types of values it requires.

Some examples:

10 + " objects" // => "10 objects": Number 10 converts to a string

"7" \* "4" // => 28: both strings convert to numbers

let n = 1 - "x"; // n == NaN; string "x" can't convert to a number

n + " objects" // => "NaN objects": NaN converts to string "NaN"

1. JavaScript type conversions

|  |  |  |  |
| --- | --- | --- | --- |
| Value | to String | to Number | to Boolean |
| undefined | "undefined" | NaN | false |
| null | "null" | 0 | false |
| true | "true" | 1 |  |
| false | "false" | 0 |  |
| "" |  | 0 | false |
| "1.2" |  | 1.2 | true |
| "one" |  | NaN | true |
| 0 | "0" |  | false |
| -0 | "0" |  | false |
| 1 | "1" |  | true |
| Infinity | "Infinity" |  | true |
| -Infinity | "-Infinity" |  | true |
| NaN | "NaN" |  | false |
| [] | "" | 0 | true |
| [9] | "9" | 9 | true |
| ["a"] (any other array) | Use join() | NaN | true |
| function(){}  (any other function) |  | NaN | true |

1. The simplest way to perform an explicit type conversion is to use the Boolean(), Number(), and String() functions:

Number("3") // => 3

String(false) // => "false": Or use false.toString()

Boolean([]) // => true

Any value other than null or undefined has a toString() method, and the result of this method is usually the same as that returned by the String() function.

1. If one operand of the + operator is a string, it converts the other one to a string. The unary + operator converts its operand to a number. And the unary ! operator converts its operand to a boolean and negates it. These facts lead to the following type conversion idioms that you may see in some code:

x + "" // => String(x)

+x // => Number(x)

x-0 // => Number(x)

!!x // => Boolean(x): Note double !

1. The toString() method defined by the Number class accepts an optional argument that specifies a radix, or base, for the conversion. If you do not specify the argument, the conversion is done in base 10. However, you can also convert numbers in other bases (between 2 and 36). For example:

let n = 17;

let binary = "0b" + n.toString(2); // binary == "0b10001"

let octal = "0o" + n.toString(8); // octal == "0o21"

let hex = "0x" + n.toString(16); // hex == "0x11"

1. let n = 123456.789;

toFixed() converts a number to a string with a specified number of digits after the decimal point. It never uses exponential notation

n.toFixed(0) // => "123457"

n.toFixed(2) // => "123456.79"

n.toFixed(5) // => "123456.78900"

toExponential() converts a number to a string using exponential notation, with one digit before the decimal point and a specified number of digits after the decimal point (which means that the number of significant digits is one larger than the value you specify).

n.toExponential(1) // => "1.2e+5"

n.toExponential(3) // => "1.235e+5"

toPrecision() converts a number to a string with the number of significant digits you specify. It uses exponential notation if the number of significant digits is not large enough to display the entire integer portion of the number. Note that all three methods round the trailing digits or pad with zeros as appropriate.

n.toPrecision(4) // => "1.235e+5"

n.toPrecision(7) // => "123456.8"

n.toPrecision(10) // => "123456.7890"

1. If you pass a string to the Number() conversion function, it attempts to parse that string as an integer or floating-point literal. That function only works for base-10 integers and does not allow trailing characters that are not part of the literal. The parseInt() and parseFloat() functions (these are global functions, not methods of any class) are more flexible. parseInt() parses only integers, while parseFloat() parses both integers and floating-point numbers. If a string begins with “0x” or “0X”, parseInt() interprets it as a hexadecimal number. Both parseInt() and parseFloat() skip leading whitespace, parse as many numeric characters as they can, and ignore anything that follows. If the first nonspace character is not part of a valid numeric literal, they return NaN:

parseInt("3 blind mice") // => 3

parseFloat(" 3.14 meters") // => 3.14

parseInt("-12.34") // => -12

parseInt("0xFF") // => 255

parseInt("0xff") // => 255

parseInt("-0XFF") // => -255

parseFloat(".1") // => 0.1

parseInt("0.1") // => 0

parseInt(".1") // => NaN: integers can't start with "." parseFloat("$72.47") // => NaN: numbers can't start with "$"

parseInt() accepts an optional second argument specifying the radix (base) of the number to be parsed. Legal values are between 2 and 36. For example:

parseInt("11", 2) // => 3: (1\*2 + 1)

parseInt("ff", 16) // => 255: (15\*16 + 15)

parseInt("zz", 36) // => 1295: (35\*36 + 35)

parseInt("077", 8) // => 63: (7\*8 + 7)

parseInt("077", 10) // => 77: (7\*10 + 7)

1. One reason for the complexity of JavaScript’s object-to-primitive conversions is that some types of objects have more than one primitive representation. Date objects, for example, can be represented as strings or as numeric timestamps. The JavaScript specification defines three fundamental algorithms for converting objects to primitive values:

*prefer-string* -- This algorithm returns a primitive value, preferring a string value, if a conversion to string is possible.

*prefer-number* -- This algorithm returns a primitive value, preferring a number, if such a conversion is possible.

*no-preference* -- This algorithm expresses no preference about what type of primitive value is desired, and classes can define their own conversions. Of the built-in JavaScript types, all except Date implement this algorithm as prefer-number. The Date class implements this algorithm as prefer-string.

*OBJECT-TO-BOOLEAN CONVERSIONS --* All objects convert to true. Notice that this conversion does not require the use of the object-to- primitive algorithms described, and that it literally applies to all objects, including empty arrays and even the wrapper object new Boolean(false).

*OBJECT-TO-STRING CONVERSIONS --* When an object needs to be converted to a string, JavaScript first converts it to a primitive using the prefer-string algorithm, then converts the resulting primitive value to a string.

*OBJECT-TO-NUMBER CONVERSIONS --* When an object needs to be converted to a number, JavaScript first converts it to a primitive value using the prefer-number algorithm, then converts the resulting primitive value to a number. Built-in JavaScript functions and methods that expect numeric arguments convert object arguments to numbers in this way, and most (see the exceptions that follow) JavaScript operators that expect numeric operands convert objects to numbers in this way as well.

*SPECIAL CASE OPERATOR CONVERSIONS --* Here, we explain the special case operators that do not use the basic object-to-string and object-to-number conversions described earlier.

The + operator in JavaScript performs numeric addition and string concatenation. If either of its operands is an object, JavaScript converts them to primitive values using the no-preference algorithm. Once it has two primitive values, it checks their types. If either argument is a string, it converts the other to a string and concatenates the strings. Otherwise, it converts both arguments to numbers and adds them.

The == and != operators perform equality and inequality testing in a loose way that allows type conversions. If one operand is an object and the other is a primitive value, these operators convert the object to primitive using the no-preference algorithm and then compare the two primitive values.

Finally, the relational operators <, <=, >, and >= compare the order of their operands and can be used to compare both numbers and strings. If either operand is an object, it is converted to a primitive value using the prefer-number algorithm. Note, however, that unlike the object-tonumber conversion, the primitive values returned by the prefer-number conversion are not then converted to numbers.

Note that the numeric representation of Date objects is meaningfully comparable with < and >, but the string representation is not. For Date objects, the no-preference algorithm converts to a string, so the fact that JavaScript uses the prefer-number algorithm for these operators means that we can use them to compare the order of two Date objects.

*THE TOSTRING() AND VALUEOF() METHODS* -- All objects inherit two conversion methods that are used by object-toprimitive conversions, and before we can explain the prefer-string, prefer-number, and no-preference conversion algorithms, we have to explain these two methods.

The first method is toString(), and its job is to return a string representation of the object. The default toString() method does not return a very interesting value

({x: 1, y: 2}).toString() // => "[object Object]"

Many classes define more specific versions of the toString() method. The toString() method of the Array class, for example, converts each array element to a string and joins the resulting strings together with commas in between. The toString() method of the Function class converts user-defined functions to strings of JavaScript source code. The Date class defines a toString() method that returns a human-readable (and JavaScript-parsable) date and time string. The RegExp class defines a toString() method that converts RegExp objects to a string that looks like a RegExp literal:

[1,2,3].toString() // => "1,2,3"

(function(x) { f(x); }).toString() // => "function(x) { f(x); }" /\d+/g.toString() // => "/\\d+/g"

let d = new Date(2020,0,1);

d.toString() // => "Wed Jan 01 2020 00:00:00 GMT-0800 (Pacific Standard Time)"

The other object conversion function is called valueOf(). The job of this method is less well defined: it is supposed to convert an object to a primitive value that represents the object, if any such primitive value exists. Objects are compound values, and most objects cannot really be represented by a single primitive value, so the default valueOf() method simply returns the object itself rather than returning a primitive. Wrapper classes such as String, Number, and Boolean define valueOf() methods that simply return the wrapped primitive value. Arrays, functions, and regular expressions simply inherit the default method. Calling valueOf() for instances of these types simply returns the object itself. The Date class defines a valueOf() method that returns the date in its internal representation: the number of milliseconds since January 1, 1970:

let d = new Date(2010, 0, 1); // January 1, 2010, (Pacific time) d.valueOf() // => 1262332800000

*OBJECT-TO-PRIMITIVE CONVERSION* ALGORITHMS -- With the toString() and valueOf() methods explained, we can now explain approximately how the three object-to-primitive algorithms work.

The prefer-string algorithm first tries the toString() method. If the method is defined and returns a primitive value, then JavaScript uses that primitive value (even if it is not a string!). If toString() does not exist or if it returns an object, then JavaScript tries the valueOf() method. If that method exists and returns a primitive value, then JavaScript uses that value. Otherwise, the conversion fails with a TypeError.

The prefer-number algorithm works like the prefer-string algorithm, except that it tries valueOf() first and toString() second.

The no-preference algorithm depends on the class of the object being converted. If the object is a Date object, then JavaScript uses the prefer-string algorithm. For any other object, JavaScript uses the prefer-number algorithm. The rules described here are true for all built-in JavaScript types and are the default rules for any classes you define yourself.

Before we leave this topic, it is worth noting that the details of the prefer-number conversion explain why empty arrays convert to the number 0 and single-element arrays can also convert to numbers:

Number([]) // => 0: this is unexpected!

Number([99]) // => 99: really?

The object-to-number conversion first converts the object to a primitive using the prefer-number algorithm, then converts the resulting primitive value to a number. The prefer-number algorithm tries valueOf() first and then falls back on toString(). But the Array class inherits the default valueOf() method, which does not return a primitive value. So when we try to convert an array to a number, we end up invoking the toString() method of the array. Empty arrays convert to the empty string. And the empty string converts to the number 0. An array with a single element converts to the same string that that one element does. If an array contains a single number, that number is converted to a string, and then back to a number.

1. Variables declared with var do not have block scope. Instead, they are scoped to the body of the containing function no matter how deeply nested they are inside that function.

If you use var outside of a function body, it declares a global variable. But global variables declared with var differ from globals declared with let in an important way. Globals declared with var are implemented as properties of the global object. The global object can be referenced as globalThis. So if you write var x = 2; outside of a function, it is like you wrote globalThis.x = 2;. Note however, that the analogy is not perfect: the properties created with global var declarations cannot be deleted with the delete operator. Global variables and constants declared with let and const are not properties of the global object.

Unlike variables declared with let, it is legal to declare the same variable multiple times with var.

One of the most unusual features of var declarations is known as hoisting. When a variable is declared with var, the declaration is lifted up (or “hoisted”) to the top of the enclosing function. The initialization of the variable remains where you wrote it, but the definition of the variable moves to the top of the function. So variables declared with var can be used, without error, anywhere in the enclosing function. If the initialization code has not run yet, then the value of the variable may be undefined, but you won’t get an error if you use the variable before it is initialized.

In strict mode, if you attempt to use an undeclared variable, you’ll get a reference error when you run your code. Outside of strict mode, however, if you assign a value to a name that has not been declared with let, const, or var, you’ll end up creating a new global variable. It will be a global no matter now deeply nested within functions and blocks your code is, which is almost certainly not what you want, is bug-prone, and is one of the best reasons for using strict mode! Global variables created in this accidental way are like global variables declared with var: they define properties of the global object. But unlike the properties defined by proper var declarations, these properties can be deleted with the delete operator.

1. *Destructuring Assignment* -- Here are simple destructuring assignments using arrays of values:

let [x,y] = [1,2]; // Same as let x=1, y=2

[x,y] = [x+1,y+1]; // Same as x = x + 1, y = y + 1

[x,y] = [y,x]; // Swap the value of the two variables

[x,y] // => [3,2]: the incremented and swapped values

// Convert [x,y] coordinates to [r,theta] polar coordinates

function toPolar(x, y) {

return [Math.sqrt(x\*x+y\*y), Math.atan2(y,x)]; }

// Convert polar to Cartesian coordinates

function toCartesian(r, theta) {

return [r\*Math.cos(theta), r\*Math.sin(theta)]; }

let [r,theta] = toPolar(1.0, 1.0);//r== Math.sqrt(2); theta == Math.PI/4

let [x,y] = toCartesian(r,theta); // [x, y] == [1.0, 1,0]

let o = { x: 1, y: 2 }; // The object we'll loop over

for(const [name, value] of Object.entries(o)) {

console.log(name, value); // Prints "x 1" and "y 2" }

let [x,y] = [1]; // x == 1; y == undefined

[x,y] = [1,2,3]; // x == 1; y == 2

[,x,,y] = [1,2,3,4]; // x == 2; y == 4

let [x, ...y] = [1,2,3,4]; // y == [2,3,4]

let [a, [b, c]] = [1, [2,2.5], 3]; // a == 1; b == 2; c == 2.5

A powerful feature of array destructuring is that it does not actually require an array! You can use any iterable object on the righthand side of the assignment; any object that can be used with a for/of loop can also be destructured:

let [first, ...rest] = "Hello"; // first == "H"; rest == ["e","l","l","o"]

let transparent = {r: 0.0, g: 0.0, b: 0.0, a: 1.0}; // A RGBA color

let {r, g, b} = transparent; // r == 0.0; g == 0.0; b == 0.0

The next example copies global functions of the Math object into variables, which might simplify code that does a lot of trigonometry:

// Same as const sin=Math.sin, cos=Math.cos, tan=Math.tan

const {sin, cos, tan} = Math;

// Same as const cosine = Math.cos, tangent = Math.tan;

const { cos: cosine, tan: tangent } = Math;

let points = [{x: 1, y: 2}, {x: 3, y: 4}]; // An array of two point objects let [{x:x1,y:y1},{x: x2, y: y2}]=points;// destructured into 4 variables.

(x1 === 1 && y1 === 2 && x2 === 3 && y2 === 4) // => true

let points = { p1: [1,2], p2: [3,4] }; // An object with 2 array props

let { p1: [x1, y1], p2: [x2, y2] } = points; // destructured into 4 vars (x1 === 1 && y1 === 2 && x2 === 3 && y2 === 4) // => true

1. *Object and Array Initializers* -- Undefined elements can be included in an array literal by simply omitting a value between commas. For example, the following array contains five elements, including three undefined elements: let sparseArray = [1,,,,5];
2. *Conditional Property Access* -- property access with ?. is “short-circuiting”: if the subexpression to the left of ?. evaluates to null or undefined, then the entire expression immediately evaluates to undefined without any further property access attempts.
3. *Conditional Invocation* -- In ES2020, you can also invoke a function using ?.() instead of (). Normally when you invoke a function, if the expression to the left of the parentheses is null or undefined or any other non-function, a TypeError is thrown. With the new ?.() invocation syntax, if the expression to the left of the ?. evaluates to null or undefined, then the entire invocation expression evaluates to undefined and no exception is thrown. Note, however, that ?.() only checks whether the lefthand side is null or undefined. It does not verify that the value is actually a function.
4. *Object Creation Expressions* -- An object creation expression creates a new object and invokes a function (called a constructor) to initialize the properties of that object. Object creation expressions are like invocation expressions except that they are prefixed with the keyword new:

new Object()

new Point(2,3)

If no arguments are passed to the constructor function in an object creation expression, the empty pair of parentheses can be omitted:

new Object

new Date

1. *Number of Operands* -- Operators can be categorized based on the number of operands they expect (their arity). Most JavaScript operators, like the \* multiplication operator, are binary operators that combine two expressions into a single, more complex expression. That is, they expect two operands. JavaScript also supports a number of unary operators, which convert a single expression into a single, more complex expression. The − operator in the expression −x is a unary operator that performs the operation of negation on the operand x. Finally, JavaScript supports one ternary operator, the conditional operator ?:, which combines three expressions into a single expression.
2. When new operators are added to JavaScript, they do not always fit naturally into this precedence scheme. The ?? operator is shown in the table as lower-precedence than || and &&, but, in fact, its precedence relative to those operators is not defined, and ES2020 requires you to explicitly use parentheses if you mix ?? with either || or &&. Similarly, the new \*\* exponentiation operator does not have a well-defined precedence relative to the unary negation operator, and you must use parentheses when combining negation with exponentiation.
3. The \*\* operator has higher precedence than \*, /, and % (which in turn have higher precedence than + and -). Unlike the other operators, \*\* works right-to-left, so 2\*\*2\*\*3 is the same as 2\*\*8, not 4\*\*3. There is a natural ambiguity to expressions like -3\*\*2. Depending on the relative precedence of unary minus and exponentiation, that expression could mean (-3)\*\*2 or -(3\*\*2). Different languages handle this differently, and rather than pick sides, JavaScript simply makes it a syntax error to omit parentheses in this case, forcing you to write an unambiguous expression.
4. The / operator divides its first operand by its second. If you are used to programming languages that distinguish between integer and floatingpoint numbers, you might expect to get an integer result when you divide one integer by another. In JavaScript, however, all numbers are floating-point, so all division operations have floating-point results: 5/2 evaluates to 2.5, not 2. Division by zero yields positive or negative infinity, while 0/0 evaluates to NaN: neither of these cases raises an error.
5. Technically, the + operator behaves like this:

If either of its operand values is an object, it converts it to a primitive using the object-to-primitive algorithm. Date objects are converted by their toString() method, and all other objects are converted via valueOf(), if that method returns a primitive value. However, most objects do not have a useful valueOf() method, so they are converted via toString() as well.

After object-to-primitive conversion, if either operand is a string, the other is converted to a string and concatenation is performed. Otherwise, both operands are converted to numbers (or to NaN) and addition is performed. Here are some examples:

1 + 2 // => 3: addition

"1" + "2" // => "12": concatenation

"1" + 2 // => "12": concatenation after number-to- string

1 + {} // => "1[object Object]"

true + true // => 2: addition after boolean-to-number

2 + null // => 2: addition after null converts to 0

2 + undefined // => NaN: addition after undefined converts to NaN

Finally, it is important to note that when the + operator is used with strings and numbers, it may not be associative. That is, the result may depend on the order in which operations are performed. For example:

1 + 2 + " blind mice" // => "3 blind mice"

1 + (2 + " blind mice") // => "12 blind mice"

The first line has no parentheses, and the + operator has left-to-right associativity, so the two numbers are added first, and their sum is concatenated with the string.

1. *Unary Arithmetic Operators* -- Unary operators modify the value of a single operand to produce a new value. In JavaScript, the unary operators all have high precedence and are all right-associative. The arithmetic unary operators described in this section (+, -, ++, and --) all convert their single operand to a number, if necessary. Note that the punctuation characters + and - are used as both unary and binary operators. The unary arithmetic operators are the following:

*Unary plus (+)* -- The unary plus operator converts its operand to a number (or to NaN) and returns that converted value. When used with an operand that is already a number, it doesn’t do anything. This operator may not be used with BigInt values, since they cannot be converted to regular numbers.

*Unary minus (-)* -- When - is used as a unary operator, it converts its operand to a number, if necessary, and then changes the sign of the result.

*Increment (++)* -- The ++ operator increments (i.e., adds 1 to) its single operand, which must be an lvalue (a variable, an element of an array, or a property of an object). The operator converts its operand to a number, adds 1 to that number, and assigns the incremented value back into the variable, element, or property. The return value of the ++ operator depends on its position relative to the operand.

*pre-increment* -- When used before the operand, it increments the operand and evaluates to the incremented value of that operand.

*post-increment* -- When used after the operand, it increments its operand but evaluates to the unincremented value of that operand.

Consider the difference between these two lines of code:

let i = 1, j = ++i; // i and j are both 2

let n = 1, m = n++; // n is 2, m is 1

Note that the expression x++ is not always the same as x=x+1. The ++ operator never performs string concatenation: it always converts its operand to a number and increments it. If x is the string “1”, ++x is the number 2, but x+1 is the string “11”.

Also note that, because of JavaScript’s automatic semicolon insertion, you cannot insert a line break between the post-increment operator and the operand that precedes it. If you do so, JavaScript will treat the operand as a complete statement by itself and insert a semicolon before it.

*Decrement (--) --* The -- operator expects an lvalue operand. It converts the value of the operand to a number, subtracts 1, and assigns the decremented value back to the operand. Like the ++ operator, the return value of -- depends on its position relative to the operand. When used before the operand, it decrements and returns the decremented value. When used after the operand, it decrements the operand but returns the undecremented value. When used after its operand, no line break is allowed between the operand and the operator.

1. *STRICT EQUALITY* -- The strict equality operator === evaluates its operands, then compares the two values as follows, performing no type conversion:

If the two values have different types, they are not equal.

If both values are null or both values are undefined, they are equal.

If both values are the boolean value true or both are the boolean value false, they are equal.

If one or both values is NaN, they are not equal. (This is surprising, but the NaN value is never equal to any other value, including itself! To check whether a value x is NaN, use x !== x, or the global isNaN() function.)

If both values are numbers and have the same value, they are equal.

If one value is 0 and the other is -0, they are also equal.

If both values are strings and contain exactly the same 16-bit values in the same positions, they are equal. If the strings differ in length or content, they are not equal. Two strings may have the same meaning and the same visual appearance, but still be encoded using different sequences of 16-bit values. JavaScript performs no Unicode normalization, and a pair of strings like this is not considered equal to the === or == operators.

If both values refer to the same object, array, or function, they are equal. If they refer to different objects, they are not equal, even if both objects have identical properties.

1. *EQUALITY WITH TYPE CONVERSION* -- The equality operator == is like the strict equality operator, but it is less strict. If the values of the two operands are not the same type, it attempts some type conversions and tries the comparison again:

If the two values have the same type, test them for strict equality as described previously. If they are strictly equal, they are equal. If they are not strictly equal, they are not equal.

If the two values do not have the same type, the == operator may still consider them equal. It uses the following rules and type conversions to check for equality:

* If one value is null and the other is undefined, they are equal.
* If one value is a number and the other is a string, convert the string to a number and try the comparison again, using the converted value.
* If either value is true, convert it to 1 and try the comparison again. If either value is false, convert it to 0 and try the comparison again.
* If one value is an object and the other is a number or string, convert the object to a primitive. An object is converted to a primitive value by either its toString() method or its valueOf() method. The built-in classes of core JavaScript attempt valueOf() conversion before toString() conversion, except for the Date class, which performs toString() conversion.
* Any other combinations of values are not equal.

As an example of testing for equality, consider the comparison:

"1" == true // => true

This expression evaluates to true, indicating that these very differentlooking values are in fact equal. The boolean value true is first converted to the number 1, and the comparison is done again. Next, the string "1" is converted to the number 1. Since both values are now the same, the comparison returns true.

1. *Comparison Operators --* Comparison and conversion occur as follows:

* If either operand evaluates to an object, that object is converted to a primitive value; if its valueOf() method returns a primitive value, that value is used. Otherwise, the return value of its toString() method is used.
* If, after any required object-to-primitive conversion, both operands are strings, the two strings are compared, using alphabetical order, where “alphabetical order” is defined by the numerical order of the 16-bit Unicode values that make up the strings.
* If, after object-to-primitive conversion, at least one operand is not a string, both operands are converted to numbers and compared numerically. 0 and -0 are considered equal. Infinity is larger than any number other than itself, and - Infinity is smaller than any number other than itself. If either operand is (or converts to) NaN, then the comparison operator always returns false. Although the arithmetic operators do not allow BigInt values to be mixed with regular numbers, the comparison operators do allow comparisons between numbers and BigInts.

