Chapter 4. Expressions and Operators

This chapter documents JavaScript expressions and the operators with which many of those expressions are built. An *expression* is a phrase of JavaScript that can be *evaluated* to produce a value. A constant embedded literally in your program is a very simple kind of expression. A variable name is also a simple expression that evaluates to whatever value has been assigned to that variable. Complex expressions are built from simpler expressions. An array access expression, for example, consists of one expression that evaluates to an array followed by an open square bracket, an expression that evaluates to an integer, and a close square bracket. This new, more complex expression evaluates to the value stored at the specified index of the specified array. Similarly, a function invocation expression consists of one expression that evaluates to a function object and zero or more additional expressions that are used as the arguments to the function.

The most common way to build a complex expression out of simpler expressions is with an *operator*. An operator combines the values of its operands (usually two of them) in some way and evaluates to a new value. The multiplication operator `*` is a simple example. The expression `x * y` evaluates to the product of the values of the expressions `x` and `y`. For simplicity, we sometimes say that an operator *returns* a value rather than "evaluates to" a value.

This chapter documents all of JavaScript's operators, and it also explains expressions (such as array indexing and function invocation) that do not use operators. If you already know another programming language that uses C-style syntax, you'll find that the syntax of most of JavaScript's expressions and operators is already familiar to you.

4.1 Primary Expressions

The simplest expressions, known as *primary expressions*, are those that stand alone—they do not include any simpler expressions. Primary expressions in JavaScript are constant or *literal* values, certain language keywords, and variable references.

Literals are constant values that are embedded directly in your program. They look like these:

```
1.23         // A number literal
"hello"      // A string literal
/pattern/    // A regular expression literal
```

JavaScript syntax for number literals was covered in §3.2. String literals were documented in §3.3. The regular expression literal syntax was introduced in §3.3.5 and will be documented in detail in §11.3.

Some of JavaScript's reserved words are primary expressions:

```
true       // Evalutes to the boolean true value
false      // Evaluates to the boolean false value
null       // Evaluates to the null value
this       // Evaluates to the "current" object
```

We learned about `true`, `false`, and `null` in §3.4 and §3.5. Unlike the other keywords, `this` is not a constant—it evaluates to different values in different places in the program. The `this` keyword is used in object-oriented programming. Within the body of a method, `this` evaluates to the object on which the method was invoked. See §4.5, Chapter 8 (especially §8.2.2), and Chapter 9 for more on `this`.

Finally, the third type of primary expression is a reference to a variable, constant, or property of the global object:

```
i             // Evaluates to the value of the variable i.
sum           // Evaluates to the value of the variable sum.
undefined     // The value of the "undefined" property of the global object
```

When any identifier appears by itself in a program, JavaScript assumes it is a variable or constant or property of the global object and looks up its value. If no variable with that name exists, an attempt to evaluate a nonexistent variable throws a ReferenceError instead.

4.2 Object and Array Initializers

Object and *array initializers* are expressions whose value is a newly created object or array. These initializer expressions are sometimes called *object literals* and *array literals*. Unlike true literals, however, they are not primary expressions, because they include a number of subexpressions that specify property and element values. Array initializers have a slightly simpler syntax, and we'll begin with those.

An array initializer is a comma-separated list of expressions contained within square brackets. The value of an array initializer is a newly created array. The elements of this new array are initialized to the values of the comma-separated expressions:

```
[]         // An empty array: no expressions inside brackets means no eleme
[1+2,3+4]  // A 2-element array.  First element is 3, second is 7
```

The element expressions in an array initializer can themselves be array initializers, which means that these expressions can create nested arrays:

```
let matrix = [[1,2,3], [4,5,6], [7,8,9]];
```

The element expressions in an array initializer are evaluated each time the array initializer is evaluated. This means that the value of an array initializer expression may be different each time it is evaluated.