1. Both the + operator and the comparison operators behave differently for numeric and string operands. + favors strings: it performs concatenation if either operand is a string. The comparison operators favor numbers and only perform string comparison if both operands are strings.
2. *The in Operator* -- The in operator expects a left-side operand that is a string, symbol, or value that can be converted to a string. It expects a right-side operand that is an object. It evaluates to true if the left-side value is the name of a property of the right-side object. For example:

let point = {x: 1, y: 1}; // Define an object

"x" in point // => true: object has property named "x"

"z" in point // => false: object has no "z" property.

"toString" in point // => true: object inherits toString method

let data = [7,8,9]; // An array with elements (indices) 0, 1, and 2

"0" in data // => true: array has an element "0"

1 in data // => true: numbers are converted to strings

3 in data // => false: no element 3

1. *The instanceof Operator* -- The instanceof operator expects a left-side operand that is an object and a right-side operand that identifies a class of objects. The operator evaluates to true if the left-side object is an instance of the right-side class and evaluates to false otherwise. In JavaScript, classes of objects are defined by the constructor function that initializes them. Thus, the right-side operand of instanceof should be a function. Here are examples:

let d = new Date(); // Create a new object with the Date() constructor d instanceof Date // => true: d was created with Date()

d instanceof Object // => true: all objects are instances of Object

d instanceof Number // => false: d is not a Number object

let a = [1, 2, 3]; // Create an array with array literal syntax

a instanceof Array // => true: a is an array

a instanceof Object // => true: all arrays are objects

a instanceof RegExp // => false: arrays are not regular expressions

Note that all objects are instances of Object. instanceof considers the “superclasses” when deciding whether an object is an instance of a class. If the left-side operand of instanceof is not an object, instanceof returns false. If the righthand side is not a class of objects, it throws a TypeError.

1. *Logical AND (&&) --* This operator starts by evaluating its first operand, the expression on its left. If the value on the left is falsy, the value of the entire expression must also be falsy, so && simply returns the value on the left and does not even evaluate the expression on the right.

On the other hand, if the value on the left is truthy, then the overall value of the expression depends on the value on the righthand side. If the value on the right is truthy, then the overall value must be truthy, and if the value on the right is falsy, then the overall value must be falsy. So when the value on the left is truthy, the && operator evaluates and returns the value on the right:

let o = {x: 1};

let p = null;

o && o.x // => 1: o is truthy, so return value of o.x

p && p.x // => null: p is falsy, so return it and don't evaluate p.x

The behavior of && is sometimes called short circuiting, and you may sometimes see code that purposely exploits this behavior to conditionally execute code. For example, the following two lines of JavaScript code have equivalent effects:

if (a === b) stop(); // Invoke stop() only if a === b

(a === b) && stop(); // This does the same thing

1. *Assignment Expressions* -- JavaScript uses the = operator to assign a value to a variable or property. Although assignment expressions are usually quite simple, you may sometimes see the value of an assignment expression used as part of a larger expression. For example, you can assign and test a value in the same expression with code like this:

(a = b) === 0 // assign b to a and then check whether a===0

The assignment operator has right-to-left associativity, which means that when multiple assignment operators appear in an expression, they are evaluated from right to left. Thus, you can write code like this to assign a single value to multiple variables:

i = j = k = 0; // Initialize 3 variables to 0

1. *Assignment with Operation* -- In most cases, the expression:

a op= b where op is an operator, is equivalent to the expression:

a = a op b

In the first line, the expression a is evaluated once. In the second, it is evaluated twice. The two cases will differ only if a includes side effects such as a function call or an increment operator. The following two assignments, for example, are not the same:

let data=[3,2],i=0;

data[i++] \*= 2; // => [6, 2]

let data=[3,2],i=0;

data[i++] = data[i++] \* 2; // => [4, 2]

1. *Evaluation Expressions* -- Like many interpreted languages, JavaScript has the ability to interpret strings of JavaScript source code, evaluating them to produce a value. JavaScript does this with the global function eval():

eval("3+2") // => 5

In particular, eval() can be a security hole, and you should never pass any string derived from user input to eval().

eval() is a function, but it is included in this chapter on expressions because it really should have been an operator. The earliest versions of the language defined an eval() function, and ever since then, language designers and interpreter writers have been placing restrictions on it that make it more and more operator-like. Modern JavaScript interpreters perform a lot of code analysis and optimization. Generally speaking, if a function calls eval(), the interpreter cannot optimize that function. The problem with defining eval() as a function is that it can be given other names: let f = eval; let g = f; If this is allowed, then the interpreter can’t know for sure which functions call eval(), so it cannot optimize aggressively.

1. *eval()* -- eval() expects one argument. If you pass any value other than a string, it simply returns that value. If you pass a string, it attempts to parse the string as JavaScript code, throwing a SyntaxError if it fails. If it successfully parses the string, then it evaluates the code and returns the value of the last expression or statement in the string or undefined if the last expression or statement had no value. If the evaluated string throws an exception, that exception propogates from the call to eval().

The key thing about eval() (when invoked like this) is that it uses the variable environment of the code that calls it. That is, it looks up the values of variables and defines new variables and functions in the same way that local code does. If a function defines a local variable x and then calls eval("x"), it will obtain the value of the local variable. If it calls eval("x=1"), it changes the value of the local variable. And if the function calls eval("var y = 3;"), it declares a new local variable y. On the other hand, if the evaluated string uses let or const, the variable or constant declared will be local to the evaluation and will not be defined in the calling environment.

Similarly, a function can declare a local function with code like this: eval("function f() { return x+1; }"); If you call eval() from top-level code, it operates on global variables and global functions, of course.

1. *Global eval()* -- It is the ability of eval() to change local variables that is so problematic to JavaScript optimizers. As a workaround, however, interpreters simply do less optimization on any function that calls eval(). But what should a JavaScript interpreter do, however, if a script defines an alias for eval() and then calls that function by another name? The JavaScript specification declares that when eval() is invoked by any name other than “eval”, it should evaluate the string as if it were top-level global code. The evaluated code may define new global variables or global functions, and it may set global variables, but it will not use or modify any variables local to the calling function, and will not, therefore, interfere with local optimizations.

A “direct eval” is a call to the eval() function with an expression that uses the exact, unqualified name “eval” (which is beginning to feel like a reserved word). Direct calls to eval() use the variable environment of the calling context. Any other call—an indirect call— uses the global object as its variable environment and cannot read, write, or define local variables or functions. (Both direct and indirect calls can define new variables only with var. Uses of let and const inside an evaluated string create variables and constants that are local to the evaluation and do not alter the calling or global environment.) The following code demonstrates:

const geval = eval; // Using another name does a global eval

let x = "global", y = "global"; // Two global variables

function f() { // This function does a local eval

let x = "local"; // Define a local variable

eval("x += 'changed';"); // Direct eval sets local variable

return x; // Return changed local variable

}

function g() { // This function does a global eval

let y = "local"; // A local variable

geval("y += 'changed';"); // Indirect eval sets global variable

return y; // Return unchanged local variable

}

console.log(f(), x); //Local var changed:prints "localchanged global"

console.log(g(), y); // Global var changed: prints "local globalchanged"

1. *First-Defined (??)* -- The first-defined operator ?? evaluates to its first defined operand: if its left operand is not null and not undefined, it returns that value. Otherwise, it returns the value of the right operand. Like the && and || operators, ?? is short-circuiting: it only evaluates its second operand if the first operand evaluates to null or undefined. If the expression a has no side effects, then the expression a ?? b is equivalent to: (a !== null && a !== undefined) ? a : b
2. *The delete Operator* -- Note that a deleted property or array element is not merely set to the undefined value. When a property is deleted, the property ceases to exist. Attempting to read a nonexistent property returns undefined, but you can test for the actual existence of a property with the in operator. Deleting an array element leaves a “hole” in the array and does not change the array’s length. The resulting array is sparse.

delete expects its operand to be an lvalue. If it is not an lvalue, the operator takes no action and returns true. Otherwise, delete attempts to delete the specified lvalue. delete returns true if it successfully deletes the specified lvalue. Not all properties can be deleted, however: non-configurable properties are immune from deletion.

In strict mode, delete raises a SyntaxError if its operand is an unqualified identifier such as a variable, function, or function parameter: it only works when the operand is a property access expression. Strict mode also specifies that delete raises a TypeError if asked to delete any non-configurable (i.e., nondeleteable) property. Outside of strict mode, no exception occurs in these cases, and delete simply returns false to indicate that the operand could not be deleted. Here are some example uses of the delete operator:

let o = {x: 1, y: 2};

delete o.x; // Delete one of the object properties; returns true.

typeof o.x; // Property does not exist; returns "undefined".

delete o.x; // Delete a nonexistent property; returns true.

delete 1; // This makes no sense, but it just returns true.

delete o; // Can't delete a variable; returns false,

or SyntaxError in strict mode.

delete Object.prototype; // Undeletable property: returns false,

or TypeError in strict mode.

1. *The void Operator* -- void is a unary operator that appears before its single operand, which may be of any type. This operator is unusual and infrequently used; it evaluates its operand, then discards the value and returns undefined. Since the operand value is discarded, using the void operator makes sense only if the operand has side effects. The void operator is so obscure that it is difficult to come up with a practical example of its use. One case would be when you want to define a function that returns nothing but also uses the arrow function shortcut syntax where the body of the function is a single expression that is evaluated and returned. If you are evaluating the expression solely for its side effects and do not want to return its value, then the simplest thing is to use curly braces around the function body. But, as an alternative, you could also use the void operator in this case:

let counter = 0;

const increment = () => void counter++;

increment() // => undefined

counter // => 1

1. *The comma Operator (,)* -- The comma operator is a binary operator whose operands may be of any type. It evaluates its left operand, evaluates its right operand, and then returns the value of the right operand. Thus, the following line:

i=0, j=1, k=2; evaluates to 2 and is basically equivalent to:

i = 0; j = 1; k = 2;

The lefthand expression is always evaluated, but its value is discarded, which means that it only makes sense to use the comma operator when the lefthand expression has side effects. The only situation in which the comma operator is commonly used is with a for loop that has multiple loop variables.

console.log((1,2,3,4,5)) // => 5

1. *Compound and Empty Statements* -- A compound statement allows you to use multiple statements where JavaScript syntax expects a single statement. The empty statement is the opposite: it allows you to include no statements where one is expected. The empty statement looks like this: ; The JavaScript interpreter takes no action when it executes an empty statement. The empty statement is occasionally useful when you want to create a loop that has an empty body. Consider the following for loop :

// Initialize an array a

for(let i = 0; i < a.length; a[i++] = 0) ;

In this loop, all the work is done by the expression a[i++] = 0, and no loop body is necessary. JavaScript syntax requires a statement as a loop body, however, so an empty statement—just a bare semicolon—is used.

Note that the accidental inclusion of a semicolon after the right parenthesis of a for loop, while loop, or if statement can cause frustrating bugs that are difficult to detect. For example, the following code probably does not do what the author intended:

if ((a === 0) || (b === 0)); // Oops! This line does nothing...

o = null; // and this line is always executed.

1. *for* -- In all our loop examples so far, the loop variable has been numeric. This is quite common but is not necessary. The following code uses a for loop to traverse a linked list data structure and return the last object in the list (i.e., the first object that does not have a next property):

function tail(o) { // Return the tail of linked list o

for(; o.next; o = o.next) ; // Traverse while o.next is truthy

return o;

}

Note that this code has no initialize expression. Any of the three expressions may be omitted from a for loop, but the two semicolons are required. If you omit the test expression, the loop repeats forever, and for(;;) is another way of writing an infinite loop, like while(true).

1. *for/of* -- The for/of loop works with iterable objects. Arrays, strings, sets, and maps are iterable: they represent a sequence or set of elements that you can loop or iterate through using a for/of loop. Here, for example, is how we can use for/of to loop through the elements of an array of numbers and compute their sum:

let data = [1, 2, 3, 4, 5, 6, 7, 8, 9], sum = 0;

for(let element of data) { sum += element; } sum // => 45

Arrays are iterated “live”—changes made during the iteration may affect the outcome of the iteration.

Objects are not (by default) iterable. Attempting to use for/of on a regular object throws a TypeError at runtime.

Note that strings are iterated by Unicode codepoint, not by UTF-16 character. The string “I ❤ 🐈” has a .length of 5 (because the two emoji characters each require two UTF-16 characters to represent). But if you iterate that string with for/of, the loop body will run three times, once for each of the three code points “I”, “❤”, and “🐈.”

1. *for/in* -- A for/in loop looks a lot like a for/of loop, with the of keyword changed to in. While a for/of loop requires an iterable object after the of, a for/in loop works with any object after the in. The for/of loop is new in ES6, but for/in has been part of JavaScript since the very beginning (which is why it has the more natural sounding syntax). The for/in statement loops through the property names of a specified object. The syntax looks like this: for (variable in object)

statement variable typically names a variable, but it may be a variable declaration or anything suitable as the left-hand side of an assignment expression. object is an expression that evaluates to an object. As usual, statement is the statement or statement block that serves as the body of the loop.

To execute a for/in statement, the JavaScript interpreter first evaluates the object expression. If it evaluates to null or undefined, the interpreter skips the loop and moves on to the next statement. The interpreter now executes the body of the loop once for each enumerable property of the object. Before each iteration, however, the interpreter evaluates the variable expression and assigns the name of the property (a string value) to it.

Note that the variable in the for/in loop may be an arbitrary expression, as long as it evaluates to something suitable for the left side of an assignment. This expression is evaluated each time through the loop, which means that it may evaluate differently each time. For example, you can use code like the following to copy the names of all object properties into an array:

let o = { x: 1, y: 2, z: 3 };

let a = [], i = 0;

for(a[i++] in o) ;

The for/in loop does not actually enumerate all properties of an object. It does not enumerate properties whose names are symbols. And of the properties whose names are strings, it only loops over the enumerable properties. The various built-in methods defined by core JavaScript are not enumerable. All objects have a toString() method, for example, but the for/in loop does not enumerate this toString property. In addition to built-in methods, many other properties of the built-in objects are non-enumerable. All properties and methods defined by your code are enumerable, by default. Enumerable inherited properties are also enumerated by the for/in loop. This means that if you use for/in loops and also use code that defines properties that are inherited by all objects, then your loop may not behave in the way you expect. For this reason, many programmers prefer to use a for/of loop with Object.keys() instead of a for/in loop. If the body of a for/in loop deletes a property that has not yet been enumerated, that property will not be enumerated. If the body of the loop defines new properties on the object, those properties may or may not be enumerated.

1. *Labeled Statements* -- Any statement may be labeled by preceding it with an identifier and a colon:

identifier: statement

By labeling a statement, you give it a name that you can use to refer to it elsewhere in your program. You can label any statement, although it is only useful to label statements that have bodies, such as loops and conditionals. By giving a loop a name, you can use break and continue statements inside the body of the loop to exit the loop or to jump directly to the top of the loop to begin the next iteration. *break and continue are the only JavaScript statements that use statement labels*. Here is an example of a labeled while loop and a continue statement that uses the label.

mainloop: while(token !== null) { continue mainloop; }

The identifier you use to label a statement can be any legal JavaScript identifier that is not a reserved word. The namespace for labels is different than the namespace for variables and functions, so you can use the same identifier as a statement label and as a variable or function name. Statement labels are defined only within the statement to which they apply (and within its substatements, of course). A statement may not have the same label as a statement that contains it, but two statements may have the same label as long as neither one is nested within the other. Labeled statements may themselves be labeled. Effectively, this means that any statement may have multiple labels.

1. *break* -- The break statement, used alone, causes the innermost enclosing loop or switch statement to exit immediately. Because it causes a loop or switch to exit, this form of the break statement is legal only if it appears inside one of these statements.

When break is used with a label, it jumps to the end of, or terminates, the enclosing statement that has the specified label. It is a syntax error to use break in this form if there is no enclosing statement with the specified label. With this form of the break statement, the named statement need not be a loop or switch: break can “break out of” any enclosing statement. This statement can even be a statement block grouped within curly braces for the sole purpose of naming the block with a label. A newline is not allowed between the break keyword and the labelname. This is a result of JavaScript’s automatic insertion of omitted semicolons.

x: {

c=1;

console.log("g");

forOne: for(let i=0;i<10;i++)

forTwo: for(let j=0;j<8;j++){ break x;}

console.log("c");

} // => prints g

1. Like the break statement, the continue statement can be used in its labeled form within nested loops when the loop to be restarted is not the immediately enclosing loop. Also, as with the break statement, line breaks are not allowed between the continue statement and its labelname.
2. *throw* -- The throw statement has the following syntax: throw expression; expression may evaluate to a value of any type. You might throw a number that represents an error code or a string that contains a human readable error message. The Error class and its subclasses are used when the JavaScript interpreter itself throws an error, and you can use them as well. An Error object has a name property that specifies the type of error and a message property that holds the string passed to the constructor function.

throw new Error("x must not be negative");

When an exception is thrown, the JavaScript interpreter immediately stops normal program execution and jumps to the nearest exception handler. Exception handlers are written using the catch clause of the try/catch/finally statement. If the block of code in which the exception was thrown does not have an associated catch clause, the interpreter checks the next highest enclosing block of code to see if it has an exception handler associated with it. This continues until a handler is found. If an exception is thrown in a function that does not contain a try/catch/finally statement to handle it, the exception propagates up to the code that invoked the function. In this way, exceptions propagate up through the lexical structure of JavaScript methods and up the call stack. If no exception handler is ever found, the exception is treated as an error and is reported to the user.

1. *BARE CATCH CLAUSES* -- Occasionally you may find yourself using a catch clause solely to detect and stop the propagation of an exception, even though you do not care about the type or the value of the exception. In ES2019 and later, you can omit the parentheses and the identifier and use the catch keyword bare in this case.
2. *With* -- The with statement runs a block of code as if the properties of a specified object were variables in scope for that code. It has the following syntax: with (object) statement This statement creates a temporary scope with the properties of object as variables and then executes statement within that scope.
3. *“use strict” --* "use strict" is a directive introduced in ES5. Directives are not statements (but are close enough that "use strict" is documented here). There are two important differences between the "use strict" directive and regular statements: It does not include any language keywords: the directive is just an expression statement that consists of a special string literal (in single or double quotes). It can appear only at the start of a script or at the start of a function body, before any real statements have appeared.

In addition to code explicitly declared to be strict, any code in a class body or in an ES6 module is automatically strict code.

The with statement is not allowed in strict mode. All variables must be declared, functions invoked as functions (rather than as methods) have a this value of undefined. (In non-strict mode, global object as their this value.) Also, in strict mode, when a function is invoked with call() or apply(), the this value is exactly the value passed as the first argument to call() or apply().

In strict mode, assignments to nonwritable properties and attempts to create new properties on non-extensible objects throw a TypeError. (In non-strict mode, these attempts fail silently.)

In strict mode, code passed to eval() cannot declare variables or define functions in the caller’s scope as it can in non-strict mode. Instead, variable and function definitions live in a new scope created for the eval(). This scope is discarded when the eval() returns.

In strict mode, the Arguments object in a function holds a static copy of the values passed to the function. In nonstrict mode, the Arguments object has “magical” behavior in which elements of the array and named function parameters both refer to the same value.

In strict mode, a SyntaxError is thrown if the delete operator is followed by an unqualified identifier such as a variable, function, or function parameter. (In nonstrict mode, such a delete expression does nothing and evaluates to false.)

In strict mode, an attempt to delete a nonconfigurable property throws a TypeError. (In non-strict mode, the attempt fails and the delete expression evaluates to false.)

In strict mode, it is a syntax error for an object literal to define two or more properties by the same name. (In non-strict mode, no error occurs.)

In strict mode, it is a syntax error for a function declaration to have two or more parameters with the same name. (In nonstrict mode, no error occurs.)

In strict mode, octal integer literals (beginning with a 0 that is not followed by an x) are not allowed. (In non-strict mode, some implementations allow octal literals.)

In strict mode, the identifiers eval and arguments are treated like keywords, and you are not allowed to change their value. You cannot assign a value to these identifiers, declare them as variables, use them as function names, use them as function parameter names, or use them as the identifier of a catch block.

In strict mode, the ability to examine the call stack is restricted. arguments.caller and arguments.callee both throw a TypeError within a strict mode function. Strict mode functions also have caller and arguments properties that throw TypeError when read.

1. *Objects* -- It is sometimes important to be able to distinguish between properties defined directly on an object and those that are inherited from a prototype object. JavaScript uses the term own property to refer to non-inherited properties. In addition to its name and value, each property has three property attributes:

The *writable* attribute specifies whether the value of the property can be set.

The *enumerable* attribute specifies whether the property name is returned by a for/in loop.

The *configurable* attribute specifies whether the property can be deleted and whether its attributes can be altered.

Many of JavaScript’s built-in objects have properties that are read-only, non-enumerable, or non-configurable. By default, however, all properties of the objects you create are writable, enumerable, and configurable.

1. Objects can be created with object literals, with the new keyword, and with the Object.create() function.

*Inheritance* -- JavaScript objects have a set of “own properties,” and they also inherit a set of properties from their prototype object. To understand this, we must consider property access in more detail. The examples in this section use the Object.create() function to create objects with specified prototypes. However, that every time you create an instance of a class with new, you are creating an object that inherits properties from a prototype object.

Suppose you query the property x in the object o. If o does not have an own property with that name, the prototype object of o is queried for the property x. If the prototype object does not have an own property by that name, but has a prototype itself, the query is performed on the prototype of the prototype. This continues until the property x is found or until an object with a null prototype is searched. As you can see, the prototype attribute of an object creates a chain or linked list from which properties are inherited:

let o = {}; // o inherits object methods from Object.prototype

o.x = 1; // and it now has an own property x.

let p =Object.create(o); //p inherits from o and Object.prototype

p.y = 2; // and has an own property y.

let q = Object.create(p); // q inherits properties from p, o, and...

q.z = 3; // ...Object.prototype and has an own property z.

let f = q.toString(); // toString is inherited from Object.prototype

q.x + q.y // => 3; x and y are inherited from o and p

Now suppose you assign to the property x of the object o. If o already has an own (non-inherited) property named x, then the assignment simply changes the value of this existing property. Otherwise, the assignment creates a new property named x on the object o. If o previously inherited the property x, that inherited property is now hidden by the newly created own property with the same name.

Property assignment examines the prototype chain only to determine whether the assignment is allowed. If o inherits a read-only property named x, for example, then the assignment is not allowed. If the assignment is allowed, however, it always creates or sets a property in the original object and never modifies objects in the prototype chain. The fact that inheritance occurs when querying properties but not when setting them is a key feature of JavaScript because it allows us to selectively override inherited properties:

let unitcircle = { r: 1 }; // An object to inherit from

let c = Object.create(unitcircle); // c inherits the property r

c.x = 1; c.y = 1; // c defines two properties of its own

c.r = 2; // c overrides its inherited property

unitcircle.r // => 1: the prototype is not affected

There is one exception to the rule that a property assignment either fails or creates or sets a property in the original object. If o inherits the property x, and that property is an accessor property with a setter method, then that setter method is called rather than creating a new property x in o. Note, however, that the setter method is called on the object o, not on the prototype object that defines the property, so if the setter method defines any properties, it will do so on o, and it will again leave the prototype chain unmodified.

An attempt to set a property p of an object o fails in these circumstances:

* has an own property p that is read-only: it is not possible to set read-only properties.
* o has an inherited property p that is read-only: it is not possible to hide an inherited read-only property with an own property of the same name.
* o does not have an own property p; o does not inherit a property p with a setter method, and o’s extensible attribute is false. Since p does not already exist in o, and if there is no setter method to call, then p must be added to o. But if o is not extensible, then no new properties can be defined on it.

1. The delete operator only deletes own properties, not inherited ones. (To delete an inherited property, you must delete it from the prototype object in which it is defined. Doing this affects every object that inherits from that prototype.)

A delete expression evaluates to true if the delete succeeded or if the delete had no effect (such as deleting a nonexistent property). delete also evaluates to true when used (meaninglessly) with an expression that is not a property access expression:

let o = {x: 1}; // o has own property x and inherits property toString delete o.x // => true: deletes property x

delete o.x // => true: does nothing (x doesn't exist) but true anyway delete o.toString //=>true:does nothing (toString isn't an own property)

delete 1 // => true: nonsense, but true anyway

delete does not remove properties that have a configurable attribute of false. Certain properties of built-in objects are non-configurable, as are properties of the global object created by variable declaration and function declaration. In strict mode, attempting to delete a nonconfigurable property causes a TypeError. In non-strict mode, delete simply evaluates to false in this case:

// In strict mode, all these deletions throw TypeError instead of returning false delete

Object.prototype // => false: property is nonconfigurable

var x = 1; // Declare a global variable

delete globalThis.x // => false: can't delete this property

function f() {} // Declare a global function

delete globalThis.f // => false: can't delete this property either

When deleting configurable properties of the global object in non-strict mode, you can omit the reference to the global object and simply follow the delete operator with the property name:

globalThis.x = 1; // Create a configurable global property (no let or var) delete x // => true: this property can be deleted

In strict mode, however, delete raises a SyntaxError if its operand is an unqualified identifier like x, and you have to be explicit about the property access:

delete x; // SyntaxError in strict mode delete

globalThis.x; // This works

1. *Testing Properties* – test properties with in operator, with the hasOwnProperty() and propertyIsEnumerable() methods, or simply by querying the property. The in operator expects a property name on its left side and an object on its right. in can distinguish between properties that do not exist and properties that exist but have been set to undefined

let o = { x: 1, k: undefined };

"x" in o // => true: o has an own property "x"

"y" in o // => false: o doesn't have a property "y"

"toString" in o // => true: o inherits a toString property

"k" in o // = > true

The hasOwnProperty() method of an object tests whether that object has an own property with the given name. It returns false for inherited properties:

let o = { x: 1 };

o.hasOwnProperty("x") // => true: o has an own property x o.hasOwnProperty("y") // => false: o doesn't have a property y o.hasOwnProperty("toString") // => false: toString is inherited

The propertyIsEnumerable() refines the hasOwnProperty() test. It returns true only if the named property is an own property and its enumerable attribute is true. Certain built-in properties are not enumerable. Properties created by normal JavaScript code are enumerable

let o = { x: 1 };

o.propertyIsEnumerable("x") // => true:

o.propertyIsEnumerable("toString") // => false: not an own property Object.prototype.propertyIsEnumerable("toString")//=>false:not enu..

1. *Object.keys()* -- returns an array of the names of the enumerable own properties of an object. It does not include non-enumerable properties, inherited properties, or properties whose name is a Symbol.

*Object.getOwnPropertyNames()* -- works like Object.keys() but returns an array of the names of nonenumerable own properties as well, as long as their names are strings.

*Object.getOwnPropertySymbols()* -- returns own properties whose names are Symbols, whether or not they are enumerable.//only symbols

*Reflect.ownKeys()* -- returns all own property names, both enumerable and non-enumerable, and both string and Symbol.

1. *Property Enumeration Order* -- ES6 formally defines the order in which the own properties of an object are enumerated. Object.keys(), Object.getOwnPropertyNames(), Object.getOwnPropertySymbols(), Reflect.ownKeys(), and related methods such as JSON.stringify() all list properties in the following order,

* String properties whose names are non-negative integers are listed first, in numeric order from smallest to largest. This rule means that arrays and array-like objects will have their properties enumerated in order.
* After all properties that look like array indexes are listed, all remaining properties with string names are listed (including properties that look like negative numbers or floating-point numbers). These properties are listed in the order in which they were added to the object. For properties defined in an object literal, this order is the same order they appear in the literal.
* Finally, the properties whose names are Symbol objects are listed in the order in which they were added to the object.

1. *Object.assign()* -- expects two or more objects as its arguments. It modifies and returns the first argument, which is the target object, but does not alter the second or any subsequent arguments, which are the source objects. For each source object, it copies the enumerable own properties of that object (including those whose names are Symbols) into the target object. It processes the source objects in argument list order so that properties in the first source object override properties by the same name in the target object and properties in the second source object (if there is one) override properties with the same name in the first source object.

Object.assign() copies properties with ordinary property get and set operations, so if a source object has a getter method or the target object has a setter method, they will be invoked during the copy, but they will not themselves be copied.

Object.assign(o, defaults); // overwrites everything in o with defaults Instead, what you can do is to create a new object, copy the defaults into it, and then override those defaults with the properties in o:

o = Object.assign({}, defaults, o);

you can also express this object copy-and override operation using the ... spread operator like this: o = {...defaults, ...o};

1. *Object Methods* – All JavaScript objects (except those explicitly created without a prototype) inherit properties from Object.prototype.

*The toString() Method* -- The toString() method takes no arguments; it returns a string that somehow represents the value of the object on which it is invoked. JavaScript invokes this method of an object whenever it needs to convert the object to a string.

let s = { x: 1, y: 1 }.toString(); // s == "[object Object]"

*The toLocaleString() Method* -- The purpose of this method is to return a localized string representation of the object. The default toLocaleString() method defined by Object doesn’t do any localization itself: it simply calls toString() and returns that value. The Date and Number classes define customized versions of toLocaleString() that attempt to format numbers, dates, and times according to local conventions.

*The valueOf() Method* -- called when JavaScript needs to convert an object to some primitive type other than a string—typically, a number. JavaScript calls this method automatically if an object is used in a context where a primitive value is required. The default valueOf() method does nothing interesting, but some of the built-in classes define their own valueOf() method. The Date class defines valueOf() to convert dates to numbers, and this allows Date objects to be chronologically compared with < and >.

*The toJSON() Method* -- Object.prototype does not actually define a toJSON() method, but the JSON.stringify() method looks for a toJSON() method on any object it is asked to serialize. If this method exists on the object to be serialized, it is invoked, and the return value is serialized, instead of the original object. The Date class defines a toJSON() method that returns a serializable string representation of the date.

1. *Spread Operator* -- In ES2018 and later, you can copy the properties of an existing object into a new object using the “spread operator”. Note that this ... syntax is often called a spread operator but is not a true JavaScript operator in any sense. Instead, it is a special-case syntax available only within object literals. (Three dots are used for other purposes in other JavaScript contexts, but object literals are the only context where the three dots cause this kind of interpolation of one object into another one.). Also note that the spread operator only spreads the own properties of an object, not any inherited ones:

let o = Object.create({x: 1}); // o inherits the property x

let p = { ...o }; p.x // => undefined

Finally, it is worth noting that, although the spread operator is just three little dots in your code, it can represent a substantial amount of work to the JavaScript interpreter. If an object has n properties, the process of spreading those properties into another object is likely to be an O(n) operation. This means that if you find yourself using ... within a loop or recursive function as a way to accumulate data into one large object, you may be writing an inefficient O(n ) algorithm that will not scale well as n gets larger.

1. *Shorthand Methods* -- When a function is defined as a property of an object, we call that function a method. Prior to ES6, you would define a method in an object literal using a function definition expression just as you would define any other property of an object:

let square = { area: function() { return this.side \* this.side; }, side: 10 }; square.area() // => 100

In ES6, however, the object literal syntax has been extended to allow a shortcut where the function keyword and the colon are omitted, resulting in code like this:

let square = { area() { return this.side \* this.side; }, side: 10 };

you can also use string literals and computed property names, which can include Symbol property names:

const METHOD\_NAME = "m";

const symbol = Symbol();

let weirdMethods = {

"method With Spaces"(x) { return x + 1; },

[METHOD\_NAME](x) { return x + 2; },

[symbol](x) { return x + 3; } };

weirdMethods["method With Spaces "](1) // => 2 weirdMethods[METHOD\_NAME](1) // => 3

weirdMethods[symbol](1) // => 4

Using a Symbol as a method name is not as strange as it seems. In order to make an object iterable (so it can be used with a for/of loop), you must define a method with the symbolic name.

1. *Property Getters and Setters* -- All of the object properties we’ve discussed so far in this chapter have been data properties with a name and an ordinary value. JavaScript also supports accessor properties, which do not have a single value but instead have one or two accessor methods: a getter and/or a setter. When a program queries the value of an accessor property, JavaScript invokes the getter method (passing no arguments). The return value of this method becomes the value of the property access expression. When a program sets the value of an accessor property, JavaScript invokes the setter method, passing the value of the righthand side of the assignment. This method is responsible for “setting,” in some sense, the property value. The return value of the setter method is ignored. If a property has both a getter and a setter method, it is a read/write property. If it has only a getter method, it is a read-only property. And if it has only a setter method, it is a write-only property and attempts to read it always evaluate to undefined.

Accessor properties can be defined with an extension to the object literal syntax

let o = {

dataProp: value, // An ordinary data property

// An accessor property defined as a pair of functions.

get accessorProp() { return this.dataProp; },

set accessorProp(value) { this.dataProp = value; } };

Accessor properties are defined as one or two methods whose name is the same as the property name. These look like ordinary methods defined using the ES6 shorthand except that getter and setter definitions are prefixed with get or set. (In ES6, you can also use computed property names when defining getters and setters. Simply replace the property name after get or set with an expression in square brackets.) The accessor methods defined above simply get and set the value of a data property, and there is no reason to prefer the accessor property over the data property. But as a more interesting example, consider the following object that represents a 2D Cartesian point. It has ordinary data properties to represent the x and y coordinates of the point, and it has accessor properties that give the equivalent polar coordinates of the point:

let p = {

x: 1.0, y: 1.0, // x and y are regular read-write data properties.

// r is a read-write accessor property with getter and setter.

get r() { return Math.hypot(this.x, this.y); },

set r(newvalue) {

let oldvalue = Math.hypot(this.x, this.y);

let ratio = newvalue/oldvalue;

this.x \*= ratio; this.y \*= ratio; },

// theta is a read-only accessor property with getter only.

get theta() { return Math.atan2(this.y, this.x); } };

p.r // => Math.SQRT2

p.theta // => Math.PI / 4

Accessor properties are inherited, just as data properties are, so you can use the object p defined above as a prototype for other points.

let q = Object.create(p); // A new object that inherits getters and setters q.x = 3; q.y = 4; // Create q's own data properties

q.r // => 5: the inherited accessor properties work

q.theta // => Math.atan2(4, 3)

The code above uses accessor properties to define an API that provides two representations (Cartesian coordinates and polar coordinates) of a single set of data.

Other reasons to use accessor properties include sanity checking of property writes and returning different values on each property read:

// This object generates strictly increasing serial numbers

const serialnum = { // This data property holds the next serial number. // The \_ in the property name hints that it is for internal use only.

\_n: 0,

// Return the current value and increment it

get next() { return this.\_n++; },

// Set a new value of n, but only if it is larger than current

set next(n) { if (n > this.\_n) this.\_n = n;

else throw new Error("serial number can only be set to a larger value"); } };

serialnum.next = 10; // Set the starting serial number

serialnum.next // => 10

serialnum.next // => 11: different value each time we get next

Finally, here is one more example that uses a getter method to implement a property with “magical” behavior:

// This object has accessor properties that return random numbers.

// The expression "random.octet", for example, yields a random number // between 0 and 255 each time it is evaluated.

const random = {

get octet() { return Math.floor(Math.random()\*256); },

get uint16() { return Math.floor(Math.random()\*65536); },

get int16() { return Math.floor(Math.random()\*65536)-32768; } };

1. *Array() Constructor* -- You can invoke this constructor in three distinct ways: Call it with no arguments: let a = new Array(); This method creates an empty array with no elements and is equivalent to the array literal []. Call it with a single numeric argument, which specifies a length: let a = new Array(10); This technique creates an array with the specified length. This form of the Array() constructor can be used to preallocate an array when you know in advance how many elements will be required. Note that no values are stored in the array, and the array index properties “0”, “1”, and so on are not even defined for the array. Explicitly specify two or more array elements or a single nonnumeric element for the array: let a = new Array(5, 4, 3, 2, 1, "testing, testing");
2. *Array.of()* -- When the Array() constructor function is invoked with one numeric argument, it uses that argument as an array length. But when invoked with more than one numeric argument, it treats those arguments as elements for the array to be created. This means that the Array() constructor cannot be used to create an array with a single numeric element. In ES6, the Array.of() function addresses this problem: it is a factory method that creates and returns a new array, using its argument values (regardless of how many of them there are) as the array elements.
3. *Array.from()* -- Array.from is another array factory method introduced in ES6. It expects an iterable or array-like object as its first argument and returns a new array that contains the elements of that object. With an iterable argument, Array.from(iterable) works like the spread operator [...iterable] does. It is also a simple way to make a copy of an array: let copy = Array.from(original);

Array.from() is also important because it defines a way to make a true-array copy of an array-like object. Array-like objects are non-array objects that have a numeric length property and have values stored with properties whose names happen to be integers. When working with client-side JavaScript, the return values of some web browser methods are array-like, and it can be easier to work with them if you first convert them to true arrays: let truearray = Array.from(arraylike);

Array.from() also accepts an optional second argument. If you pass a function as the second argument, then as the new array is being built, each element from the source object will be passed to the function you specify, and the return value of the function will be stored in the array instead of the original value.

1. Note that you can index an array using numbers that are negative or that are not integers. When you do this, the number is converted to a string, and that string is used as the property name. Since the name is not a non-negative integer, it is treated as a regular object property, not an array index. Also, if you index an array with a string that happens to be a non-negative integer, it behaves as an array index, not an object property. The same is true if you use a floating-point number that is the same as an integer:

a[-1.23] = true; // This creates a property named "-1.23"

a["1000"] = 0; // This the 1001st element of the array

a[1.000] = 1; // Array index 1. Same as a[1] = 1;

console.log(a) //=>[<1 empty>,1,<998 empty>, 0, '-1.23': true ]

The fact that array indexes are simply a special type of object property name means that JavaScript arrays have no notion of an “out of bounds” error. When you try to query a nonexistent property of any object, you don’t get an error; you simply get undefined. This is just as true for arrays as it is for objects.

Note that when you omit a value in an array literal (using repeated commas as in [1,,3]), the resulting array is sparse, and the omitted elements simply do not exist:

let a1 = [,]; // This array has no elements and length 1

let a2 = [undefined]; // This array has one undefined element

0 in a1 // => false: a1 has no element with index 0

0 in a2 // => true: a2 has the undefined value at index 0

1. If you set the length property to a nonnegative integer n smaller than its current value, any array elements whose index is greater than or equal to n are deleted from the array:

a = [1,2,3,4,5]; // Start with a 5-element array.

a.length = 3; // a is now [1,2,3].

a.length = 0; // Delete all elements. a is [].

a.length = 5; // Length is 5, but no elements, like new Array(5)

You can also set the length property of an array to a value larger than its current value. Doing this does not actually add any new elements to the array; it simply creates a sparse area at the end of the array.

1. *Array Iterator Methods* -- The methods described in this section iterate over arrays by passing array elements, in order, to a function you supply, and they provide convenient ways to iterate, map, filter, test, and reduce arrays. First, all of these methods accept a function as their first argument and invoke that function once for each element (or some elements) of the array. If the array is sparse, the function you pass is not invoked for nonexistent elements. In most cases, the function you supply is invoked with three arguments: the value of the array element, the index of the array element, and the array itself.

Most of the iterator methods described in the following subsections accept an optional second argument. If specified, the function is invoked as if it is a method of this second argument. That is, the second argument you pass becomes the value of the this keyword inside of the function you pass as the first argument. The return value of the function you pass is usually important, but different methods handle the return value in different ways. None of the methods described here modify the array on which they are invoked (though the function you pass can modify the array, of course)

*FOREACH()* -- The forEach() method iterates through an array, invoking a function you specify for each element. As we’ve described, you pass the function as the first argument to forEach(). forEach() then invokes your function with three arguments: the value of the array element, the index of the array element, and the array itself. If you only care about the value of the array element, you can write a function with only one parameter—the additional arguments will be ignored:

let data = [1,2,3,4,5], sum = 0;

// Compute the sum of the elements of the array

data.forEach(value => { sum += value; }); // sum == 15

// Now increment each array element

data.forEach(function(v, i, a) { a[i] = v + 1; }); // data == [2,3,4,5,6]

Note that forEach() does not provide a way to terminate iteration before all elements have been passed to the function. That is, there is no equivalent of the break statement you can use with a regular for loop.

*MAP()* -- The map() method passes each element of the array on which it is invoked to the function you specify and returns an array containing the values returned by your function. For example:

let a = [1, 2, 3]; a.map(x => x\*x) // => [1, 4, 9]:

the function takes input x and returns x\*x The function you pass to map() is invoked in the same way as a function passed to forEach(). For the map() method, however, the function you pass should return a value. Note that map() returns a new array: it does not modify the array it is invoked on. If that array is sparse, your function will not be called for the missing elements, but the returned array will be sparse in the same way as the original array: it will have the same length and the same missing elements.

*FILTER()* -- The filter() method returns an array containing a subset of the elements of the array on which it is invoked. The function you pass to it should be predicate: a function that returns true or false. The predicate is invoked just as for forEach() and map(). If the return value is true, or a value that converts to true, then the element passed to the predicate is a member of the subset and is added to the array that will become the return value. Examples:

let a = [5, 4, 3, 2, 1];

a.filter(x => x < 3) // => [2, 1]; values less than 3

a.filter((x,i) => i%2 === 0) // => [5, 3, 1]; every other value

Note that filter() skips missing elements in sparse arrays and that its return value is always dense. To close the gaps in a sparse array, you can do this: let dense = sparse.filter(() => true); And to close gaps and remove undefined and null elements, you can use filter, like this:

a = a.filter(x => x !== undefined && x !== null);

*FIND() AND FINDINDEX()* -- The find() and findIndex() methods are like filter() in that they iterate through your array looking for elements for which your predicate function returns a truthy value. Unlike filter(), however, these two methods stop iterating the first time the predicate finds an element. When that happens, find() returns the matching element, and findIndex() returns the index of the matching element. If no matching element is found, find() returns undefined and findIndex() returns -1:

let a = [1,2,3,4,5];

a.findIndex(x => x === 3) // => 2; the value 3 appears at index 2 a.findIndex(x => x < 0) // => -1; no negative numbers

a.find(x => x % 5 === 0) // => 5: this is a multiple of 5

a.find(x => x % 7 === 0) // => undefined: no multiples of 7

*EVERY() AND SOME()* -- they apply a predicate function you specify to the elements of the array, then return true or false. The every() method returns true if and only if your predicate function returns true for all elements in the array. The some() method returns true if there exists at least one element in the array for which the predicate returns true and returns false if and only if the predicate returns false for all elements of the array

Note that both every() and some() stop iterating array elements as soon as they know what value to return. some() returns true the first time your predicate returns true and only iterates through the entire array if your predicate always returns false. every() is the opposite: it returns false the first time your predicate returns false and only iterates all elements if your predicate always returns true. every() returns true and some() returns false when invoked on an empty array.

*REDUCE() AND REDUCERIGHT()* -- The reduce() and reduceRight() methods combine the elements of an array, using the function you specify, to produce a single value.

let a = [1,2,3,4,5];

a.reduce((x,y) => x+y, 0) // => 15; the sum of the values a.reduce((x,y) => x\*y, 1) // => 120; the product of the values a.reduce((x,y) => (x > y) ? x : y) // => 5; the largest of the values

reduce() takes two arguments. The first is the function that performs the reduction operation. The second (optional) argument is an initial value to pass to the function. The familiar value, index, and array values are passed as the second, third, and fourth arguments. The first argument is the accumulated result of the reduction so far. On the first call to the function, this first argument is the initial value you passed as the second argument to reduce(). On subsequent calls, it is the value returned by the previous invocation of the function.

When you invoke reduce() with no initial value(second (optional) argument), it uses the first element of the array as the initial value. This means that the first call to the reduction function will have the first and second array elements as its first and second arguments.

Calling reduce() on an empty array with no initial value argument causes a TypeError. If you call it with only one value—either an array with one element and no initial value or an empty array and an initial value—it simply returns that one value without ever calling the reduction function.

reduceRight() works just like reduce(), except that it processes the array from highest index to lowest (right-to-left), rather than from lowest to highest. You might want to do this if the reduction operation has right-to-left associativity, for example:

// Compute 2^(3^4). Exponentiation has right-to-left precedence

let a = [2, 3, 4]; a.reduceRight((acc,val) => Math.pow(val,acc)) // => 2.4178516392292583e+24

Note that neither reduce() nor reduceRight() accepts an optional argument that specifies the this value on which the reduction function is to be invoked. The optional initial value argument takes its place.

1. *Flattening arrays with flat() and flatMap()* – Introduced in ES2019,

[1, [2, 3]].flat() // => [1, 2, 3]

[1, [2, [3]]].flat() // => [1, 2, [3]]

When called with no arguments, flat() flattens one level of nesting.

let a = [1, [2, [3, [4]]]];

a.flat(1) // => [1, 2, [3, [4]]]

a.flat(2) // => [1, 2, 3, [4]]

a.flat(3) // => [1, 2, 3, 4]

a.flat(4) // => [1, 2, 3, 4]

*The flatMap() method* -- works just like the map() method, except that the returned array is automatically flattened as if passed to flat(). That is, calling a.flatMap(f) is the same as (but more efficient than) a.map(f).flat():

let phrases = ["hello world", "the definitive guide"];

let words = phrases.flatMap(phrase => phrase.split(" "));

words // => ["hello", "world", "the", "definitive", "guide"];

You can think of flatMap() as a generalization of map() that allows each element of the input array to map to any number of elements of the output array. In particular, flatMap() allows you to map input elements to an empty array, which flattens to nothing in the output array:

// Map non-negative numbers to their square roots

[-2, -1, 1, 2].flatMap(x => x < 0 ? [] : Math.sqrt(x)) // => [1, 2\*\*0.5]

1. *Adding arrays with concat()* –

let a = [1,2,3];

a.concat(4, 5) // => [1,2,3,4,5]

a.concat([4,5],[6,7]) // => [1,2,3,4,5,6,7]; arrays are flattened

a.concat(4, [5,[6,7]]) // => [1,2,3,4,5,[6,7]]; but not nested arrays

a // => [1,2,3]; the original array is unmodified

1. *Stacks and Queues with push(), pop(), shift(), and unshift()* –

The push() method appends one or more new elements to the end of an array and returns the new length of the array. Unlike concat(), push() does not flatten array arguments. The pop() method does the reverse: it deletes the last element of an array, decrements the array length, and returns the value that it removed. Note that both methods modify the array in place.

unshift() adds an element or elements to the beginning of the array, shifts the existing array elements up to higher indexes to make room, and returns the new length of the array. shift() removes and returns the first element of the array, shifting all subsequent elements down one place to occupy the newly vacant space at the start of the array.

let a = []; // a == []

a.unshift(1) // a == [1]

a.unshift(2) // a == [2, 1]

a = []; // a == []

a.unshift(1,2) // a == [1, 2]

1. *Subarrays with slice(), splice(), fill(), and copyWithin()* –

let a = [1,2,3,4,5];

a.slice(0,3); // Returns [1,2,3]

a.slice(3); // Returns [4,5]

a.slice(1,-1); // Returns [2,3,4]

a.slice(-3,-2); // Returns [3]

let a = [1,2,3,4,5,6,7,8];

a.splice(4) // => [5,6,7,8]; a is now [1,2,3,4]

a.splice(1,2) // => [2,3]; a is now [1,4]

a.splice(1,1) // => [4]; a is now [1]

The first two arguments to splice() specify which array elements are to be deleted. These arguments may be followed by any number of additional arguments that specify elements to be inserted into the array, starting at the position specified by the first argument. For example:

let a = [1,2,3,4,5];

a.splice(2,0,"a","b") // => []; a is now [1,2,"a","b",3,4,5] a.splice(2,2,[1,2],3) // => ["a","b"]; a is now [1,2, [1,2],3,3,4,5]

*FILL()* -- The fill() method sets the elements of an array, or a slice of an array, to a specified value. It mutates the array it is called on, and also returns the modified array:

let a = new Array(5); // Start with no elements and length 5

a.fill(0) // => [0,0,0,0,0]; fill the array with zeros

a.fill(9, 1) // => [0,9,9,9,9]; fill with 9 starting at index 1

a.fill(8, 2, -1) // => [0,9,8,8,9]; fill with 8 at indexes 2, 3

1. *INDEXOF() AND LASTINDEXOF()* -- indexOf() and lastIndexOf() search an array for an element with a specified value and return the index of the first such element found, or -1 if none is found. indexOf() searches the array from beginning to end, and lastIndexOf() searches from end to beginning:

let a = [0,1,2,1,0];

a.indexOf(1) // => 1: a[1] is 1

a.lastIndexOf(1) // => 3: a[3] is 1

a.indexOf(3) // => -1: no element has value 3

indexOf() and lastIndexOf() compare their argument to the array elements using the equivalent of the === operator. If your array contains objects instead of primitive values, these methods check to see if two references both refer to exactly the same object. If you want to actually look at the content of an object, try using the find() method with your own custom predicate function instead.

indexOf() and lastIndexOf() take an optional second argument that specifies the array index at which to begin the search. If this argument is omitted, indexOf() starts at the beginning and lastIndexOf() starts at the end. Negative values are allowed for the second argument and are treated as an offset from the end of the array, as they are for the slice() method: a value of –1, for example, specifies the last element of the array. The following function searches an array for a specified value and returns an array of all matching indexes. This demonstrates how the second argument to indexOf() can be used to find matches beyond the first.

function findall(a, x) {

let results = [], len = a.length, pos = 0;

while(pos < len) {

pos = a.indexOf(x, pos);

if (pos === -1) break; // If nothing found, we're done.

results.push(pos);

pos = pos + 1; }

return results; }

Note that strings have indexOf() and lastIndexOf() methods that work like these array methods, except that a negative second argument is treated as zero.

*The includes()* – this method is slightly different than the indexOf() method in one important way. indexOf() tests equality using the same algorithm that the === operator does, and that equality algorithm considers the not-a-number value to be different from every other value, including itself. includes() uses a slightly different version of equality that does consider NaN to be equal to itself. This means that indexOf() will not detect the NaN value in an array, but includes() will:

let a = [1,true,3,NaN];

a.includes(true) // => true

a.includes(2) // => false

a.includes(NaN) // => true

a.indexOf(NaN) // => -1; indexOf can't find NaN

*SORT()* -- sort() sorts the elements of an array in place and returns the sorted array. When sort() is called with no arguments, it sorts the array elements in alphabetical order (temporarily converting them to strings to perform the comparison, if necessary):

let a = ["banana", "cherry", "apple"];

a.sort(); // a == ["apple", "banana", "cherry"]

If an array contains undefined elements, they are sorted to the end of the array.

To sort an array into some order other than alphabetical, you must pass a comparison function as an argument to sort(). This function decides which of its two arguments should appear first in the sorted array. If the first argument should appear before the second, the comparison function should return a number less than zero. If the first argument should appear after the second in the sorted array, the function should return a number greater than zero. And if the two values are equivalent (i.e., if their order is irrelevant), the comparison function should return 0. So, for example, to sort array elements into numerical rather than alphabetical order, you might do this:

let a = [33, 4, 1111, 222];

a.sort(); // a == [1111, 222, 33, 4]; alphabetical order a.sort(function(a,b) { return a-b; });

// a == [4, 33, 222, 1111]; numerical order

a.sort((a,b) => b-a); // a == [1111, 222, 33, 4]; reverse

As another example of sorting array items, you might perform a case- insensitive alphabetical sort on an array of strings by passing a comparison function that converts both of its arguments to lowercase (with the toLowerCase() method) before comparing them:

let a = ["ant", "Bug", "cat", "Dog"];

a.sort(); // a == ["Bug","Dog","ant","cat"]; case sensitive

sort a.sort(function(s,t) {

let a = s.toLowerCase();

let b = t.toLowerCase();

if (a < b) return -1;

if (a > b) return 1;

return 0; });

// a == ["ant","Bug","cat","Dog"]; case-insensitive sort

*REVERSE()* -- The reverse() method reverses the order of the elements of an array and returns the reversed array. It does this in place; in other words, it doesn’t create a new array with the elements rearranged but instead rearranges them in the already existing array:

let a = [1,2,3]; a.reverse(); // a == [3,2,1]

1. *Functions* -- If a function is assigned to a property of an object, it is known as a method of that object. When a function is invoked on or through an object, that object is the invocation context or this value for the function. Functions designed to initialize a newly created object are called constructors.

One of the important things to understand about function declarations is that the name of the function becomes a variable whose value is the function itself. Function declaration statements are “hoisted” to the top of the enclosing script, function, or block so that functions defined in this way may be invoked from code that appears before the definition. Another way to say this is that all of the functions declared in a block of JavaScript code will be defined throughout that block, and they will be defined before the JavaScript interpreter begins to execute any of the code in that block.

1. *Function Expressions* -- Function expressions look a lot like function declarations, but they appear within the context of a larger expression or statement, and the name is optional. Here are some example function expressions:

// This function expression defines a function that squares its argument.

const square = function(x) { return x\*x; };

// Function expressions can include names, which is useful for recursion. const f = function fact(x) {

if (x <= 1) return 1; else return x\*fact(x-1); };

// Function expressions can also be used as arguments to other funct

[3,2,1].sort(function(a,b) { return a-b; });

//Function expressions are sometimes defined & immediately invoked:

let tensquared = (function(x) {return x\*x;}(10));

Note that the function name is optional for functions defined as expressions. A function declaration actually declares a variable and assigns a function object to it. A function expression, on the other hand, does not declare a variable: it is up to you to assign the newly defined function object to a constant or variable. It is a good practice to use const with function expressions so you don’t accidentally overwrite your functions by assigning new values.

If a function expression includes a name, the local function scope for that function will include a binding of that name to the function object. In effect, the function name becomes a local variable within the function.

There is an important difference between defining a function f() with a function declaration and assigning a function to the variable f after creating it as an expression. When you use the declaration form, the function objects are created before the code that contains them starts to run, and the definitions are hoisted so that you can call these functions from code that appears above the definition statement. This is not true for functions defined as expressions, however: these functions do not exist until the expression that defines them are actually evaluated. Furthermore, in order to invoke a function, you must be able to refer to it, and you can’t refer to a function defined as an expression until it is assigned to a variable, so functions defined with expressions cannot be invoked before they are defined.

1. *Arrow Functions* -- if an arrow function has exactly one parameter, you can omit the parentheses around the parameter list:

const polynomial = x => x\*x + 2\*x + 3;

Note, however, that an arrow function with no arguments at all must be written with an empty pair of parentheses:

const constantFunc = () => 42;

Note that, when writing an arrow function, you must not put a new line between the function parameters and the => arrow. Otherwise, you could end up with a line like const polynomial = x, which is a syntactically valid assignment statement on its own.

Also, if the body of your arrow function is a single return statement but the expression to be returned is an object literal, then you have to put the object literal inside parentheses to avoid syntactic ambiguity between the curly braces of a function body and the curly braces of an object literal:

const f = x => { return { value: x }; }; // Good: f() returns an object const g = x => ({ value: x }); // Good: g() returns an object

const h = x => { value: x }; // Bad: h() returns nothing

const i = x => { v: x, w: x }; // Bad: Syntax Error

Arrow functions differ from functions defined in other ways in one critical way: they inherit the value of the this keyword from the environment in which they are defined rather than defining their own invocation context as functions defined in other ways do. This is an important and very useful feature of arrow functions, and we’ll return to it again later in this chapter. Arrow functions also differ from other functions in that they do not have a prototype property, which means that they cannot be used as constructor functions for new classes.

1. *Invoking Functions* -- The JavaScript code that makes up the body of a function is not executed when the function is defined, but rather when it is invoked. JavaScript functions can be invoked in five ways:

* As functions
* As methods
* As constructors
* Indirectly through their call() and apply() methods
* Implicitly, via JavaScript language features that do not appear like normal function invocations.

In ES2020 you can insert ?. after the function expression and before the open parenthesis in a function invocation in order to invoke the function only if it is not null or undefined. That is, the expression f?.(x) is equivalent (assuming no side effects) to:

(f !== null && f !== undefined) ? f(x) : undefined

For function invocation in non-strict mode, the invocation context (the this value) is the global object. In strict mode, however, the invocation context is undefined. Note that functions defined using the arrow syntax behave differently: they always inherit the this value that is in effect where they are defined. Functions written to be invoked as functions (and not as methods) do not typically use the this keyword at all. The keyword can be used, however, to determine whether strict mode is in effect: // Define and invoke a function to determine if we're in strict mode. const strict = (function() { return !this; }());

*RECURSIVE FUNCTIONS AND THE STACK* -- A recursive function is one, like the factorial() function at the start of this chapter, that calls itself. Some algorithms, such as those involving tree-based data structures, can be implemented particularly elegantly with recursive functions. When writing a recursive function, however, it is important to think about memory constraints. When a function A calls function B, and then function B calls function C, the JavaScript interpreter needs to keep track of the execution contexts for all three functions. When function C completes, the interpreter needs to know where to resume executing function B, and when function B completes, it needs to know where to resume executing function A. You can imagine these execution contexts as a stack. When a function calls another function, a new execution context is pushed onto the stack. When that function returns, its execution context object is popped off the stack. If a function calls itself recursively 100 times, the stack will have 100 objects pushed onto it, and then have those 100 objects popped off. This call stack takes memory. On modern hardware, it is typically fine to write recursive functions that call themselves hundreds of times. But if a function calls itself ten thousand times, it is likely to fail with an error such as “Maximum call-stack size exceeded.”

When you write a method that does not have a return value of its own, consider having the method return this. If you do this consistently throughout your API, you will enable a style of programming known as method chaining in which an object can be named once and then multiple methods can be invoked on it:

new Square().x(100).y(100).size(50).outline("red").fill("blue").draw();

1. It is a common mistake to assume that a nested function defined within a method and invoked as a function can use this to obtain the invocation context of the method. The following code demonstrates the problem:

let o = {

o. m: function() {

let self = this;

this === o // => true:

f();

function f() {

this === o // => false: "this" is global or undefined

self === o // => true: self is the outer "this" value. }

} };

o.m();

Inside the nested function f(), the this keyword is not equal to the object o. This is widely considered to be a flaw in the JavaScript language, and it is important to be aware of it. The code above demonstrates one common workaround. Within the method m, we assign the this value to a variable self, and within the nested function f, we can use self instead of this to refer to the containing object.

In ES6 and later, another workaround to this issue is to convert the nested function f into an arrow function, which will properly inherit the this value:

const f = () => { this === o // true, since arrow functions inherit this };

Another workaround is to invoke the bind() method of the nested function to define a new function that is implicitly invoked on a specified object:

const f = (function() { this === o // true, since we bound this function to the outer this }).bind(this);

1. A constructor invocation creates a new, empty object that inherits from the object specified by the prototype property of the constructor. Constructor functions are intended to initialize objects, and this newly created object is used as the invocation context, so the constructor function can refer to it with the this keyword. Note that the new object is used as the invocation context even if the constructor invocation looks like a method invocation. That is, in the expression new o.m(), o is not used as the invocation context.

Constructor functions do not normally use the return keyword. They typically initialize the new object and then return implicitly when they reach the end of their body. In this case, the new object is the value of the constructor invocation expression. If, however, a constructor explicitly uses the return statement to return an object, then that object becomes the value of the invocation expression. If the constructor uses return with no value, or if it returns a primitive value, that return value is ignored and the new object is used as the value of the invocation.

1. In ES6 and later, you can define a default value for each of your function parameters directly in the parameter list of your function. Simply follow the parameter name with an equals sign and the default value to use when no argument is supplied for that parameter:

function getPropertyNames(o, a = []) {

for(let property in o) a.push(property); return a; }

Parameter default expressions are evaluated when your function is called, not when it is defined, It is probably easiest to reason about functions if the parameter defaults are constants (or literal expressions like [] and {}). But this is not required: you can use variables, or function invocations, for example, to compute the default value of a parameter. One interesting case is that, for functions with multiple parameters, you can use the value of a previous parameter to define the default value of the parameters that follow it:

const rectangle = (width, height=width\*2) => ({width, height}); rectangle(1) // => { width: 1, height: 2 }

1. *Rest Parameters* -- A rest parameter is preceded by three periods, and it must be the last parameter in a function declaration. When you invoke a function with a rest parameter, the arguments you pass are first assigned to the non-rest parameters, and then any remaining arguments (i.e., the “rest” of the arguments) are stored in an array that becomes the value of the rest parameter. This last point is important: within the body of a function, the value of a rest parameter will always be an array. The array may be empty, but a rest parameter will never be undefined.

Functions like the previous example that can accept any number of arguments are called variadic functions, variable arity functions, or vararg functions. This book uses the most colloquial term, varargs, which dates to the early days of the C programming language. Don’t confuse the ... that defines a rest parameter in a function definition with the ... spread operator, which can be used in function invocations.

using ... in a function definition gathers multiple function arguments into an array. Rest parameters and the spread operator are often useful together, as in the following function, which takes a function argument and returns an instrumented version of the function for testing:

function timed(f) {

return function(...args) {

console.log(`Entering function ${f.name}`);

let startTime = Date.now();

try { return f(...args); }

finally {

console.log(`Exiting${f.name}after ${Date.now()-

startTime}ms`); }

}; }

function benchmark(n) { let sum = 0; for(let i = 1; i <= n; i++) sum += i; return sum; }

timed(benchmark)(1000000) // => 500000500000;

1. When you destructure an array, you can define a rest parameter for extra values within the array that is being unpacked. That rest parameter within the square brackets is completely different than the true rest parameter for the function:

function f([x, y, ...coords], ...rest) {

return [x+y, ...rest, ...coords]; // Note: spread operator here }

f([1, 2, 3, 4], 5, 6) // => [3, 5, 6, 3, 4]

In ES2018, you can also use a rest parameter when you destructure an object. The value of that rest parameter will be an object that has any properties that did not get destructured. Object rest parameters are often useful with the object spread operator, which is also a new feature of ES2018:

function vectorMultiply({x, y, z=0, ...props}, scalar) {

return { x: x\*scalar, y: y\*scalar, z: z\*scalar, ...props }; } vectorMultiply({x: 1, y: 2, w: -1}, 2) // => {x: 2, y: 4, z: 0, w: -1}

function drawCircle({ x, y, radius, color: [r, g, b] }) {

console.log({ x, y, radius, r, g, b }); }

drawCircle({ x: "x", y: "y", radius: 5, color: ["r", "g", "b"] });

1. Functions don’t even require names at all, as when they’re assigned to array elements:

let a = [ x => x\*x, 20]; // An array literal

a[0](a[1]) // => 400

1. *Defining Your Own Function Properties* -- Functions are not primitive values in JavaScript, but a specialized kind of object, which means that functions can have properties. When a function needs a “static” variable whose value persists across invocations, it is often convenient to use a property of the function itself.

uniqueInteger.counter = 0;

function uniqueInteger() { return uniqueInteger.counter++;} uniqueInteger() // => 0

uniqueInteger() // => 1

As another example, consider the following factorial() function that uses properties of itself (treating itself as an array) to cache previously computed results:

factorial[1] = 1;

function factorial(n) {

if (Number.isInteger(n) && n > 0) {

if (!(n in factorial)) factorial[n] = n \* factorial(n - 1);

return factorial[n]; }

else return NaN; }

console.log(factorial(100)); //=> 720

console.log(factorial); //=> [Function: factorial] { '1': 1, '2': 2, '3': 6, '4': 24, '5': 120, '6': 720 }

*Functions as Namespaces* -- Variables declared within a function are not visible outside of the function. For this reason, it is sometimes useful to define a function simply to act as a temporary namespace in which you can define variables without cluttering the global namespace.

function chunkNamespace() {

// Chunk of code goes here // Any variables defined in the chunk are local to this function // instead of cluttering up the global namespace. } chunkNamespace();

If defining even a single property is too much, you can define and invoke an anonymous function in a single expression:

(function() { // chunkNamespace() function rewritten as an unnamed expression. // Chunk of code goes here }());

This technique of defining and invoking a function in a single expression is called “immediately invoked function expression.” The open parenthesis before function is required because without it, the JavaScript interpreter tries to parse the function keyword as a function declaration statement. With the parenthesis, the interpreter correctly recognizes this as a function definition expression.

1. Closures -- Like most modern programming languages, JavaScript uses lexical scoping. This means that functions are executed using the variable scope that was in effect when they were defined, not the variable scope that is in effect when they are invoked. In order to implement lexical scoping, the internal state of a JavaScript function object must include not only the code of the function but also a reference to the scope in which the function definition appears. This combination of a function object and a scope (a set of variable bindings) in which the function’s variables are resolved is called a closure in the computer science literature.

Closures become interesting when they are invoked from a different scope than the one they were defined in.

let scope = "global scope";

function checkscope() {

let scope = "local scope";

function f() { return scope; }

return f(); }

checkscope() //=> local scope

let scope = "global scope";

function checkscope() {

let scope = "local scope";

function f() {return scope;}

return f; }

checkscope()() // => local scope

JavaScript functions are executed using the scope they were defined in. The nested function f() was defined in a scope where the variable scope was bound to the value “local scope”. That binding is still in effect when f is executed, no matter where it is executed from. This, in a nutshell, is the surprising and powerful nature of closures: they capture the local variable (and parameter) bindings of the outer function within which they are defined.

let uniqueInteger = (function() {

let counter = 0;

return function() { return counter++; }; }()); uniqueInteger() // => 0

uniqueInteger() // => 1

The nested function has access to the variables in its scope and can use the counter variable defined in the outer function. Once that outer function returns, no other code can see the counter variable: the inner function has exclusive access to it. Private variables like counter need not be exclusive to a single closure: it is perfectly possible for two or more nested functions to be defined within the same outer function and share the same scope. Consider the following code:

function counter() {

let n = 0;

return {

count: function() { return n++; },

reset: function() { n = 0; } }; }

let c = counter(), d = counter(); // Create two counters

c.count() // => 0

d.count() // => 0: they count independently

c.reset(); // reset() and count() methods share state

c.count() // => 0: because we reset c

d.count() // => 1: d was not reset

It is worth noting here that you can combine this closure technique with property getters and setters.

function counter(n) {

return {

get count() { return n++; },

set count(m) {

if (m > n) n = m;

else throw Error("count can only be set to a larger value"); } }; }

let c = counter(1000);

c.count // => 1000

c.count // => 1001

c.count = 2000;

c.count // => 2000

c.count = 2000; // !Error: count can only be set to a larger value

function addPrivateProperty(o, name, predicate) {

let value;

o[`get${name}`] = function () { return value; };

o[`set${name}`] = function (v) {

if (predicate && !predicate(v))

throw new TypeError(`set${name}: invalid value ${v}`);

else value = v; }; }

let o = {};

addPrivateProperty(o, "Name", (x) => typeof x === "string"); o.setName("Frank");

o.getName(); // => "Frank"

o.setName(0); //=> error

We’ve now seen a number of examples in which two closures are defined in the same scope and share access to the same private variable or variables. This is an important technique, but it is just as important to recognize when closures inadvertently share access to a variable that they should not share. Consider the following code:

function constfunc(v) { return () => v; }

let funcs = [];

for(var i = 0; i < 10; i++) funcs[i] = constfunc(i);

funcs[5]() // => 5

When working with code like this that creates multiple closures using a loop, it is a common error to try to move the loop within the function that defines the closures. Think about the following code, for example:

function constfuncs() {

let funcs = [];

for(var i = 0; i < 10; i++) { funcs[i] = () => i; }

console.log(i) // => 10

return funcs; }

let funcs = constfuncs();

funcs[5]() // => 10;

Why doesn't this return 5? This code creates 10 closures and stores them in an array. The closures are all defined within the same invocation of the function, so they share access to the variable i. When constfuncs() returns, the value of the variable i is 10, and all 10 closures share this value. Therefore, all the functions in the returned array of functions return the same value, which is not what we wanted at all. It is important to remember that the scope associated with a closure is “live.” Nested functions do not make private copies of the scope or make static snapshots of the variable bindings. Fundamentally, the problem here is that variables declared with var are defined throughout the function. Our for loop declares the loop variable with var i, so the variable i is defined throughout the function rather than being more narrowly scoped to the body of the loop. The code demonstrates a common category of bugs in ES5 and before, but the introduction of block-scoped variables in ES6 addresses the issue. If we just replace the var with a let or a const, then the problem goes away. Because let and const are block scoped, each iteration of the loop defines a scope that is independent of the scopes for all other iterations, and each of these scopes has its own independent binding of i.

Another thing to remember when writing closures is that this is a JavaScript keyword, not a variable. As discussed earlier, arrow functions inherit the this value of the function that contains them, but functions defined with the function keyword do not. So if you’re writing a closure that needs to use the this value of its containing function, you should use an arrow function, or call bind(), on the closure before returning it, or assign the outer this value to a variable that your closure will inherit: const self = this; // Make the this value available to nested functions

1. *Function Properties, Methods, and Constructor* -- We’ve seen that functions are values in JavaScript programs. The typeof operator returns the string “function” when applied to a function, but functions are really a specialized kind of JavaScript object. Since functions are objects, they can have properties and methods, just like any other object. There is even a Function() constructor to create new function objects.

*The length Property* -- The read-only length property of a function specifies the arity of the function—the number of parameters it declares in its parameter list, which is usually the number of arguments that the function expects. If a function has a rest parameter, that parameter is not counted for the purposes of this length property.

*The name Property* -- The read-only name property of a function specifies the name that was used when the function was defined, if it was defined with a name, or the name of the variable or property that an unnamed function expression was assigned to when it was first created. This property is primarily useful when writing debugging or error messages.

*The prototype Property* -- All functions, except arrow functions, have a prototype property that refers to an object known as the prototype object. Every function has a different prototype object. When a function is used as a constructor, the newly created object inherits properties from the prototype object.

*The call() and apply() Methods* -- call() and apply() allow you to indirectly invoke a function as if it were a method of some other object. The first argument to both call() and apply() is the object on which the function is to be invoked; this argument is the invocation context and becomes the value of the this keyword within the body of the function.

To invoke the function f() as a method of the object o (passing no arguments), you could use either call() or apply(): f.call(o); f.apply(o);

Either of these lines of code are similar to the following (which assume that o does not already have a property named m):

o.m = f; // Make f a temporary method of o.

o.m(); // Invoke it, passing no arguments.

delete o.m; // Remove the temporary method.

Remember that arrow functions inherit the this value of the context where they are defined. This cannot be overridden with the call() and apply() methods. If you call either of those methods on an arrow function, the first argument is effectively ignored. Any arguments to call() after the first invocation context argument are the values that are passed to the function that is invoked (and these arguments are not ignored for arrow functions). For example, to pass two numbers to the function f() and invoke it as if it were a method of the object o, you could use code like this: f.call(o, 1, 2);

The apply() method is like the call() method, except that the arguments to be passed to the function are specified as an array: f.apply(o, [1,2]);

The trace() function defined in the following is similar to the timed() function, but it works for methods instead of functions. It uses the apply() method instead of a spread operator, and by doing that, it is able to invoke the wrapped method with the same arguments and the same this value as the wrapper method:

function trace(o, m) {

let original = o[m];

o[m] = function (...args) {

console.log(new Date(), "Entering:", m);

let result = original.apply(this, args);

console.log(new Date(), "Exiting:", m);

return result; }; }

*The bind() Method* -- The primary purpose of bind() is to bind a function to an object. When you invoke the bind() method on a function f and pass an object o, the method returns a new function. Invoking the new function (as a function) invokes the original function f as a method of o. Any arguments you pass to the new function are passed to the original function. For example:

function f(y) { return this.x + y; }

let o = { x: 1 };

let g = f.bind(o); // Calling g(x) invokes f() on o

g(2) // => 3

let p = { x: 10, g }; // Invoke g() as a method of this object

p.g(2) // => 3: g is still bound to o, not p.

Arrow functions inherit their this value from the environment in which they are defined, and that value cannot be overridden with bind(), so if the function f() in the preceding code was defined as an arrow function, the binding would not work. The most common use case for calling bind() is to make non-arrow functions behave like arrow functions, however, so this limitation on binding arrow functions is not a problem in practice.

The bind() method does more than just bind a function to an object, however. It can also perform partial application: any arguments you pass to bind() after the first are bound along with the this value. This partial application feature of bind() does work with arrow functions. Partial application is a common technique in functional programming and is sometimes called currying. Here are some examples of the bind() method used for partial application:

let sum = (x,y) => x + y;

let succ = sum.bind(null, 1); // Bind the first argument to 1

succ(2) // => 3: x is bound to 1, and we pass 2 for the y argument function f(y,z) { return this.x + y + z; }

let g = f.bind({x: 1}, 2); // Bind this and y

g(3) // => 6: this.x is bound to 1, y is bound to 2 and z is 3

The name property of the function returned by bind() is the name property of the function that bind() was called on, prefixed with the word “bound”

*The toString() Method* -- Like all JavaScript objects, functions have a toString() method. The ECMAScript spec requires this method to return a string that follows the syntax of the function declaration statement. In practice, most (but not all) implementations of this toString() method return the complete source code for the function. Built-in functions typically return a string that includes something like “[native code]” as the function body.

*The Function() Constructor* -- Because functions are objects, there is a Function() constructor that can be used to create new functions:

const f = new Function("x", "y", "return x\*y;");

This line of code creates a new function that is more or less equivalent to a function defined with the familiar syntax:

const f = function(x, y) { return x\*y; };

The Function() constructor expects any number of string arguments. The last argument is the text of the function body; it can contain arbitrary JavaScript statements, separated from each other by semicolons. All other arguments to the constructor are strings that specify the parameter names for the function. If you are defining a function that takes no arguments, you would simply pass a single string —the function body—to the constructor.

Notice that the Function() constructor is not passed any argument that specifies a name for the function it creates. Like function literals, the Function() constructor creates anonymous functions. There are a few points that are important to understand about the Function() constructor:

* The Function() constructor allows JavaScript functions to be dynamically created and compiled at runtime.
* The Function() constructor parses the function body and creates a new function object each time it is called. If the call to the constructor appears within a loop or within a frequently called function, this process can be inefficient.
* A last, very important point about the Function() constructor is that the functions it creates do not use lexical scoping; instead, they are always compiled as if they were toplevel functions, as the following code demonstrates:

let scope = "global";

function constructFunction() {

let scope = "local"; return new Function("return scope"); } constructFunction()() // => "global"

1. *Processing Arrays with Functions* -- Suppose we have an array of numbers and we want to compute the mean and standard deviation of those values. We might do that in nonfunctional style like this:

let data = [1,1,3,5,5]; // The mean is the sum of the elements divided by the number of elements

let total = 0;

for(let i = 0; i < data.length; i++) total += data[i];

let mean = total/data.length; // mean == 3;

// To compute the standard deviation, we first sum the squares of // the deviation of each element from the mean.

total = 0;

for(let i = 0; i < data.length; i++) {

let deviation = data[i] - mean; total += deviation \* deviation; } let stddev = Math.sqrt(total/(data.length-1)); // stddev == 2

We can perform these same computations in concise functional style using the array methods map() and reduce()

// First, define two simple functions

const sum = (x,y) => x+y;

const square = x => x\*x;

let data = [1,1,3,5,5];

let mean = data.reduce(sum)/data.length; // mean == 3

let deviations = data.map(x => x-mean);

let stddev = Math.sqrt(deviations.map(square).reduce(sum)/(data.length1)); stddev // => 2

This new version of the code looks quite different than the first one, but it is still invoking methods on objects, so it has some objectoriented conventions remaining. Let’s write functional versions of the map() and reduce() methods:

const map = function(a, ...args) { return a.map(...args); };

const reduce = function(a, ...args) { return a.reduce(...args); };

With these map() and reduce() functions defined, our code to compute the mean and standard deviation now looks like this:

const sum = (x,y) => x+y; const square = x => x\*x;

let data = [1,1,3,5,5];

let mean = reduce(data, sum)/data.length;

let deviations = map(data, x => x-mean);

let stddev = Math.sqrt(reduce(map(deviations, square), sum)/(data.length-1)); stddev // => 2

1. *Higher-Order Functions* -- A higher-order function is a function that operates on functions, taking one or more functions as arguments and returning a new function. Here is an example:

function not(f) {

return function(...args) {

let result = f.apply(this, args }; }

const even = x => x % 2 === 0;

const odd = not(even);

[1,1,3,5,5].every(odd) // => true: every element of the array is odd

Here is another, more general, example that takes two functions, f and g, and returns a new function that computes f(g()):

function compose(f, g) {

return function(...args) {

return f.call(this, g.apply(this, args)); }; }

const sum = (x,y) => x+y;

const square = x => x\*x;

compose(square, sum)(2,3) // => 25; the square of the sum

1. *Partial Application of Functions* -- The bind() method of a function f returns a new function that invokes f in a specified context and with a specified set of arguments. We say that it binds the function to an object and partially applies the arguments. The bind() method partially applies arguments on the left—that is, the arguments you pass to bind() are placed at the start of the argument list that is passed to the original function. But it is also possible to partially apply arguments on the right:

function partialLeft(f, ...outerArgs) {

return function(...innerArgs) {

let args = [...outerArgs, ...innerArgs];

return f.apply(this, args); }; }

function partialRight(f, ...outerArgs) {

return function(...innerArgs) {

let args = [...innerArgs, ...outerArgs];

return f.apply(this, args); }; }

function partial(f, ...outerArgs) {

return function(...innerArgs) {

let args = [...outerArgs];

let innerIndex=0;

for(let i = 0; i < args.length; i++)

if (args[i] === undefined) args[i] = innerArgs[innerIndex++];

args.push(...innerArgs.slice(innerIndex));

return f.apply(this, args); }; }

const f = function(x,y,z) { return x \* (y - z); };

partialLeft(f, 2)(3,4) // => -2: Bind first argument: 2 \* (3 - 4) partialRight(f, 2)(3,4) // => 6: Bind last argument: 3 \* (4 - 2)

partial(f, undefined, 2)(3,4) // => -6: Bind middle argument:3 \* (2 - 4)

These partial application functions allow us to easily define interesting functions out of functions we already have defined. Here are some examples:

const increment = partialLeft(sum, 1);

const cuberoot = partialRight(Math.pow, 1/3);

cuberoot(increment(26)) // => 3

1. *Memoization* -- function that caches its previously computed results. In functional programming, this kind of caching is called memoization. The code that follows shows a higher-order function, memoize(), that accepts a function as its argument and returns a memoized version of the function:

function memoize(f) {

const cache = new Map();

return function(...args) {

let key = args.length + args.join("+");

if (cache.has(key)) { return cache.get(key); }

else { let result = f.apply(this, args);

cache.set(key, result); return result; } }; }

The memoize() function creates a new object to use as the cache and assigns this object to a local variable so that it is private to (in the closure of) the returned function. The returned function converts its arguments array to a string and uses that string as a property name for the cache object. If a value exists in the cache, it returns it directly. Otherwise, it calls the specified function to compute the value for these arguments, caches that value, and returns it. Here is how we might use memoize():

Note that when we write a recursive function that we will be memoizing, // we typically want to recurse to the memoized version, not the original.

const factorial = memoize(function(n) {

return (n <= 1) ? 1 : n \* factorial(n-1); }); factorial(5) // => 120: also caches values for 4, 3, 2 and 1.

1. *Classes* -- In JavaScript, classes use prototype-based inheritance: if two objects inherit properties (generally function-valued properties, or methods) from the same prototype, then we say that those objects are instances of the same class. That, in a nutshell, is how JavaScript classes work.

JavaScript has always allowed the definition of classes. ES6 introduced a brand-new syntax (including a class keyword) that makes it even easier to create classes.

If you’re familiar with strongly typed object-oriented programming languages like Java or C++, you’ll notice that JavaScript classes are quite different from classes in those languages. There are some syntactic similarities, and you can emulate many features of “classical” classes in JavaScript, but it is best to understand up front that JavaScript’s classes and prototype-based inheritance mechanism are substantially different from the classes and class-based inheritance mechanism of Java and similar languages.

Example 9-1:A simple JavaScript class –

function range(from, to) {

let r = Object.create(range.methods);

r.from = from; r.to = to;

return r; }

range.methods = {

includes(x) { return this.from <= x && x <= this.to; },

\*[Symbol.iterator]() { for(let x = Math.ceil(this.from); x <=

this.to; x++) yield x; },

toString() { return "(" + this.from + "..." + this.to + ")"; } };

let r = range(1,3);

r.includes(2) // => true: 2 is in the range

r.toString() // => "(1...3)"

[...r] // => [1, 2, 3]; convert to an array via iterator

1. *Classes and Constructors* -- Example 9-1 demonstrates a simple way to define a JavaScript class. It is not the idiomatic way to do so, however, because it did not define a constructor. A constructor is a function designed for the initialization of newly created objects. Constructor invocations using new automatically create the new object, so the constructor itself only needs to initialize the state of that new object. The critical feature of constructor invocations is that the prototype property of the constructor is used as the prototype of the new object. Almost all objects have a prototype, only a few objects have a prototype property. Finally, we can clarify this: it is function objects that have a prototype property. This means that all objects created with the same constructor function inherit from the same object and are therefore members of the same class. Example 9-2 shows how we could alter the Range class of Example 9-1 to use a constructor function instead of a factory function.

Example 9-2: A Range class using a constructor –

function Range(from, to) {

this.from = from; this.to = to; }

Range.prototype = {

includes: function(x) { return this.from <= x && x <= this.to; },

[Symbol.iterator]: function\*() { for(let x = Math.ceil(this.from); x

<= this.to; x++) yield x; },

toString: function() { return "(" + this.from + "..." + this.to + ")"; }

let r = new Range(1,3)

r.includes(2) // => true: 2 is in the range

r.toString() // => "(1...3)"

[...r] // => [1, 2, 3];

constructor functions define, in a sense, classes, and classes have names that (by convention) begin with capital letters. Next, notice that the Range() constructor is invoked (at the end of the example) with the new keyword while the range() factory function was invoked without it. Example 9-1 uses regular function invocation to create the new object, and Example 9-2 uses constructor invocation. Because the Range() constructor is invoked with new, it does not have to call Object.create() or take any action to create a new object. The new object is automatically created before the constructor is called, and it is accessible as the this value. The Range() constructor merely has to initialize this. Constructors do not even have to return the newly created object. Constructor invocation automatically creates a new object, invokes the constructor as a method of that object, and returns the new object. The fact that constructor invocation is so different from regular function invocation is another reason that we give constructors names that start with capital letters. Constructors are written to be invoked as constructors, with the new keyword, and they usually won’t work properly if they are invoked as regular functions.

1. *Constructors, Class Identity, and instanceof* -- As we’ve seen, the prototype object is fundamental to the identity of a class: two objects are instances of the same class if and only if they inherit from the same prototype object. The constructor function that initializes the state of a new object is not fundamental: two constructor functions may have prototype properties that point to the same prototype object. Then, both constructors can be used to create instances of the same class. Even though constructors are not as fundamental as prototypes, the constructor serves as the public face of a class. Most obviously, the name of the constructor function is usually adopted as the name of the class. We say, for example, that the Range() constructor creates Range objects.

More fundamentally, however, constructors are used as the righthand operand of the instanceof operator when testing objects for membership in a class. If we have an object r and want to know if it is a Range object, we can write: r instanceof Range // => true: r inherits from Range.prototype. The lefthand operand should be the object that is being tested, and the righthand operand should be a constructor function that names a class. The expression o instanceof C evaluates to true if o inherits from C.prototype. The inheritance need not be direct: if o inherits from an object that inherits from an object that inherits from C.prototype, the expression will still evaluate to true.

Technically speaking, in the previous code example, the instanceof operator is not checking whether r was actually initialized by the Range constructor. Instead, it is checking whether r inherits from Range.prototype. If we define a function Strange() and set its prototype to be the same as Range.prototype, then objects created with new Strange() will count as Range objects as far as instanceof is concerned

function Strange() {}

Strange.prototype = Range.prototype;

new Strange() instanceof Range // => true

Even though instanceof cannot actually verify the use of a constructor, it still uses a constructor function as its righthand side because constructors are the public identity of a class. If you want to test the prototype chain of an object for a specific prototype and do not want to use the constructor function as an intermediary, you can use the isPrototypeOf() method. In Example 9-1, for example, we defined a class without a constructor function, so there is no way to use instanceof with that class. Instead, however, we could test whether an object r was a member of that constructor-less class with this code: range.methods.isPrototypeOf(r); // range.methods is the prototype object.

1. *The constructor Property* -- In Example 9-2, we set Range.prototype to a new object that contained the methods for our class. Although it was convenient to express those methods as properties of a single object literal, it was not actually necessary to create a new object. Any regular JavaScript function (excluding arrow functions, generator functions, and async functions) can be used as a constructor, and constructor invocations need a prototype property. Therefore, every regular JavaScript function automatically has a prototype property. The value of this property is an object that has a single, non-enumerable constructor property. The value of the constructor property is the function object:

let F = function() {}; // This is a function object.

F.prototype.constructor === F //true

The existence of this predefined prototype object with its constructor property means that objects typically inherit a constructor property that refers to their constructor:

let o = new F(); // Create an object o of class F

o.constructor === F // => true:

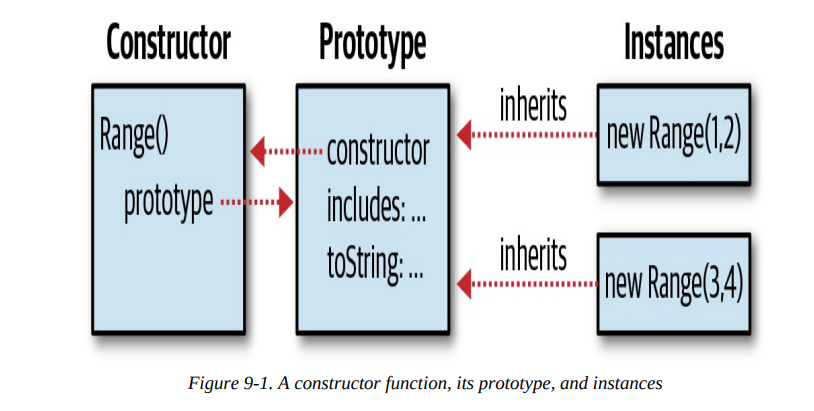


Figure 9-1 illustrates this relationship between the constructor function, its prototype object, the back reference from the prototype to the constructor, and the instances created with the constructor. Notice that Figure 9-1 uses our Range() constructor as an example. In fact, however, the Range class defined in Example 9-2 overwrites the predefined Range.prototype object with an object of its own. And the new prototype object it defines does not have a constructor property. So instances of the Range class, as defined, do not have a constructor property. We can remedy this problem by explicitly adding a constructor to the prototype:

Range.prototype = {

constructor: Range, // Explicitly set the constructor back-reference /\* method definitions go here \*/ };

Another common technique that you are likely to see in older JavaScript code is to use the predefined prototype object with its constructor property and add methods to it one at a time.

1. *Classes with the class Keyword* -- Classes have been part of JavaScript since the very first version of the language, but in ES6, they finally got their own syntax with the introduction of the class keyword.

class Range { constructor(from, to) {

this.from = from; this.to = to;

includes(x) { return this.from <= x && x <= this.to; }

\*[Symbol.iterator]() { for(let x = Math.ceil(this.from); x <= this.to; x++) yield x; }

toString() { return `(${this.from}...${this.to})`; } }

let r = new Range(1,3); // Create a Range object

r.includes(2) // => true: 2 is in the range

r.toString() // => "(1...3)"

[...r] // => [1, 2, 3]; convert to an array via iterator

The introduction of the class keyword to the language does not alter the fundamental nature of JavaScript’s prototype-based classes. The new class syntax is clean and convenient but is best thought of as “syntactic sugar” for the more fundamental class definition mechanism shown in Example 9-2.

The class body includes method definitions that use object literal method shorthand where the function keyword is omitted. Unlike object literals, however, no commas are used to separate the methods from each other. (Although class bodies are superficially similar to object literals, they are not the same thing. In particular, they do not support the definition of properties with name/value pairs.)

The keyword constructor is used to define the constructor function for the class. The function defined is not actually named “constructor”, however. The class declaration statement defines a new variable Range and assigns the value of this special constructor function to that variable. If your class does not need to do any initialization, you can omit the constructor keyword and its body, and an empty constructor function will be implicitly created for you.

1. Like function declarations, class declarations have both statement and expression forms. Just as we can write:

let square = function(x) { return x \* x; }; square(3) // => 9

we can also write:

let Square = class { constructor(x) { this.area = x \* x; } };

new Square(3).area // => 9

As with function definition expressions, class definition expressions can include an optional class name. If you provide such a name, that name is only defined within the class body itself.

1. *Static Methods* -- You can define a static method within a class body by prefixing the method declaration with the static keyword. Static methods are defined as properties of the constructor function rather than properties of the prototype object. For example, suppose we added the following code to Example 9-3:

static parse(s) {

let matches = s.match(/^\((\d+)\.\.\.(\d+)\)$/);

if (!matches) { throw new TypeError(`Cannot parse Range from "${s}".`) }

return new Range(parseInt(matches[1]), parseInt(matches[2])); }

The method defined by this code is Range.parse(), not Range.prototype.parse(), and you must invoke it through the constructor, not through an instance:

let r = Range.parse('(1...10)'); // Returns a new Range object r.parse('(1...10)'); // TypeError: r.parse is not a function

You’ll sometimes see static methods called class methods because they are invoked using the name of the class/constructor. When this term is used, it is to contrast class methods with the regular instance methods that are invoked on instances of the class. Because static methods are invoked on the constructor rather than on any particular instance, it almost never makes sense to use the this keyword in a static method.

1. Public, Private, and Static Fields -- If you want to define a field (which is just an object-oriented synonym for “property”) on a class instance, you must do that in the constructor function or in one of the methods. And if you want to define a static field for a class, you must do that outside the class body, after the class has been defined. Example 9-4 includes examples of both kinds of fields. Standardization is underway, however, for extended class syntax that allows the definition of instance and static fields, in both public and private forms.

Suppose you’re writing a class like this one, with a constructor that initializes three fields:

class Buffer {

constructor() {

this.size = 0;

this.capacity = 4096;

this.buffer = new Uint8Array(this.capacity); } }

With the new instance field syntax that is likely to be standardized, you could instead write:

class Buffer {

size = 0;

capacity = 4096;

buffer = new Uint8Array(this.capacity); }

The field initialization code has moved out of the constructor and now appears directly in the class body. (That code is still run as part of the constructor, of course. If you do not define a constructor, the fields are initialized as part of the implicitly created constructor.) The this. prefixes that appeared on the lefthand side of the assignments are gone, but note that you still must use this. to refer to these fields, even on the righthand side of the initializer assignments.

The same proposal that seeks to standardize these instance fields also defines private instance fields. If you use the instance field initialization syntax shown in the previous example to define a field whose name begins with # (which is not normally a legal character in JavaScript identifiers), that field will be usable (with the # prefix) within the class body but will be invisible and inaccessible (and therefore immutable) to any code outside of the class body. If, for the preceding hypothetical Buffer class, you wanted to ensure that users of the class could not inadvertently modify the size field of an instance, you could use a private #size field instead, then define a getter function to provide read-only access to the value:

class Buffer { #size = 0; get size() { return this.#size; } }

Note that private fields must be declared using this new field syntax before they can be used. You can’t just write this.#size = 0; in the constructor of a class unless you include a “declaration” of the field directly in the class body.

Finally, a related proposal seeks to standardize the use of the static keyword for fields. If you add static before a public or private field declaration, those fields will be created as properties of the constructor function instead of properties of instances. Consider the static Range.parse() method we’ve defined. It included a fairly complex regular expression that might be good to factor out into its own static field. With the proposed new static field syntax, we could do that like this:

static integerRangePattern = /^\((\d+)\.\.\.(\d+)\)$/;

static parse(s) { let matches = s.match(Range.integerRangePattern);

if (!matches) { throw new TypeError(`Cannot parse Range

from "${s}".`) }

return new Range(parseInt(matches[1]), matches[2]); }

If we wanted this static field to be accessible only within the class, we could make it private using a name like #pattern.

1. Example 9-4. Complex.js: a complex number class

class Complex {

#r = 0; #i = 0;

constructor(real, imaginary) {

this.#r = real;

this.#i = imaginary }

plus(that) {

return new Complex(this.#r + that.#r, this.#i + that.#i); }

times(that) {

return new Complex(this.#r \* that.#r - this.#i \* that.#i,

this.#r \* that.#i + this.#i \* that.#r); }

static sum(c, d) { return c.plus(d); }

static product(c, d) { return c.times(d); }

get real() { return this.#r; }

get imaginary() { return this.#i; }

get magnitude() { return Math.hypot(this.#r, this.#i); }

toString() { return `{${this.#r},${this.#i}}`; }

equals(that) {

return that instanceof Complex && this.#r === that.#r && this.#i === that.#i; }

static ZERO = new Complex(0,0);

static ONE = new Complex(1,0);

static I = new Complex(0,1) }

With the Complex class of Example 9-4 defined, we can use the constructor, instance fields, instance methods, class fields, and class methods with code like this:

let c = new Complex(2, 3); // Create a new object with the constructor let d = new Complex(c.imaginary, c.real); // Use instance fields of c c.plus(d).toString() // => "{5,5}"; use instance methods

c.magnitude // => Math.hypot(2,3); use a getter function Complex.product(c, d) // => new Complex(0, 13); a static method Complex.ZERO.toString() // => "{0,0}"; a static property

1. *Adding Methods to Existing Classes* -- JavaScript’s prototype-based inheritance mechanism is dynamic: an object inherits properties from its prototype, even if the properties of the prototype change after the object is created. This means that we can augment JavaScript classes simply by adding new methods to their prototype objects. Here, for example, is code that adds a method for computing the complex conjugate to the Complex class of Example 9-4:

Complex.prototype.conj = function() {

return new Complex(this.#r, -this.#i); };

The prototype object of built-in JavaScript classes is also open like this, which means that we can add methods to numbers, strings, arrays, functions, and so on. This is useful for implementing new language features in older versions of the language:

if (!String.prototype.startsWith) {

String.prototype.startsWith = function(s) {

return this.indexOf(s) === 0; }; }

Here is another example: // Invoke the function f this many times, passing the iteration number // For example, to print "hello" 3 times: // let n = 3; // n.times(i => { console.log(`hello ${i}`); });

Number.prototype.times = function(f, context) {

let n = this.valueOf();

for(let i = 0; i < n; i++) f.call(context, i); };

Adding methods to the prototypes of built-in types like this is generally considered to be a bad idea because it will cause confusion and compatibility problems in the future if a new version of JavaScript defines a method with the same name. It is even possible to add methods to Object.prototype, making them available for all objects. But this is never a good thing to do because properties added to Object.prototype are visible to for/in loops (though you can avoid this by using Object.defineProperty() to make the new property non-enumerable).

1. *Subclasses* -- In object-oriented programming, a class B can extend or subclass another class A. We say that A is the superclass and B is the subclass. Instances of B inherit the methods of A. The class B can define its own methods, some of which may override methods of the same name defined by class A. If a method of B overrides a method of A, the overriding method in B often needs to invoke the overridden method in A. Similarly, the subclass constructor B() must typically invoke the superclass constructor A() in order to ensure that instances are completely initialized.

*Subclasses and Prototypes* -- Suppose we wanted to define a Span subclass of the Range class from Example 9-2. This subclass will work just like a Range, but instead of initializing it with a start and an end, we’ll instead specify a start and a distance, or span. An instance of this Span class is also an instance of the Range superclass. A span instance inherits a customized toString() method from Span.prototype, but in order to be a subclass of Range, it must also inherit methods (such as includes()) from Range.prototype.

Example 9-5. Span.js: a simple subclass of Range

function Span(start, span) {

if (span >= 0) { this.from = start; this.to = start + span; }

else { this.to = start; this.from = start + span; } }

Span.prototype = Object.create(Range.prototype);

Span.prototype.constructor = Span;

Span.prototype.toString = function() { return `(${this.from}... +${this.to - this.from})`; You may notice that our Span() constructor sets the same from and to properties that the Range() constructor does and so does not need to invoke the Range() constructor to initialize the new object. Similarly, Span’s toString() method completely re-implements the string conversion without needing to call Range’s version of toString(). This makes Span a special case, and we can only really get away with this kind of subclassing because we know the implementation details of the superclass. A robust subclassing mechanism needs to allow classes to invoke the methods and constructor of their superclass, but prior to ES6, JavaScript did not have a simple way to do these things.

1. *Subclasses with extends and super* -- In ES6 and later, you can create a superclass simply by adding an extends clause to a class declaration, and you can do this even for built-in classes:

class EZArray extends Array {

get first() { return this[0]; }

get last() { return this[this.length-1]; } }

let a = new EZArray();

a instanceof EZArray // => true: a is subclass instance

a instanceof Array // => true: a is also a superclass instance. a.push(1,2,3,4); // a.length == 4; we can use inherited methods a.pop() // => 4: another inherited method

a.first // => 1: first getter defined by subclass

a.last // => 3: last getter defined by subclass

a[1] // => 2: regular array access syntax still works.

Array.isArray(a) // => true: subclass instance really is an array EZArray.isArray(a) // => true: subclass inherits static methods, too!

EZArray inherits instance methods because EZArray.prototype inherits from Array.prototype Array.prototype.isPrototypeOf(EZArray.prototype) // => true

And EZArray inherits static methods and properties because EZArray inherits from Array. This is a special feature of the extends keyword and is not possible before ES6.

Array.isPrototypeOf(EZArray) // => true

*Example 9-6. TypedMap.js:*

class TypedMap extends Map {

constructor(keyType, valueType, entries) {

if (entries) {

for(let [k, v] of entries) {

if (typeof k !== keyType || typeof v !== valueType) {

throw new TypeError(`Wrong type for entry [${k}, ${v}]`); } } } super(entries);

this.keyType = keyType; this.valueType = valueType; }

set(key, value) {

if (this.keyType && typeof key !== this.keyType) {

throw new TypeError(`${key} is not of type ${this.keyType}`); }

if (this.valueType && typeof value !== this.valueType) {

throw new TypeError(`${value} is not of type ${this.valueType}`); } return super.set(key, value); } }

The Map() constructor takes one optional argument: an iterable object of [key,value] arrays. So the optional third argument of the TypedMap() constructor is the optional first argument to the Map() constructor, and we pass it to that superclass constructor with super(entries).

If you define a class with the extends keyword, then the constructor for your class must use super() to invoke the superclass constructor. If you don’t define a constructor in your subclass, one will be defined automatically for you. This implicitly defined constructor simply takes whatever values are passed to it and passes those values to super().

You may not use the this keyword in your constructor until after you have invoked the superclass constructor with super(). This enforces a rule that superclasses get to initialize themselves before subclasses do.

The special expression new.target is undefined in functions that are invoked without the new keyword. In constructor functions, however, new.target is a reference to the constructor that was invoked. When a subclass constructor is invoked and uses super() to invoke the superclass constructor, that superclass constructor will see the subclass constructor as the value of new.target. A welldesigned superclass should not need to know whether it has been subclassed, but it might be useful to be able to use new.target.name in logging messages.

As the class is written now, a user could change the keyType or valueType properties to subvert the type checking. Once private fields are supported, we could change these properties to #keyType and #valueType so that they could not be altered from the outside.

1. Delegation Instead of Inheritance The extends keyword makes it easy to create subclasses. But that does not mean that you should create lots of subclasses. If you want to write a class that shares the behavior of some other class, you can try to inherit that behavior by creating a subclass. But it is often easier and more flexible to get that desired behavior into your class by having your class create an instance of the other class and simply delegating to that instance as needed. You create a new class not by subclassing, but instead by wrapping or “composing” other classes. This delegation approach is often called “composition,” and it is an oft-quoted maxim of object-oriented programming that one should “favor composition over inheritance.”

Suppose, for example, we wanted a Histogram class that behaves something like JavaScript’s Set class, except that instead of just keeping track of whether a value has been added to set or not, it instead maintains a count of the number of times the value has been added. Because the API for this Histogram class is similar to Set, we might consider subclassing Set and adding a count() method. On the other hand, once we start thinking about how we might implement this count() method, we might realize that the Histogram class is more like a Map than a Set because it needs to maintain a mapping between values and the number of times they have been added. So instead of subclassing Set, we can create a class that defines a Set-like API but implements those methods by delegating to an internal Map object.

Example 9-7 Histogram.js:

class Histogram {

constructor() { this.map = new Map(); }

count(key) { return this.map.get(key) || 0; }

has(key) { return this.count(key) > 0; }

get size() { return this.map.size; }

add(key) { this.map.set(key, this.count(key) + 1); }

delete(key) { let count = this.count(key); if (count === 1) {

this.map.delete(key); } else if (count > 1) { this.map.set(key, count - 1); } }

[Symbol.iterator]() { return this.map.keys(); }

keys() { return this.map.keys(); }

values() { return this.map.values(); }

entries() { return this.map.entries(); } }

All the Histogram() constructor does in Example 9-7 is create a Map object. And most of the methods are one-liners that just delegate to a method of the map, making the implementation quite simple. Because we used delegation rather than inheritance, a Histogram object is not an instance of Set or Map.

1. Class Hierarchies and Abstract Classes -- classes that do not include a complete implementation—to serve as a common superclass for a group of related subclasses. An abstract superclass can define a partial implementation that all subclasses inherit and share. The subclasses, then, only need to define their own unique behavior by implementing the abstract methods defined—but not implemented—by the superclass. Note that JavaScript does not have any formal definition of abstract methods or abstract classes; I’m simply using that name here for unimplemented methods and incompletely implemented classes.
2. *Modules* -- Until recently, JavaScript had no built-in support for modules, and programmers working on large code bases did their best to use the weak modularity available through classes, objects, and closures. Closure-based modularity, with support from code-bundling tools, led to a practical form of modularity based on a require() function, which was adopted by Node. require()-based modules are a fundamental part of the Node programming environment but were never adopted as an official part of the JavaScript language. Instead, ES6 defines modules using import and export keywords.

*Modules with Classes, Objects, and Closures* -- The reason that the methods of one class are independent of the methods of other, unrelated classes is that the methods of each class are defined as properties of independent prototype objects. The reason that classes are modular is that objects are modular. Using classes and objects for modularity is a common and useful technique in JavaScript programming, but it doesn’t go far enough. In particular, it doesn’t offer us any way to hide internal implementation details inside the module.

local variables and nested functions declared within a function are private to that function. This means that we can use immediately invoked function expressions to achieve a kind of modularity by leaving the implementation details and utility functions hidden within the enclosing function but making the public API of the module the return value of the function.

const BitSet = (function() {

// Private implementation details here

function isValid(set, n) { ... }

function has(set, byte, bit) { ... }

const BITS = new Uint8Array([1, 2, 4, 8, 16, 32, 64, 128]);

const MASKS = new Uint8Array([~1, ~2, ~4, ~8, ~16, ~32, ~64, ~128]);

//The class can use the private functions and constants defined above, //but they will be hidden from users of the class

return class BitSet extends AbstractWritableSet { // ... implementation omitted ... }; }());

This approach to modularity becomes a little more interesting when the module has more than one item in it. The following code, for example, defines a mini statistics module that exports mean() and stddev() functions while leaving the implementation details hidden

const stats = (function() {

const sum = (x, y) => x + y;

const square = x => x \* x;

function mean(data) { return data.reduce(sum)/data.length;

function stddev(data) {

let m = mean(data);

return Math.sqrt( data.map(x => x - m).map(square).reduce(sum)/(data.length-1) ); }

return { mean, stddev }; }());

stats.mean([1, 3, 5, 7, 9]) // => 5

stats.stddev([1, 3, 5, 7, 9]) // => Math.sqrt(10)

1. *Node Exports* -- Node defines a global exports object that is always defined. If you are writing a Node module that exports multiple values, you can simply assign them to the properties of this object:

const sum = (x, y) => x + y;

const square = x => x \* x;

exports.mean = data => data.reduce(sum)/data.length;

exports.stddev = function(d) {

let m = exports.mean(d);

return Math.sqrt(d.map(x => x - m).map(square).reduce(sum)/(d.length-1)); };

Often, however, you want to define a module that exports only a single function or class rather than an object full of functions or classes. To do this, you simply assign the single value you want to export to module.exports:

module.exports = class BitSet extends AbstractWritableSet { // implementation omitted };

The default value of module.exports is the same object that exports refers to. In the previous stats module, we could have assigned the mean function to module.exports.mean instead of exports.mean.

Another approach with modules like the stats module is to export a single object at the end of the module rather than exporting functions one by one as you go:

const sum = (x, y) => x + y;

const square = x => x \* x;

const mean = data => data.reduce(sum)/data.length;

const stddev = d => { let m = mean(d); return Math.sqrt(d.map(x => x - m).map(square).reduce(sum)/(d.length-1)); };

module.exports = { mean, stddev };

1. *ES6 Exports* -- To export a constant, variable, function, or class from an ES6 module, simply add the keyword export before the declaration:

export const PI = Math.PI;

export function degreesToRadians(d) { return d \* PI / 180; }

export class Circle { constructor(r) { this.r = r; } area() { return PI \* this.r \* this.r; } }

As an alternative to scattering export keywords throughout your module, you can define your constants, variables, functions, and classes as you normally would, with no export statement, and then (typically at the end of your module) write a single export statement that declares exactly what is exported in a single place. So instead of writing three individual exports in the preceding code, we could have equivalently written a single line at the end: export { Circle, degreesToRadians, PI }; This syntax looks like the export keyword followed by an object literal (using shorthand notation). But in this case, the curly braces do not actually define an object literal. This export syntax simply requires a comma-separated list of identifiers within curly braces. It is common to write modules that export only one value (typically a function or class), and in this case, we usually use export default instead of export:

export default class BitSet { // implementation omitted }

Default exports are slightly easier to import than non-default exports, so when there is only one exported value, using export default makes things easier for the modules that use your exported value. Regular exports with export can only be done on declarations that have a name. Default exports with export default can export any expression including anonymous function expressions and anonymous class expressions. This means that if you use export default, you can export object literals. So unlike the export syntax, if you see curly braces after export default, it really is an object literal that is being exported.

It is legal, but somewhat uncommon, for modules to have a set of regular exports and also a default export. If a module has a default export, it can only have one. Finally, note that the export keyword can only appear at the top level of your JavaScript code. You may not export a value from within a class, function, loop, or conditional. (This is an important feature of the ES6 module system and enables static analysis: a modules export will be the same on every run, and the symbols exported can be determined before the module is actually run.)

1. ES6 Imports -- You import values that have been exported by other modules with the import keyword. The simplest form of import is used for modules that define a default export: import BitSet from './bitset.js'; This is the import keyword, followed by an identifier, followed by the from keyword, followed by a string literal that names the module whose default export we are importing. The default export value of the specified module becomes the value of the specified identifier in the current module. The identifier to which the imported value is assigned is a constant, as if it had been declared with the const keyword. Like exports, imports can only appear at the top level of a module and are not allowed within classes, functions, loops, or conditionals. By near-universal convention, the imports needed by a module are placed at the start of the module. Interestingly, however, this is not required: like function declarations, imports are “hoisted” to the top, and all imported values are available for any of the module’s code runs. The module from which a value is imported is specified as a constant string literal in single quotes or double quotes. (You may not use a variable or other expression whose value is a string, and you may not use a string within backticks because template literals can interpolate variables and do not always have constant values.)

A module specifier string must be an absolute path starting with “/”, or a relative path starting with “./” or “../”, or a complete URL a with protocol and hostname. The ES6 specification does not allow unqualified module specifier strings like “util.js” because it is ambiguous whether this is intended to name a module in the same directory as the current one or some kind of system module that is installed in some special location.

To import values from a module that exports multiple values, we use a slightly different syntax:

import { mean, stddev } from "./stats.js";

When importing from a module that defines many exports, however, you can easily import everything with an import statement like this:

import \* as stats from "./stats.js";

An import statement like this creates an object and assigns it to a constant named stats.

Modules typically define either one default export or multiple named exports. It is legal, but somewhat uncommon, for a module to use both export and export default. But when a module does that, you can import both the default value and the named values with an import statement like this:

import Histogram, { mean, stddev } from "./histogramstats.js";

To include a no-exports module into your program, simply use the import keyword with the module specifier, A module like this runs the first time it is imported.

import "./analytics.js";

You can use the as keyword with named imports to rename them as you import them:

import { render as renderImage } from "./imageutils.js";

import { render as renderUI } from "./ui.js";

Here is another way to import both the default and named exports of that module:

import { default as Histogram, mean, stddev } from "./histogram-stats.js";

As with imports, you use the as keyword to do this:

export { layout as calculateLayout, render as renderLayout };

Keep in mind that, although the curly braces look something like object literals, they are not, and the export keyword expects a single identifier before the as, not an expression. This means, unfortunately, that you cannot use export renaming like this:

export { Math.sin as sin, Math.cos as cos }; // SyntaxError

Instead of importing a symbol simply to export it again, you can combine the import and the export steps into a single “re-export” statement that uses the export keyword and the from keyword:

export { mean } from "./stats/mean.js";

export { stddev } from "./stats/stddev.js";

If we are not being selective with a re-export and simply want to export all of the named values from another module, we can use a wildcard:

export \* from "./stats/mean.js";

export \* from "./stats/stddev.js";

Re-export syntax allows renaming with as just as regular import and export statements do. Suppose we wanted to re-export the mean() function but also define average() as another name for the function. We could do that like this:

export { mean, mean as average } from "./stats/mean.js";

export { stddev } from "./stats/stddev.js";

export { default as mean } from "./stats/mean.js";

export { default as stddev } from "./stats/stddev.js";

export { mean as default } from "./stats.js"

export { default } from "./stats/mean.js"

So instead of importing the “./stats.js” module statically, like this:

import \* as stats from "./stats.js";

we might import it and use it dynamically, like this:

import("./stats.js").then(stats => { let average = stats.mean(data); })

Or, in an async function (again, you may need to read Chapter 13 before you’ll understand this code), we can simplify the code with await:

async analyzeData(data) {

let stats = await import("./stats.js");

return { average: stats.mean(data), stddev: stats.stddev(data) }; }

The argument to import() should be a module specifier, exactly like one you’d use with a static import directive. But with import(), you are not constrained to use a constant string literal: any expression that evaluates to a string in the proper form will do.

Dynamic import() looks like a function invocation, but it actually is not. Instead, import() is an operator and the parentheses are a required part of the operator syntax. The reason for this unusual bit of syntax is that import() needs to be able to resolve module specifiers as URLs relative to the currently running module, and this requires a bit of implementation magic that would not be legal to put in a JavaScript function. The function versus operator distinction rarely makes a difference in practice, but you’ll notice it if you try writing code like console.log(import); or let require = import;.

1. *The JavaScript Standard Library*

*The Set Class* -- Create a Set object with the Set() constructor:

let s = new 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"

You can add and remove elements at any time with add(), delete(), and clear().

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.

Sets are often described as “unordered collections.” This isn’t exactly true for the JavaScript Set class. 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. In addition to being iterable, the Set class also implements a forEach() method that is similar to the array method of the same name. 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.

*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.

let m = new Map();

let n = new Map([ ["one", 1], ["two", 2] ]);

let copy = new Map(n);

let o = { x: 1, y: 2};

let p = new Map(Object.entries(o));

let m = new Map

m.size // => 0

m.set("one", 1);

m.get("two") // => 2: return the value associated with key "two" m.has("one") // => true:

m.delete("one") // => true: the key existed and deletion succeeded m.clear(); // Remove all keys and values from the map

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(); m.set({}, 1); m.set({}, 2); m.size // => 2: m.get({}) // => undefined: m.set(m, undefined); // Map the map itself to the value undefined. m.has(m) // => true: m.get(m) // => undefined: same value we'd get if m wasn't a key

let m = new Map([["x", 1], ["y", 2]]);

[...m] // => [["x", 1], ["y", 2]]

for(let [key, value] of m) { }

[...m.keys()] // => ["x", "y"]: just the keys

[...m.values()] // => [1, 2]: just the values

[...m.entries()] // => [["x", 1], ["y", 2]]: same as [...m]

m.forEach((value, key) => { // note value, key NOT key, value });

*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()* -- It takes a regular expression as its first argument and a replacement string as its second argument. 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:

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…

*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"]

*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"]

*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.

*Dates and Times* -- 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

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().

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.

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 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) {

if (key[0] === "\_") return undefined;

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); }

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.

let text = JSON.stringify(address, ["city","state","country"]);

let json=JSON.stringify(o,(k, v)=>v instanceof RegExp ? undefined : v);

*console.assert()* -- If the first argument is truthy, then this function does nothing. But if the first argument is false or another falsy value, then the remaining arguments are printed

*console.table()* -- 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.

*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.

*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.

*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.

1. *URL APIs* -- 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#frag");

url.href//=>"https://example.com:8000/path/name?q=term#frag" 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"

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 value

url.search // => "?q=x"

url.searchParams.get("q") // => "x"

url.searchParams.has("q") // => true:

url.searchParams.append("opts", "1"); // Add another

url.search // => "?q=x&opts=1"

url.searchParams.append("opts", "&"); // Add another

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

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

1. *Timers* -- 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. Note that setTimeout() does not wait for the time to elapse before returning.

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.

let clock = setInterval(() => { console.clear(); console.log(new Date().toLocaleTimeString()); }, 1000);

// After 10 seconds: stop the repeating code above.

setTimeout(() => { clearInterval(clock); }, 10000);

1. *How Iterators Work* -- The for/of loop and spread operator work seamlessly with iterable objects, but it is worth understanding what is actually happening to make the iteration work. There are three separate types that you need to understand to understand iteration in JavaScript. First, there are the iterable objects: these are types like Array, Set, and Map that can be iterated. Second, there is the iterator object itself, which performs the iteration. And third, there is the iteration result object that holds the result of each step of the iteration. An iterable object is any object with a special iterator method that returns an iterator object. An iterator is any object with a next() method that returns an iteration result object. And an iteration result object is an object with properties named value and done.

To iterate an iterable object, you first call its iterator method to get an iterator object. Then, you call the next() method of the iterator object repeatedly until the returned value has its done property set to true. The tricky thing about this is that the iterator method of an iterable object does not have a conventional name but uses the Symbol Symbol.iterator as its name. So a simple for/of loop over an iterable object iterable could also be written the hard way, like this:

let iterable = [99];

let iterator = iterable[Symbol.iterator]();

for(let result = iterator.next(); !result.done; result = iterator.next()) { console.log(result.value) // result.value == 99 }

The iterator object of the built-in iterable datatypes is itself iterable. (That is, it has a method named Symbol.iterator that just returns itself.) This is occasionally useful in code like the following when you want to iterate though a “partially used” iterator:

let list = [1,2,3,4,5];

let iter = list[Symbol.iterator]();

let head = iter.next().value; // head == 1

let tail = [...iter]; // tail == [2,3,4,5]

In order to make a class iterable, you must implement a method whose name is the Symbol Symbol.iterator. That method must return an iterator object that has a next() method. And the next() method must return an iteration result object that has a value property and/or a boolean done property. Example 12-1 implements an iterable Range class and demonstrates how to create iterable, iterator, and iteration result objects.

Example 12-1. An iterable numeric Range class

class Range {

constructor (from, to) { this.from = from; this.to = to; }

has(x) { return typeof x === "number" && this.from <= x && x <= this.to; }

toString() { return `{ x | ${this.from} ≤ x ≤ ${this.to} }`; }

[Symbol.iterator]() {

let next = Math.ceil(this.from);

let last = this.to;

return {

next() {

return (next <= last) ? { value: next++ }: { done: true }; },

[Symbol.iterator]() { return this; } }; } }

for(let x of new Range(1,10))

console.log(x); // Logs numbers 1 to 10

[...new Range(-2,2)] // => [-2, -1, 0, 1, 2]

In addition to making your classes iterable, it can be quite useful to define functions that return iterable values.

function map(iterable, f) {

let iterator = iterable[Symbol.iterator]();

return { // This object is both iterator and iterable

[Symbol.iterator]() { return this; },

next() {

let v = iterator.next();

if (v.done) { return v; }

else { return { value: f(v.value) }; } } }; }

[...map(new Range(1,4), x => x\*x)] // => [1, 4, 9, 16]

function filter(iterable, predicate) {

let iterator = iterable[Symbol.iterator]();

return {

// This object is both iterator and iterable [Symbol.iterator]() { return this; },

next() {

for(;;) {

let v = iterator.next();

if (v.done || predicate(v.value)) { return v; } } } }; }

[...filter(new Range(1,10), x => x % 2 === 0)] // => [2,4,6,8,10]

One key feature of iterable objects and iterators is that they are inherently lazy: when computation is required to compute the next value, that computation can be deferred until the value is actually needed. Suppose, for example, that you have a very long string of text that you want to tokenize into space-separated words. You could simply use the split() method of your string, but if you do this, then the entire string has to be processed before you can use even the first word. And you end up allocating lots of memory for the returned array and all of the strings within it.

t iterators don’t always run all the way to the end: a for/of loop might be terminated with a break or return or by an exception. Similarly, when an iterator is used with destructuring assignment, the next() method is only called enough times to obtain values for each of the specified variables. The iterator may have many more values it could return, but they will never be requested.

For this reason, iterator objects may implement a return() method to go along with the next() method. If iteration stops before next() has returned an iteration result with the done property set to true (most commonly because you left a for/of loop early via a break statement), then the interpreter will check to see if the iterator object has a return() method. If this method exists, the interpreter will invoke it with no arguments, giving the iterator the chance to close files, release memory, and otherwise clean up after itself. The return() method must return an iterator result object. The properties of the object are ignored, but it is an error to return a non-object value.

1. *Generators* -- A generator is a kind of iterator defined with powerful new ES6 syntax; it’s particularly useful when the values to be iterated are not the elements of a data structure, but the result of a computation.

To create a generator, you must first define a generator function. A generator function is syntactically like a regular JavaScript function but is defined with the keyword function\* rather than function. (Technically, this is not a new keyword, just a \* after the keyword function and before the function name.) When you invoke a generator function, it does not actually execute the function body, but instead returns a generator object. This generator object is an iterator. Calling its next() method causes the body of the generator function to run from the start (or whatever its current position is) until it reaches a yield statement. yield is new in ES6 and is something like a return statement. The value of the yield statement becomes the value returned by the next() call on the iterator. An example makes this clearer:

function\* oneDigitPrimes() {

yield 2; yield 3; yield 5; yield 7; }

let primes = oneDigitPrimes();

primes.next().value // => 2 primes.next().value // => 3 primes.next().value // => 5 primes.next().value // => 7 primes.next().done // => true

[...oneDigitPrimes()] // => [2,3,5,7]

let sum = 0;

for(let prime of oneDigitPrimes()) sum += prime; sum // => 17

const seq = function\*(from,to) {

for(let i = from; i <= to; i++) yield i; };

let o = { x: 1, y: 2, z: 3,

\*g() { for(let key of Object.keys(this)) { yield key; } } };

[...o.g()] // => ["x", "y", "z", "g"]

Note that there is no way to write a generator function using arrow function syntax

1. *Generator Examples* -- Generators are more interesting if they actually generate the values they yield by doing some kind of computation. Here, for example, is a generator function that yields Fibonacci numbers:

function\* fibonacciSequence() {

let x = 0, y = 1;

for(;;) { yield y; [x, y] = [y, x+y]; } }

Note that the fibonacciSequence() generator function here has an infinite loop and yields values forever without returning. If this generator is used with the ... spread operator, it will loop until memory is exhausted and the program crashes. With care, it is possible to use it in a for/of loop, however:

function fibonacci(n) {

for(let f of fibonacciSequence()) { if (n-- <= 0) return f; } } fibonacci(20) // => 10946

This kind of infinite generator becomes more useful with a take() generator like this: // Yield the first n elements of the specified iterable

function\* take(n, iterable) {

let it = iterable[Symbol.iterator]();

while(n-- > 0) {

let next = it.next();

if (next.done) return;

else yield next.value; } }

[...take(5, fibonacciSequence())] // => [1, 1, 2, 3, 5]

Here is another useful generator function that interleaves the elements of multiple iterable objects:

function\* zip(...iterables) {

let iterators = iterables.map(i => i[Symbol.iterator]());

let index = 0;

while(iterators.length > 0) {

if (index >= iterators.length) index = 0;

let item = iterators[index].next();

if (item.done) iterators.splice(index, 1);

else { yield item.value; index++; } } }

[...zip(oneDigitPrimes(),"ab",[0])] // => [2,"a",0,3,"b",5,7]

1. *yield\* and Recursive Generators* –

yields the elements of multiple iterable objects sequentially

function\* sequence(...iterables) {

for(let iterable of iterables) {

for(let item of iterable) { yield item; } } }

[...sequence("abc",oneDigitPrimes())] // => ["a","b","c",2,3,5,7]

This process of yielding the elements of some other iterable object is common enough in generator functions that ES6 has special syntax for it. The yield\* keyword is like yield except that, rather than yielding a single value, it iterates an iterable object and yields each of the resulting values.

function\* sequence(...iterables) {

for(let iterable of iterables) { yield\* iterable; } }

[...sequence("abc",oneDigitPrimes())] // => ["a","b","c",2,3,5,7]

yield and yield\* can only be used only within generator functions, can not be used in arrow functions. yield\* can be used with any kind of iterable object, including iterables implemented with generators. This means that yield\* allows us to define recursive generators.

1. *The Return Value of a Generator Function* –

function\* oneAndDone() { yield 1; return "done"; }

// The return value does not appear in normal iteration. [...oneAndDone()] // => [1]

// But it is available if you explicitly call next()

let generator = oneAndDone();

generator.next() // => { value: 1, done: false}

generator.next() // => { value: "done", done: true }

// If the generator is already done, the return value is not returned again generator.next() // => { value: undefined, done: true }

1. *The Value of a yield Expression* -- In the preceding discussion, we’ve treated yield as a statement that takes a value but has no value of its own. In fact, however, yield is an expression, and it can have a value. When the next() method of a generator is invoked, the generator function runs until it reaches a yield expression. The expression that follows the yield keyword is evaluated, and that value becomes the return value of the next() invocation. At this point, the generator function stops executing right in the middle of evaluating the yield expression. The next time the next() method of the generator is called, the argument passed to next() becomes the value of the yield expression that was paused. So the generator returns values to its caller with yield, and the caller passes values in to the generator with next(). The generator and caller are two separate streams of execution passing values (and control) back and forth. The following code illustrates:

function\* smallNumbers() {

console.log("next() invoked the first time; argument discarded");

let y1 = yield 1; // y1 == "b"

console.log("next() invoked a second time with argument", y1);

let y2 = yield 2; // y2 == "c"

console.log("next() invoked a third time with argument", y2);

let y3 = yield 3; // y3 == "d"

console.log("next() invoked a fourth time with argument", y3);

return 4; }

let g = smallNumbers();

console.log("generator created; no code runs yet");

let n1 = g.next("a"); // n1.value == 1

console.log("generator yielded", n1.value);

let n2 = g.next("b"); // n2.value == 2

console.log("generator yielded", n2.value);

let n3 = g.next("c"); // n3.value == 3

console.log("generator yielded", n3.value);

let n4 = g.next("d"); // n4 == { value: 4, done: true } console.log("generator returned", n4.value);

When this code runs, it produces the following output that demonstrates the back-and-forth between the two blocks of code:

generator created; no code runs yet

next() invoked the first time; argument discarded

generator yielded 1

next() invoked a second time with argument b

generator yielded 2

next() invoked a third time with argument c

generator yielded 3

next() invoked a fourth time with argument d

generator returned 4

Note the asymmetry in this code. The first invocation of next() starts the generator, but the value passed to that invocation is not accessible to the generator.

1. *The return() and throw() Methods of a Generator* –

function\* gen() {

while (true) {

try { yield 42; yield 43; yield 44; }

catch (e) { console.log("err!!"); yield “catch” } } }

const g = gen();

console.log(g.next()); // =>{ value: 42, done: false }

console.log(g.throw("err")); //=> err!!{value:”catch”,done: false }

console.log(g.next()); // =>{ value: 42, done: false }

console.log(g.next()); // =>{ value: 43, done: false }

console.log(g.return("return")); // =>{ value: "return", done: true }

console.log(g.next()); // =>{ value: undefined, done: true }

1. *HANDLING ERRORS WITH PROMISES* -- Asynchronous operations, particularly those that involve networking, can typically fail in a number of ways, and robust code has to be written to handle the errors that will inevitably occur. For Promises, we can do this by passing a second function to the then() method:

getJSON("/api/user/profile").then(displayUserProfile, handleProfileError);

1. Let’s return to a simplified form of the original fetch() chain above. If we define the functions passed to the then() invocations elsewhere, we might refactor the code to look like this:

fetch(theURL) // task 1; returns promise 1

.then(callback1) // task 2; returns promise 2

.then(callback2); // task 3; returns promise 3

Let’s walk through this code in detail:

* On the first line, fetch() is invoked with a URL. It initiates an HTTP GET request for that URL and returns a Promise. We’ll call this HTTP request “task 1” and we’ll call the Promise “promise 1”.
* On the second line, we invoke the then() method of promise 1, passing the callback1 function that we want to be invoked when promise 1 is fulfilled. The then() method stores our callback somewhere, then returns a new Promise. We’ll call the new Promise returned at this step “promise 2”, and we’ll say that “task 2” begins when callback1 is invoked.
* On the third line, we invoke the then() method of promise 2, passing the callback2 function we want invoked when promise 2 is fulfilled. This then() method remembers our callback and returns yet another Promise. We’ll say that “task 3” begins when callback2 is invoked. We can call this latest Promise “promise 3”, but we don’t really need a name for it because we won’t be using it at all.
* The previous three steps all happen synchronously when the expression is first executed. Now we have an asynchronous pause while the HTTP request initiated in step 1 is sent out across the internet.
* Eventually, the HTTP response starts to arrive. The asynchronous part of the fetch() call wraps the HTTP status and headers in a Response object and fulfills promise 1 with that Response object as the value.
* When promise 1 is fulfilled, its value (the Response object) is passed to our callback1() function, and task 2 begins. The job of this task, given a Response object as input, is to obtain the response body as a JSON object.
* Let’s assume that task 2 completes normally and is able to parse the body of the HTTP response to produce a JSON object. This JSON object is used to fulfill promise 2.
* The value that fulfills promise 2 becomes the input to task 3 when it is passed to the callback2() function. This third task now displays the data to the user in some unspecified way. When task 3 is complete (assuming it completes normally), then promise 3 will be fulfilled. But because we never did anything with promise 3, nothing happens when that Promise settles, and the chain of asynchronous computation ends at this point.

1. *Resolving Promises* --

function c1(response) {

let p4 = response.json(); return p4; }

function c2(profile) { displayUserProfile(profile); }

let p1 = fetch("/api/user/profile"); // promise 1, task 1

let p2 = p1.then(c1); // promise 2, task 2

let p3 = p2.then(c2); // promise 3, task 3

Promises are about managing asynchronous tasks, after all, and if task 2 is asynchronous (which it is, in this case), then that task will not be complete by the time the callback returns. We are now ready to discuss the final detail that you need to understand to really master Promises.

When you pass a callback c to the then() method, then() returns a Promise p and arranges to asynchronously invoke c at some later time. The callback performs some computation and returns a value v. When the callback returns, p is resolved with the value v. When a Promise is resolved with a value that is not itself a Promise, it is immediately fulfilled with that value. So if c returns a non-Promise, that return value becomes the value of p, p is fulfilled and we are done. But if the return value v is itself a Promise, then p is resolved but not yet fulfilled. At this stage, p cannot settle until the Promise v settles. If v is fulfilled, then p will be fulfilled to the same value. If v is rejected, then p will be rejected for the same reason. This is what the “resolved” state of a Promise means: the Promise has become associated with, or “locked onto,” another Promise. We don’t know yet whether p will be fulfilled or rejected, but our callback c no longer has any control over that. p is “resolved” in the sense that its fate now depends entirely on what happens to Promise v.

Let’s bring this back to our URL-fetching example. When c1 returns p4, p2 is resolved. But being resolved is not the same as being fulfilled, so task 3 does not begin yet. When the full body of the HTTP response becomes available, then the .json() method can parse it and use that parsed value to fulfill p4. When p4 is fulfilled, p2 is automatically fulfilled as well, with the same parsed JSON value. At this point, the parsed JSON object is passed to c2, and task 3 begins.

1. *THE CATCH AND FINALLY METHODS* -- The .catch() method of a Promise is simply a shorthand way to call .then() with null as the first argument and an error-handling callback as the second argument. Given any Promise p and a callback c, the following two lines are equivalent:

p.then(null, c);

p.catch(c);

In ES2018, Promise objects also define a .finally() method whose purpose is similar to the finally clause in a try/catch/finally statement. If you add a .finally() invocation to your Promise chain, then the callback you pass to .finally() will be invoked when the Promise you called it on settles. Your callback will be invoked if the Promise fulfills or rejects, and it will not be passed any arguments, so you can’t find out whether it fulfilled or rejected. But if you need to run some kind of cleanup code (such as closing open files or network connections) in either case, a .finally() callback is the ideal way to do that. Like .then() and .catch(), .finally() returns a new Promise object. The return value of a .finally() callback is generally ignored.

If one of the stages in your Promise chain can fail with an error, and if the error is some kind of recoverable error that should not stop the rest of the chain from running, then you can insert a .catch() call in the chain, resulting in code that might look like this:

startAsyncOperation()

.then(doStageTwo)

.catch(recoverFromStageTwoError)

.then(doStageThree)

.then(doStageFour)

.catch(logStageThreeAndFourErrors);

Remember that the callback you pass to .catch() will only be invoked if the callback at a previous stage throws an error. If the callback returns normally, then the .catch() callback will be skipped, and the return value of the previous callback will become the input to the next .then() callback. Also remember that .catch() callbacks are not just for reporting errors, but for handling and recovering from errors. Once an error has been passed to a .catch() callback, it stops propagating down the Promise chain. A .catch() callback can throw a new error, but if it returns normally, than that return value is used to resolve and/or fulfill the associated Promise, and the error stops propagating.

Now suppose that transient network load issues are causing this to fail about 1% of the time. A simple solution might be to retry the query with a .catch() call:

queryDatabase()

.catch(e => wait(500).then(queryDatabase))

.then(displayTable)

.catch(displayDatabaseError);

If the hypothetical failures are truly random, then adding this one line of code should reduce your error rate from 1% to .01%.

1. *Promises in Parallel* -- Sometimes, though, we want to execute a number of asynchronous operations in parallel. The function Promise.all() can do this. Promise.all() takes an array of Promise objects as its input and returns a Promise. The returned Promise will be rejected if any of the input Promises are rejected. Otherwise, it will be fulfilled with an array of the fulfillment values of each of the input Promises.

const urls = [ /\* zero or more URLs here \*/ ];

promises = urls.map(url => fetch(url).then(r => r.text()));

Promise.all(promises).then(bodies=>{}).catch(e => console.error(e));

The input array can contain both Promise objects and non-Promise values. If an element of the array is not a Promise, it is treated as if it is the value of an already fulfilled Promise and is simply copied unchanged into the output array. The Promise returned by Promise.all() rejects when any of the input Promises is rejected. This happens immediately upon the first rejection and can happen while other input Promises are still pending.

In ES2020, Promise.allSettled() takes an array of input Promises and returns a Promise, just like Promise.all() does. But Promise.allSettled() never rejects the returned Promise, and it does not fulfill that Promise until all of the input Promises have settled. The Promise resolves to an array of objects, with one object for each input Promise. Each of these returned objects has a status property set to “fulfilled” or “rejected.” If the status is “fulfilled”, then the object will also have a value property that gives the fulfillment value. And if the status is “rejected”, then the object will also have a reason property that gives the error or rejection value of the corresponding Promise:

Promise.allSettled([Promise.resolve(1),Promise.reject(2),3])

.then(results => {

results[0] // => { status: "fulfilled", value: 1 }

results[1] // => { status: "rejected", reason: 2 }

results[2] // => { status: "fulfilled", value: 3 } });

Occasionally, you may want to run a number of Promises at once but may only care about the value of the first one to fulfill. In that case, you can use Promise.race() instead of Promise.all(). It returns a Promise that is fulfilled or rejected when the first of the Promises in the input array is fulfilled or rejected. (Or, if there are any non-Promise values in the input array, it simply returns the first of those.)

1. *PROMISES BASED ON SYNCHRONOUS VALUES* -- Promise.resolve() takes a value as its single argument and returns a Promise that will immediately (but asynchronously) be fulfilled to that value. Similarly, Promise.reject() takes a single argument and returns a Promise that will be rejected with that value as the reason. (To be clear: the Promises returned by these static methods are not already fulfilled or rejected when they are returned, but they will fulfill or reject immediately after the current synchronous chunk of code has finished running. Typically, this happens within a few milliseconds unless there are many pending asynchronous tasks waiting to run.)
2. PROMISES FROM SCRATCH -- Here’s how to write the Promise-based wait() function.

function wait(duration) {

return new Promise((resolve, reject) => {

if (duration < 0)

reject(new Error("Time travel not yet implemented"));

// Otherwise, wait asynchronously and then resolve the Promise. // setTimeout will invoke resolve() with no arguments, which means // that the Promise will fulfill with the undefined value.

setTimeout(resolve, duration); }); }

1. *Promises in Sequence* -- Promise.all() makes it easy to run an arbitrary number of Promises in parallel. And Promise chains make it easy to express a sequence of a fixed number of Promises. Running an arbitrary number of Promises in sequence is trickier, however. Suppose, for example, that you have an array of URLs to fetch, but that to avoid overloading your network, you want to fetch them one at a time. If the array is of arbitrary length and unknown content, you can’t write out a Promise chain in advance, so you need to build one dynamically, with code like this:

function fetchSequentially(urls) {

const bodies = [];

function fetchOne(url) {

return axios .get(url)

.then((response) => response.data)

.then((body) => { bodies.push(body); }); } let p = Promise.resolve(undefined);

for (let url of urls) p = p.then(() => fetchOne(url));

return p.then(() => bodies); }

fetchSequentially(urls)

.then((bodies) => { console.log(bodies); })

.catch((e) => console.error(e));

There is another (possibly more elegant) approach that we can take. Rather than creating the Promises in advance, we can have the callback for each Promise create and return the next Promise. That is, instead of creating and chaining a bunch of Promises, we instead create Promises that resolve to other Promises.

function promiseSequence(inputs, promiseMaker) {

inputs = [...inputs];

function handleNextInput(outputs) {

if (inputs.length === 0) return outputs;

else return promiseMaker(inputs.shift())

.then((output) => outputs.concat(output))

.then(handleNextInput); }

return Promise.resolve([]).then(handleNextInput); }

function fetchBody(url) { return axios.get(url).then((r) => r.data); }

promiseSequence(urls, fetchBody)

.then((bodies) => { console.log(bodies); })

.catch(console.error);

1. *await Expressions* -- The await keyword takes a Promise and turns it back into a return value or a thrown exception. Given a Promise object p, the expression await p waits until p settles. If p fulfills, then the value of await p is the fulfillment value of p. On the other hand, if p is rejected, then the await p expression throws the rejection value of p.

It is critical to understand right away that the await keyword does not cause your program to block and literally do nothing until the specified Promise settles. The code remains asynchronous, and the await simply disguises this fact. This means that any code that uses await is itself asynchronous.

1. async Functions -- Declaring a function async means that the return value of the function will be a Promise even if no Promise-related code appears in the body of the function.

You can use the async keyword with any kind of function. It works with the function keyword as a statement or as an expression. It works with arrow functions and with the method shortcut form in classes and object literals.

1. Awaiting Multiple Promises –

let [value1, value2] = await Promise.all([getJSON(url1), getJSON(url2)]);

let responses = await Promise.allSettled([getJSON(url1), getJSON(url2)]);

1. *Implementation Details* -- Finally, in order to understand how async functions work, it may help to think about what is going on under the hood. Suppose you write an async function like this:

async function f(x) { /\* body \*/ }

You can think about this as a Promise-returning function wrapped around the body of your original function:

function f(x) {

return new Promise(function(resolve, reject) {

try {

resolve((function(x) { /\* body \*/ })(x)); }

catch(e) { reject(e); } }); }

It is harder to express the await keyword in terms of a syntax transformation like this one. But think of the await keyword as a marker that breaks a function body up into separate, synchronous chunks. An ES2017 interpreter can break the function body up into a sequence of separate subfunctions, each of which gets passed to the then() method of the await-marked Promise that precedes it.

1. *The for/await Loop* -- Node 12 makes its readable streams asynchronously iterable. This means you can read successive chunks of data from a stream with a for/await loop like this one:

const fs = require("fs");

async function parseFile(filename) {

let stream = fs.createReadStream(filename, { encoding: "utf-8"});

for await (let chunk of stream) { parseChunk(chunk); } }

Like a regular await expression, the for/await loop is Promisebased. Roughly speaking, the asynchronous iterator produces a Promise and the for/await loop waits for that Promise to fulfill, assigns the fulfillment value to the loop variable, and runs the body of the loop. And then it starts over, getting another Promise from the iterator and waiting for that new Promise to fulfill.

const urls = [url1, url2, url3];

const promises = urls.map(url => fetch(url));

for(const promise of promises) {

response = await promise; handle(response); }

for await (const response of promises) { handle(response); }

It is important to realize, however, that we’re using for/await with a regular iterator in this example. Things are more interesting with fully asynchronous iterators.

1. *Asynchronous Iterators* -- Asynchronous iterators are quite similar to regular iterators, but there are two important differences. First, an asynchronously iterable object implements a method with the symbolic name Symbol.asyncIterator instead of Symbol.iterator. (As we saw earlier, for/await is compatible with regular iterable objects but it prefers asynchronously iterable objects, and tries the Symbol.asyncIterator method before it tries the Symbol.iterator method.) Second, the next() method of an asynchronous iterator returns a Promise that resolves to an iterator result object instead of returning an iterator result object directly.

In the previous section, when we used for/await on a regular, synchronously iterable array of Promises, we were working with synchronous iterator result objects in which the value property was a Promise object but the done property was synchronous. True asynchronous iterators return Promises for iteration result objects, and both the value and the done properties are asynchronous. The difference is a subtle one: with asynchronous iterators, the choice about when iteration ends can be made asynchronously.

1. *Asynchronous Generators* –

function elapsedTime(ms) {

return new Promise((resolve) => setTimeout(resolve, ms)); }

async function\* clock(interval, max = Infinity) {

for (let count = 1; count <= max; count++) {

await elapsedTime(interval); yield count; } }

async function test() {

for await (let tick of clock(300, 100)) { console.log(tick); } };

1. *Implementing Asynchronous Iterators* -- Instead of using async generators to implement asynchronous iterators, it is also possible to implement them directly by defining an object with a Symbol.asyncIterator() method that returns an object with a next() method that returns a Promise that resolves to an iterator result object. In the following code, we re-implement the clock() function from the preceding example so that it is not a generator and instead just returns an asynchronously iterable object. Notice that the next() method in this example does not explicitly return a Promise; instead, we just declare next() to be async:

function clock(interval, max = Infinity) {

function until(time) {

return new Promise(

(resolve) => setTimeout(resolve, time - Date.now())); }

return {

startTime: Date.now(),

count: 1,

async next() {

if (this.count > max) return { done: true };

let targetTime = this.startTime + this.count \* interval;

await until(targetTime);

return { value: this.count++ }; },

[Symbol.asyncIterator]() { return this; }, }; }

The for/await loop always waits for the Promise returned by one iteration to be fulfilled before it begins the next iteration. But if you use an asynchronous iterator without a for/await loop, there is nothing to prevent you from calling the next() method whenever you want. With the generator-based version of clock(), if you call the next() method three times at a time, you’ll get three Promises that will all fulfill one after the other, which is probably not what you want. The iterator-based version we’ve implemented here does not have that problem.

class AsyncQueue {

constructor() {

this.values = []; this.resolvers = []; this.closed = false; }

enqueue(value) {

if (this.closed) throw new Error("AsyncQueue closed");

if (this.resolvers.length > 0) {

const resolve = this.resolvers.shift();

resolve(value); }

else { this.values.push(value); } }

dequeue() {

if (this.values.length > 0) {

const value = this.values.shift();

return Promise.resolve(value); }

else if (this.closed) {

return Promise.resolve(AsyncQueue.EOS); }

else {

return new Promise(

(resolve) => { this.resolvers.push(resolve); }); } }

close() {

while (this.resolvers.length > 0) {

this.resolvers.shift()(AsyncQueue.EOS); }

this.closed = true; }

[Symbol.asyncIterator]() { return this; }

next() {

return this.dequeue().then(

(value) => value === AsyncQueue.EOS ?

{ value: undefined, done: true } :

{ value: value, done: false } ); } } AsyncQueue.EOS = Symbol("end-of-stream");

let a = new AsyncQueue();

a.dequeue().then(r=>console.log(r)) //=> 1

a.enqueue(1);

Because this AsyncQueue class defines the asynchronous iteration basics, we can create our own, more interesting asynchronous iterators simply by asynchronously queueing values. Here’s an example that uses AsyncQueue to produce a stream of web browser events that can be handled with a for/await loop: // Push events of the specified type on the specified document element // onto an AsyncQueue object, and return the queue for use as an event stream

function eventStream(elt, type) {

const q = new AsyncQueue(); // Create a queue

elt.addEventListener(type, e=>q.enqueue(e));

return q; }

async function handleKeys() {

// Get a stream of keypress events and loop once for each one

for await (const event of eventStream(document, "keypress")) {

console.log(event.key); } }

1. *Metaprogramming* -- *Property Attributes* -- The properties of a JavaScript object have names and values, of course, but each property also has three associated attributes that specify how that property behaves and what you can do with it:

* The writable attribute specifies whether or not the value of a property can change.
* The enumerable attribute specifies whether the property is enumerated by the for/in loop and the Object.keys() method.
* The configurable attribute specifies whether a property can be deleted and also whether the property’s attributes can be changed.

Properties defined in object literals or by ordinary assignment to an object are writable, enumerable, and configurable. But many of the properties defined by the JavaScript standard library are not.

1. we can say that a property has a name and four attributes. The four attributes of a data property are value, writable, enumerable, and configurable. Accessor properties don’t have a value attribute or a writable attribute: their writability is determined by the presence or absence of a setter. So the four attributes of an accessor property are get, set, enumerable, and configurable.

The JavaScript methods for querying and setting the attributes of a property use an object called a property descriptor. Thus, the property descriptor object of a data property has properties named value, writable, enumerable, and configurable. And the descriptor for an accessor property has get and set properties instead of value and writable. The writable, enumerable, and configurable properties are boolean values, and the get and set properties are function values.

Object.getOwnPropertyDescriptor({x: 1}, "x");

{value: 1, writable:true, enumerable:true, configurable:true}

const random = {

get octet() { return Math.floor(Math.random()\*256); }, };

Object.getOwnPropertyDescriptor(random, "octet");

{ get: /\*func\*/, set:undefined, enumerable:true, configurable:true}

Object.getOwnPropertyDescriptor({}, "x") // => undefined; no prop Object.getOwnPropertyDescriptor({},"toString") //undefined inherited

As its name implies, Object.getOwnPropertyDescriptor() works only for own properties.

1. To set the attributes of a property or to create a new property with the specified attributes, call Object.defineProperty(), passing the object to be modified, the name of the property to be created or altered, and the property descriptor object:

let o = {};

Object.defineProperty(o, "x", { value: 1, writable: true, enumerable: false, configurable: true });

o.x // => 1

Object.keys(o) // => []

Object.defineProperty(o, "x", { writable: false });

o.x = 2; // Fails silently or throws TypeError in strict mode

o.x // => 1

// The property is still configurable, so we can change its value like this: Object.defineProperty(o, "x", { value: 2 }); o.x // => 2

// Now change x from a data property to an accessor property Object.defineProperty(o, "x", { get: function() { return 0; } });

o.x // => 0

The property descriptor you pass to Object.defineProperty() does not have to include all four attributes. If you’re creating a new property, then omitted attributes are taken to be false or undefined. If you’re modifying an existing property, then the attributes you omit are simply left unchanged. Note that this method alters an existing own property or creates a new own property, but it will not alter an inherited property.

If you want to create or modify more than one property at a time, use Object.defineProperties().

let p = Object.defineProperties({}, {

x: { value: 1, writable: true, enumerable: true, configurable: true },

y: { value: 1, writable: true, enumerable: true, configurable: true },

r: {

get() { return Math.sqrt(this.x\*this.x + this.y\*this.y); },

enumerable: true, configurable: true } });

p.r // => Math.SQRT2

The Object.create() method, the first argument to that method is the prototype object for the newly created object. This method also accepts a second optional argument, which is the same as the second argument to Object.defineProperties(). If you pass a set of property descriptors to Object.create(), then they are used to add properties to the newly created object.

Note -- If a data property is not configurable, you cannot change its writable attribute from false to true, but you can change it from true to false.

Object.assign() only copies enumerable properties, and property values, not property attributes. This is normally what we want, but it does mean, for example, that if one of the source objects has an accessor property, it is the value returned by the getter function that is copied to the target object, not the getter function itself.

Object.defineProperty(Object, "assignDescriptors", {

  writable: true,

  enumerable: false,

  configurable: true,

  value: function (target, ...sources) {

    for (let source of sources) {

      for (let name of Object.getOwnPropertyNames(source)) {

        let desc = Object.getOwnPropertyDescriptor(source, name);

        Object.defineProperty(target, name, desc);

      }

      for (let sym of Object.getOwnPropertySymbols(source)) {

      let desc = Object.getOwnPropertyDescriptor(source, sym);

        Object.defineProperty(target, sym, desc);

      }

    }

    return target;

  },

});

let o = {

  c: 1,

  get count() {

    return this.c++;

  },

};

let p = Object.assign({}, o); //=>{c: 1,count: 1 }

let q = Object.assignDescriptors({}, o);

//=>{c: 2,count:[Getter]}

1. Object Extensibility The extensible attribute of an object specifies whether new properties can be added to the object or not. Ordinary JavaScript objects are extensible by default, but you can change that with the functions described in this section. To determine whether an object is extensible, pass it to Object.isExtensible(). To make an object non-extensible, pass it to Object.preventExtensions(). Once you have done this, any attempt to add a new property to the object will throw a TypeError in strict mode and simply fail silently without an error in non-strict mode. In addition, attempting to change the prototype of a non-extensible object will always throw a TypeError. Note that there is no way to make an object extensible again once you have made it non-extensible. Also note that calling Object.preventExtensions() only affects the extensibility of the object itself. If new properties are added to the prototype of a non- extensible object, the non-extensible object will inherit those new properties.

The purpose of the extensible attribute is to be able to “lock down” objects into a known state and prevent outside tampering. The extensible attribute of objects is often used in conjunction with the configurable and writable attributes of properties, and JavaScript defines functions that make it easy to set these attributes together:

* Object.seal() works like Object.preventExtensions(), but in addition to making the object non-extensible, it also makes all of the own properties of that object nonconfigurable. This means that new properties cannot be added to the object, and existing properties cannot be deleted or configured. Existing properties that are writable can still be set, however. There is no way to unseal a sealed object. You can use Object.isSealed() to determine whether an object is sealed.
* Object.freeze() locks objects down even more tightly. In addition to making the object non-extensible and its properties nonconfigurable, it also makes all of the object’s own data properties read-only. (If the object has accessor properties with setter methods, these are not affected and can still be invoked by assignment to the property.) Use Object.isFrozen() to determine if an object is frozen.

It is important to understand that Object.seal() and Object.freeze() affect only the object they are passed: they have no effect on the prototype of that object. If you want to thoroughly lock down an object, you probably need to seal or freeze the objects in the prototype chain as well.

Object.preventExtensions(), Object.seal(), and Object.freeze() all return the object that they are passed, which means that you can use them in nested function invocations: // Create a sealed object with a frozen prototype and a nonenumerable property

let o = Object.seal(Object.create(Object.freeze({x: 1}), {y: {value: 2, writable: true}}));

1. *The prototype Attribute* -- An object’s prototype attribute specifies the object from which it inherits properties. The prototype attribute is set when an object is created. Objects created from object literals use Object.prototype as their prototype. Objects created with new use the value of the prototype property of their constructor function as their prototype. And objects created with Object.create() use the first argument to that function (which may be null) as their prototype.

Object.getPrototypeOf({}) // => Object.prototype

Object.getPrototypeOf([]) // => Array.prototype

Object.getPrototypeOf(()=>{}) // => Function.prototype

let p = {x: 1}; // Define a prototype object.

let o = Object.create(p); // Create an object with that prototype. p.isPrototypeOf(o) // => true: o inherits from p

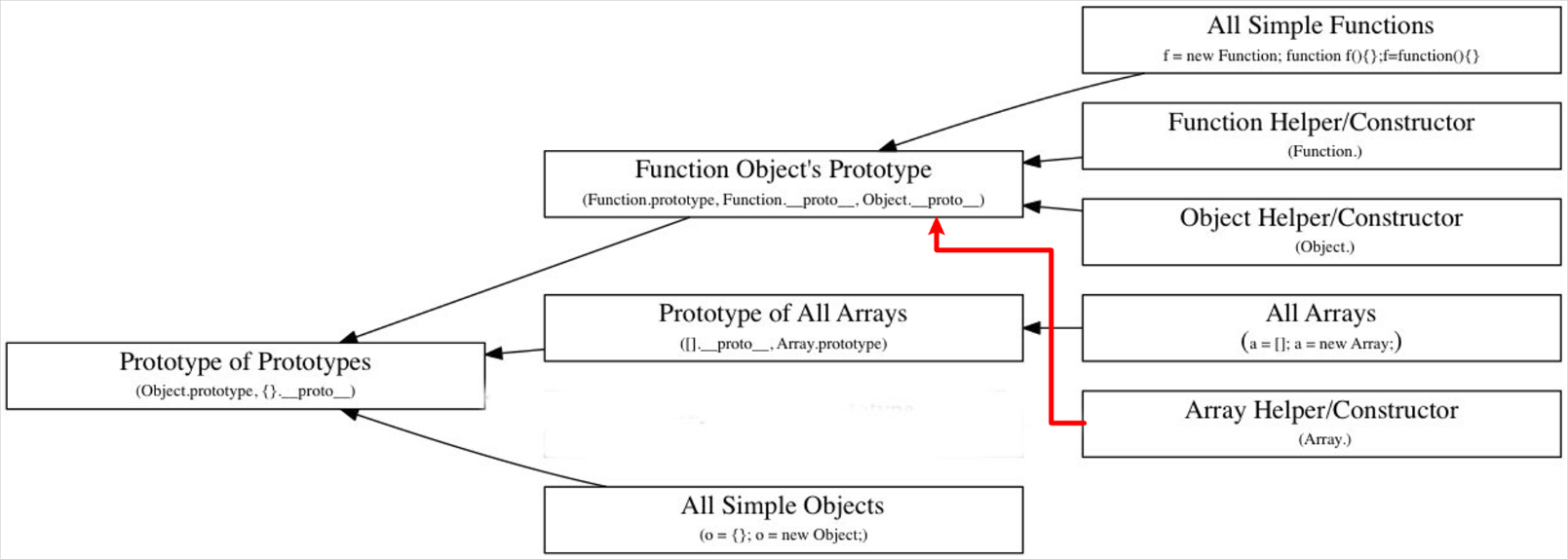
Object.prototype.isPrototypeOf(p) // => true: p inherits from Object.prototype Object.prototype.isPrototypeOf(o) // => true: o does too

Some early browser implementations of JavaScript exposed the prototype attribute of an object through the \_\_proto\_\_ property. This has long since been deprecated, but enough existing code on the web depends on \_\_proto\_\_ that the ECMAScript standard mandates it for all JavaScript implementations that run in web browsers. (Node supports it, too, though the standard does not require it for Node.) In modern JavaScript, \_\_proto\_\_ is readable and writeable. One interesting use of \_\_proto\_\_, however, is to define the prototype of an object literal:

let p = {z: 3};

let o = { x: 1, y: 2, \_\_proto\_\_: p };

o.z // => 3: o inherits from p



1. *Well-Known Symbols* -- Symbol.iterator is the best-known example of the “well-known Symbols.” These are a set of Symbol values stored as properties of the Symbol() factory function that are used to allow JavaScript code to control certain low-level behaviors of objects and classes.

*Symbol.iterator and Symbol.asyncIterator* -- The Symbol.iterator and Symbol.asyncIterator Symbols allow objects or classes to make themselves iterable or asynchronously iterable.

*Symbol.hasInstance* -- the expression o instanceof f was evaluated by looking for the value f.prototype within the prototype chain of o. That is still true, but in ES6 and beyond, Symbol.hasInstance provides an alternative. In ES6, if the righthand side of instanceof is any object with a [Symbol.hasInstance] method, then that method is invoked with the lefthand side value as its argument, and the return value of the method, converted to a boolean, becomes the value of the instanceof operator. And, of course, if the value on the righthand side does not have a [Symbol.hasInstance] method but is a function, then the instanceof operator behaves in its ordinary way. Symbol.hasInstance means that we can use the instanceof operator to do generic type checking with suitably defined pseudotype objects. For example:

let uint8 = {

[Symbol.hasInstance](x) {

return Number.isInteger(x) && x >= 0 && x <= 255; } };

128 instanceof uint8 // => true

256 instanceof uint8 // => false: too big

Math.PI instanceof uint8 // => false: not an integer

Note that this example is clever but confusing because it uses a nonclass object where a class would normally be expected. It would be just as easy—and clearer to readers of your code—to write a isUint8() function instead of relying on this Symbol.hasInstance behavior.

*Symbol.toStringTag* -- You can use this Object.prototype.toString().call() technique with any JavaScript value to obtain the “class attribute” of an object that contains type information that is not otherwise available:

Object.prototype.toString.call([]) // => "[object Array]" Object.prototype.toString.call(/./) // => "[object RegExp]" Object.prototype.toString.call(()=>{}) // => "[object Function]" Object.prototype.toString.call("") // => "[object String]" Object.prototype.toString.call(0) // => "[object Number]" Object.prototype.toString.call(false) // => "[object Boolean]"

function classof(o) { return Object.prototype.toString.call(o).slice(8,-1); } classof(null) // => "Null" classof(undefined) // => "Undefined"

classof(1) // => "Number" classof(10n\*\*100n) // => "BigInt"

classof("") // => "String" classof(false) // => "Boolean"

classof(Symbol()) // => "Symbol" classof({}) // => "Object"

classof([]) // => "Array" classof(/./) // => "RegExp"

classof(()=>{}) // => "Function" classof(new Map()) // => "Map"

classof(new Set()) // => "Set" classof(new Date()) // => "Date"

if you called this classof() function on an instance of a class you had defined yourself, it would simply return “Object”. In ES6, however, Object.prototype.toString() looks for a property with the symbolic name Symbol.toStringTag on its argument, and if such a property exists, it uses the property value in its output.

class Range { get [Symbol.toStringTag]() { return "Range"; } }

let r = new Range(1, 10);

Object.prototype.toString.call(r) // => "[object Range]" classof(r) // => "Range"

1. *Symbol.species* -- Array defines methods concat(), filter(), map(), slice(), and splice(), which return arrays. When we create an array subclass like EZArray that inherits these methods, should the inherited method return instances of Array or instances of EZArray? Good arguments can be made for either choice, but the ES6 specification says that (by default) the five array-returning methods will return instances of the subclass.

Here’s how it works:

* In ES6 and later, the Array() constructor has a property with the symbolic name Symbol.species. (Note that this Symbol is used as the name of a property of the constructor function. Most of the other well-known Symbols described here are used as the name of methods of a prototype object.)
* When we create a subclass with extends, the resulting subclass constructor inherits properties from the superclass constructor. (This is in addition to the normal kind of inheritance, where instances of the subclass inherit methods of the superclass.) This means that the constructor for every subclass of Array also has an inherited property with name Symbol.species. (Or a subclass can define its own property with this name, if it wants.)
* Methods like map() and slice() that create and return new arrays are tweaked slightly in ES6 and later. Instead of just creating a regular Array, they (in effect) invoke new this.constructor[Symbol.species]() to create the new array.

Now here’s the interesting part. Suppose that Array[Symbol.species] was just a regular data property, defined like this: Array[Symbol.species] = Array; In that case, then subclass constructors would inherit the Array() constructor as their “species,” and invoking map() on an array subclass would return an instance of the superclass rather than an instance of the subclass. That is not how ES6 actually behaves, however. The reason is that Array[Symbol.species] is a readonly accessor property whose getter function simply returns this. Subclass constructors inherit this getter function, which means that by default, every subclass constructor is its own “species.”

Sometimes this default behavior is not what you want, however. If you wanted the array-returning methods of EZArray to return regular Array objects, you just need to set EZArray[Symbol.species] to Array. But since the inherited property is a read-only accessor, you can’t just set it with an assignment operator. You can use defineProperty(), however:

EZArray[Symbol.species] = Array; // Attempt to set a readonly property fails

Object.defineProperty(EZArray, Symbol.species, {value: Array});

The simplest option is probably to explicitly define your own Symbol.species getter when creating the subclass in the first place:

class EZArray extends Array {

static get [Symbol.species]() { return Array; }

get first() { return this[0]; }

get last() { return this[this.length-1]; } }

let e = new EZArray(1,2,3);

let f = e.map(x => x - 1);

e.last // => 3

f.last // => undefined: f is a regular array with no last getter

1. *Symbol.isConcatSpreadable*  -- The Array method concat() is one of the methods described in the previous section that uses Symbol.species to determine what constructor to use for the returned array. But concat() also uses Symbol.isConcatSpreadable. concat() method of an array treats its this value and its array arguments differently than its nonarray arguments: nonarray arguments are simply appended to the new array, but the this array and any array arguments are flattened or “spread” so that the elements of the array are concatenated rather than the array argument itself.

Before ES6, concat() just used Array.isArray() to determine whether to treat a value as an array or not. In ES6, the algorithm is changed slightly: if the argument (or the this value) to concat() is an object and has a property with the symbolic name Symbol.isConcatSpreadable, then the boolean value of that property is used to determine whether the argument should be “spread.” If no such property exists, then Array.isArray() is used as in previous versions of the language.

There are two cases when you might want to use this Symbol:

* If you create an Array-like object and want it to behave like a real array when passed to concat(), you can simply add the symbolic property to your object:

let arraylike = { length: 1, 0: 1, [Symbol.isConcatSpreadable]: true }; [].concat(arraylike) // => [1]:

* Array subclasses are spreadable by default, so if you are defining an array subclass that you do not want to act like an array when used with concat(), then you can add a getter like this to your subclass:

class NonSpreadableArray extends Array {

get [Symbol.isConcatSpreadable]() { return false; } }

let a = new NonSpreadableArray(1,2,3);

[].concat(a).length // => 1;

1. *Symbol.toPrimitive* -- JavaScript has three slightly different algorithms for converting objects to primitive values. Loosely speaking, for conversions where a string value is expected or preferred, JavaScript invokes an object’s toString() method first and falls back on the valueOf() method if toString() is not defined or does not return a primitive value. For conversions where a numeric value is preferred, JavaScript tries the valueOf() method first and falls back on toString() if valueOf() is not defined or if it does not return a primitive value. And finally, in cases where there is no preference, it lets the class decide how to do the conversion. Date objects convert using toString() first, and all other types try valueOf() first.

In ES6, the well-known Symbol Symbol.toPrimitive allows you to override this default object-to-primitive behavior and gives you complete control over how instances of your own classes will be converted to primitive values. To do this, define a method with this symbolic name. The method must return a primitive value that somehow represents the object. The method you define will be invoked with a single string argument that tells you what kind of conversion JavaScript is trying to do on your object:

* If the argument is "string", it means that JavaScript is doing the conversion in a context where it would expect or prefer (but not require) a string. This happens when you interpolate the object into a template literal, for example.
* If the argument is "number", it means that JavaScript is doing the conversion in a context where it would expect or prefer (but not require) a numeric value. This happens when you use the object with a < or > operator or with arithmetic operators like - and \*.
* If the argument is "default", it means that JavaScript is converting your object in a context where either a numeric or string value could work. This happens with the +, ==, and != operators.

Many classes can ignore the argument and simply return the same primitive value in all cases. If you want instances of your class to be comparable and sortable with < and >, then that is a good reason to define a [Symbol.toPrimitive] method.

1. *Template Tags* -- When a function expression is followed by a template literal, the function is invoked. The first argument is an array of strings, and this is followed by zero or more additional arguments, which can have values of any type. The number of arguments depends on the number of values that are interpolated into the template literal. If the template literal is simply a constant string with no interpolations, then the tag function will be called with an array of that one string and no additional arguments. If the template literal includes one interpolated value, then the tag function is called with two arguments. The first is an array of two strings, and the second is the interpolated value. The strings in that initial array are the string to the left of the interpolated value and the string to its right, and either one of them may be the empty string. If the template literal includes two interpolated values, then the tag function is invoked with three arguments: an array of three strings and the two interpolated values. The three strings (any or all of which may be empty) are the text to the left of the first value, the text between the two values, and the text to the right of the second value.
2. *Reflect.ownKeys(o)* -- This function returns an array of the names of the properties of the object o or throws a TypeError if o is not an object. The names in the returned array will be strings and/or symbols. Calling this function is similar to calling Object.getOwnPropertyNames() and Object.getOwnPropertySymbols() and combining their results.
3. *Proxy Objects* -- The Proxy class, available in ES6 and later, is JavaScript’s most powerful metaprogramming feature. It allows us to write code that alters the fundamental behavior of JavaScript objects. What the Proxy class does is allows us a way to implement those fundamental operations ourselves and create objects that behave in ways that are not possible for ordinary objects.

When we create a Proxy object, we specify two other objects, the target object and the handlers object:

let proxy = new Proxy(target, handlers);

The resulting Proxy object has no state or behavior of its own. Whenever you perform an operation on it (read a property, write a property, define a new property, look up the prototype, invoke it as a function), it dispatches those operations to the handlers object or to the target object. The operations supported by Proxy objects are the same as those defined by the Reflect API (apply, construct, defineProperty, deleteProperty, get, getOwnPropertyDescriptor, getPrototypeOf, has, isExtensible, ownKeys, preventExtensions, set, setPrototypeOf).

Suppose that p is a Proxy object and you write delete p.x, it looks for a deleteProperty() method on the handlers object. If such a method exists, it invokes it. And if no such method exists, then the Proxy object performs the property deletion on the target object instead. Proxies work this way for all of the fundamental operations: if an appropriate method exists on the handlers object, it invokes that method to perform the operation.

This means that a Proxy can obtain its behavior from the target object or from the handlers object. If the handlers object is empty, then the proxy is essentially a transparent wrapper around the target object:

let t = { x: 1, y: 2 };

let p = new Proxy(t, {});

p.x // => 1

delete p.y // => true: delete property y of the proxy

t.y // => undefined: this deletes it in the target, too

p.z = 3; // Defining a new property on the proxy

t.z // => 3: defines the property on the target

This kind of transparent wrapper proxy is essentially equivalent to the underlying target object, which means that there really isn’t a reason to use it instead of the wrapped object. Transparent wrappers can be useful, however, when created as “revocable proxies.” Instead of creating a Proxy with the Proxy() constructor, you can use the Proxy.revocable() factory function. This function returns an object that includes a Proxy object and also a revoke() function. Once you call the revoke() function, the proxy immediately stops working:

function accessTheDatabase() { /\* implementation omitted \*/ return 42; }

let {proxy, revoke} = Proxy.revocable(accessTheDatabase, {});

proxy() // => 42: The proxy gives access to the underlying target function revoke(); // But that access can be turned off whenever we want

proxy(); // !TypeError: we can no longer call this function

Note that in addition to demonstrating revocable proxies, the preceding code also demonstrates that proxies can work with target functions as well as target objects. But the main point here is that revocable proxies are a building block for a kind of code isolation, and you might use them when dealing with untrusted third-party libraries, for example. If you have to pass a function to a library that you don’t control, you can pass a revocable proxy instead and then revoke the proxy when you are finished with the library. This prevents the library from keeping a reference to your function and calling it at unexpected times. This kind of defensive programming is not typical in JavaScript programs, but the Proxy class at least makes it possible.

Example : an object that appears to have an infinite number of read-only properties, where the value of each property is the same as the name of the prop

let identity = new Proxy(

  {},

  {

    get(o, name, target) { return name; },

    has(o, name) { return true; },

set(o, name, value, target) { return false; },

    deleteProperty(o, name) { return false; },

    defineProperty(o, name, desc) { return false; },

    isExtensible(o) { return false; },

    getPrototypeOf(o) { return null; },

    setPrototypeOf(o, proto) { return false; },

    ownKeys(o) {

throw new RangeError("Infinite number of properties"); },

    getOwnPropertyDescriptor(o, name) {

return { value: name, enumerable: false,

writable: false, configurable: false, }; },

  }

);

identity.x; // => "x"

identity.toString; // => "toString"

identity[0]; // => "0"

identity.x = 1; // no effect

identity.x; // => "x"

delete identity.x; // => false: can't delete

identity.x; // => "x"

Object.keys(identity); // !RangeError

Proxy objects can derive their behavior from the target object and from the handlers object, and the examples we have seen so far have used one object or the other. But it is typically more useful to define proxies that use both objects.

for example, a Proxy to create a read-only wrapper for a target object. When code tries to read values from the object, those reads are forwarded to the target object normally. But if any code tries to modify the object or its properties, methods of the handler object throw a TypeError.

Another technique when writing proxies is to define handler methods that intercept operations on an object but still delegate the operations to the target object. The functions of the Reflect API have exactly the same signatures as the handler methods, so they make it easy to do that kind of delegation.

The Proxy handler API allows us to define objects with major inconsistencies, however, and in this case, the Proxy class itself will prevent us from creating Proxy objects that are inconsistent in a bad way. At the start of this section, we described proxies as objects with no behavior of their own because they simply forward all operations to the handlers object and the target object. But this is not entirely true: after forwarding an operation, the Proxy class performs some sanity checks on the result to ensure important JavaScript invariants are not being violated. If it detects a violation, the proxy will throw a TypeError instead of letting the operation proceed. As an example, if you create a proxy for a non-extensible object, the proxy will throw a TypeError if the isExtensible() handler ever returns true:

let target = Object.preventExtensions({});

let proxy = new Proxy(target, { isExtensible() { return true; }}); Reflect.isExtensible(proxy); // !TypeError: invariant violation

Relatedly, proxy objects for non-extensible targets may not have a getPrototypeOf() handler that returns anything other than the real prototype object of the target. Also, if the target object has nonwritable, nonconfigurable properties, then the Proxy class will throw a TypeError if the get() handler returns anything other than the actual value:

let target = Object.freeze({x: 1});

let proxy = new Proxy(target, { get() { return 99; }});

proxy.x; // !TypeError: value returned by get() doesn't match target

Proxy enforces a number of additional invariants, almost all of them having to do with non-extensible target objects and nonconfigurable properties on the target object.