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];
```

A single trailing comma is allowed after the last expression in an array initializer and does not create an undefined element. However, any array access expression for an index after that of the last expression will necessarily evaluate to undefined.

Object initializer expressions are like array initializer expressions, but the square brackets are replaced by curly brackets, and each subexpression is prefixed with a property name and a colon:

```
let p = { x: 2.3, y: -1.2 };  // An object with 2 properties
let q = {};                   // An empty object with no properties
q.x = 2.3; q.y = -1.2;        // Now q has the same properties as p
```

In ES6, object literals have a much more feature-rich syntax (you can find details in §6.10). Object literals can be nested. For example:

```
let rectangle = {
    upperLeft: { x: 2, y: 2 },
    lowerRight: { x: 4, y: 5 }
};
```

We'll see object and array initializers again in Chapters 6 and 7.

4.3 Function Definition Expressions

A *function definition expression* defines a JavaScript function, and the value of such an expression is the newly defined function. In a sense, a function definition expression is a "function literal" in the same way that an object initializer is an "object literal." A function definition expression typically consists of the keyword `function` followed by a comma-separated list of zero or more identifiers (the parameter names) in parentheses and a block of JavaScript code (the function body) in curly braces. For example:

```
// This function returns the square of the value passed to it.
let square = function(x) { return x * x; };
```

A function definition expression can also include a name for the function. Functions can also be defined using a function statement rather than a function expression. And in ES6 and later, function expressions can use a compact new "arrow function" syntax. Complete details on function definition are in Chapter 8.

4.4 Property Access Expressions

A *property access expression* evaluates to the value of an object property or an array element. JavaScript defines two syntaxes for property access:

```
expression . identifier
expression [ expression ]
```

The first style of property access is an expression followed by a period and an identifier. The expression specifies the object, and the identifier specifies the name of the desired property. The second style of property access follows the first expression (the object or array) with another expression in square brackets. This second expression specifies the name of the desired property or the index of the desired array element. Here are some concrete examples:

```
let o = {x: 1, y: {z: 3}}; // An example object
let a = [o, 4, [5, 6]];    // An example array that contains the object
o.x                        // => 1: property x of expression o
o.y.z                      // => 3: property z of expression o.y
o["x"]                     // => 1: property x of object o
a[1]                       // => 4: element at index 1 of expression a
a[2]["1"]                  // => 6: element at index 1 of expression a[2]
a[0].x                     // => 1: property x of expression a[0]
```

With either type of property access expression, the expression before the `.` or `[` is first evaluated. If the value is `null` or `undefined`, the expression throws a TypeError, since these are the two JavaScript values that cannot have properties. If the object expression is followed by a dot and an identifier, the value of the property named by that identifier is looked up and becomes the overall value of the expression. If the object expression is followed by another expression in square brackets, that second expression is evaluated and converted to a string. The overall value of the expression is then the value of the property named by that string. In either case, if the named property does not exist, then the value of the property access expression is `undefined`.

The *.identifier* syntax is the simpler of the two property access options, but notice that it can only be used when the property you want to access has a name that is a legal identifier, and when you know the name when you write the program. If the property name includes spaces or punctuation characters, or when it is a number (for arrays), you must use the square bracket notation. Square brackets are also used when the property name is not static but is itself the result of a computation (see §6.3.1 for an example).

Objects and their properties are covered in detail in Chapter 6, and arrays and their elements are covered in Chapter 7.

4.4.1 Conditional Property Access

ES2020 adds two new kinds of property access expressions:

```
expression ?. identifier
expression ?.[ expression ]
```

In JavaScript, the values `null` and `undefined` are the only two values that do not have properties. In a regular property access expression using `.` or `[]`, you get a TypeError if the expression on the left evaluates to `null` or `undefined`. You can use `?.` and `?.[]` syntax to guard against errors of this type.

Consider the expression `a?.b`. If `a` is `null` or `undefined`, then the expression evaluates to `undefined` without any attempt to access the property `b`. If `a` is some other value, then `a?.b` evaluates to whatever `a.b` would evaluate to (and if `a` does not have a property named `b`, then the value will again be `undefined`).

This form of property access expression is sometimes called “optional chaining” because it also works for longer “chained” property access expressions like this one:

```
let a = { b: null };
a.b?.c.d   // => undefined
```

`a` is an object, so `a.b` is a valid property access expression. But the value of `a.b` is `null`, so `a.b.c` would throw a TypeError. By using `?.` instead of `.` we avoid the TypeError, and `a.b?.c` evaluates to `undefined`. This means that `(a.b?.c).d` will throw a TypeError, because that expression attempts to access a property of the value `undefined`. But—and this is a very important part of “optional chaining”—`a.b?.c.d` (without the parentheses) simply evaluates to `undefined` and does not throw an error. This is because 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.

Of course, if `a.b` is an object, and if that object has no property named `c`, then `a.b?.c.d` will again throw a TypeError, and we will want to use another conditional property access:

```
let a = { b: {} };
a.b?.c?.d  // => undefined
```

Conditional property access is also possible using `?.[]` instead of `[]`. In the expression `a?.[b][c]`, if the value of `a` is `null` or `undefined`, then the entire expression immediately evaluates to `undefined`, and subexpressions `b` and `c` are never even evaluated. If either of those expressions has side effects, the side effect will not occur if `a` is not defined:

```
let a;          // Oops, we forgot to initialize this variable!
let index = 0;
try {
    a[index++]; // Throws TypeError
} catch(e) {
    index       // => 1: increment occurs before TypeError is thrown
}
a?.[index++]    // => undefined: because a is undefined
index           // => 1: not incremented because ?.[] short-circuits
a[index++]      // !TypeError: can't index undefined.
```

Conditional property access with `?.` and `?.[]` is one of the newest features of JavaScript. As of early 2020, this new syntax is supported in the current or beta versions of most major browsers.

4.5 Invocation Expressions

An *invocation expression* is JavaScript's syntax for calling (or executing) a function or method. It starts with a function expression that identifies the function to be called. The function expression is followed by an open parenthesis, a comma-separated list of zero or more argument expressions, and a close parenthesis. Some examples:

```
f(0)            // f is the function expression; 0 is the argument expressi
Math.max(x,y,z) // Math.max is the function; x, y, and z are the arguments.
a.sort()        // a.sort is the function; there are no arguments.
```

When an invocation expression is evaluated, the function expression is evaluated first, and then the argument expressions are evaluated to produce a list of argument values. If the value of the function expression is not a function, a TypeError is thrown. Next, the argument values are assigned, in order, to the parameter names specified when the function was defined, and then the body of the function is executed. If the function uses a `return` statement to return a value, then that value becomes the value of the invocation expression. Otherwise, the value of the invocation expression is `undefined`. Complete details on function invocation, including an explanation of what happens when the number of argument expressions does not match the number of parameters in the function definition, are in Chapter 8.

Every invocation expression includes a pair of parentheses and an expression before the open parenthesis. If that expression is a property access expression, then the invocation is known as a *method invocation*. In method invocations, the object or array that is the subject of the property access becomes the value of the `this` keyword while the body of the function is being executed. This enables an object-oriented programming paradigm in which functions (which we call "methods" when used this way) operate on the object of which they are part. See Chapter 9 for details.

4.5.1 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.

Array objects have a `sort()` method that can optionally be passed a function argument that defines the desired sorting order for the array elements. Before ES2020, if you wanted to write a method like `sort()` that takes an optional function argument, you would typically use an `if` statement to check that the function argument was defined before invoking it in the body of the `if`:

```
function square(x, log) { // The second argument is an optional function
    if (log) {            // If the optional function is passed
        log(x);           // Invoke it
    }
    return x * x;         // Return the square of the argument
}
```

With this conditional invocation syntax of ES2020, however, you can simply write the function invocation using `?.()`, knowing that invocation will only happen if there is actually a value to be invoked:

```
function square(x, log) { // The second argument is an optional function
    log?.(x);             // Call the function if there is one
    return x * x;         // Return the square of the argument
}
```

Note, however, that `?.()` only checks whether the lefthand side is `null` or `undefined`. It does not verify that the value is actually a function. So the `square()` function in this example would still throw an exception if you passed two numbers to it, for example.

Like conditional property access expressions (§4.4.1), function invocation with `?.()` is short-circuiting: if the value to the left of `?.` is `null` or `undefined`, then none of the argument expressions within the parentheses are evaluated:

```
let f = null, x = 0;
try {
    f(x++); // Throws TypeError because f is null
} catch(e) {
    x       // => 1: x gets incremented before the exception is thrown
}
f?.(x++)    // => undefined: f is null, but no exception thrown
x           // => 1: increment is skipped because of short-circuiting
```

Conditional invocation expressions with `?.()` work just as well for methods as they do for functions. But because method invocation also involves property access, it is worth taking a moment to be sure you understand the differences between the following expressions:

```
o.m()      // Regular property access, regular invocation
o?.m()     // Conditional property access, regular invocation
o.m?.()    // Regular property access, conditional invocation
```

In the first expression, `o` must be an object with a property `m` and the value of that property must be a function. In the second expression, if `o` is `null` or `undefined`, then the expression evaluates to `undefined`. But if `o` has any other value, then it must have a property `m` whose value is a function. And in the third expression, `o` must not be `null` or `undefined`. If it does not have a property `m`, or if the value of that property is `null`, then the entire expression evaluates to `undefined`.

Conditional invocation with `?.()` is one of the newest features of JavaScript. As of the first months of 2020, this new syntax is supported in the current or beta versions of most major browsers.

4.6 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
```

The value of an object creation expression is the newly created object. Constructors are explained in more detail in Chapter 9.

4.7 Operator Overview

Operators are used for JavaScript's arithmetic expressions, comparison expressions, logical expressions, assignment expressions, and more. Table 4-1 summarizes the operators and serves as a convenient reference.

Note that most operators are represented by punctuation characters such as `+` and `=`. Some, however, are represented by keywords such as `delete` and `instanceof`. Keyword operators are regular operators, just like those expressed with punctuation; they simply have a less succinct syntax.

Table 4-1 is organized by operator precedence. The operators listed first have higher precedence than those listed last. Operators separated by a horizontal line have different precedence levels. The column labeled A gives the operator associativity, which can be L (left-to-right) or R (right-to-left), and the column N specifies the number of operands. The column labeled Types lists the expected types of the operands and (after the → symbol) the result type for the operator. The subsections that follow the table explain the concepts of precedence, associativity, and operand type. The operators themselves are individually documented following that discussion.

Table 4-1. JavaScript operators

Operator	Operation	A	N	Types
`++`	Pre- or post-increment	R	1	lval→num
`--`	Pre- or post-decrement	R	1	lval→num
`-`	Negate number	R	1	num→num
`+`	Convert to number	R	1	any→num
`~`	Invert bits	R	1	int→int
`!`	Invert boolean value	R	1	bool→bool
`delete`	Remove a property	R	1	lval→bool
`typeof`	Determine type of operand	R	1	any→str
`void`	Return undefined value	R	1	any→undef
`**`	Exponentiate	R	2	num,num→num
`*`, `/`, `%`	Multiply, divide, remainder	L	2	num,num→num
`+`, `-`	Add, subtract	L	2	num,num→num
`+`	Concatenate strings	L	2	str,str→str
`<<`	Shift left	L	2	int,int→int
`>>`	Shift right with sign extension	L	2	int,int→int

Operator	Operation	A	N	Types
`>>>`	Shift right with zero extension	L	2	int,int→int
`<`, `<=`, `>`, `>=`	Compare in numeric order	L	2	num,num→bool
`<`, `<=`, `>`, `>=`	Compare in alphabetical order	L	2	str,str→bool
`instanceof`	Test object class	L	2	obj,func→bool
`in`	Test whether property exists	L	2	any,obj→bool
`==`	Test for non-strict equality	L	2	any,any→bool
`!=`	Test for non-strict inequality	L	2	any,any→bool
`===`	Test for strict equality	L	2	any,any→bool
`!==`	Test for strict inequality	L	2	any,any→bool
`&`	Compute bitwise AND	L	2	int,int→int
`^`	Compute bitwise XOR	L	2	int,int→int
`|`	Compute bitwise OR	L	2	int,int→int
`&&`	Compute logical AND	L	2	any,any→any

Operator	Operation	A	N	Types
`||`	Compute logical OR	L	2	any,any→any
`??`	Choose 1st defined operand	L	2	any,any→any
`?:`	Choose 2nd or 3rd operand	R	3	bool,any,any→any
`=`	Assign to a variable or property	R	2	lval,any→any
`**=`, `*=`, `/=`, `%=`, `+=`, `-=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `>>>=`	Operate and assign	R	2	lval,any→any
`,`	Discard 1st operand, return 2nd	L	2	any,any→any

4.7.1 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.

4.7.2 Operand and Result Type

Some operators work on values of any type, but most expect their operands to be of a specific type, and most operators return (or evaluate to) a value of a specific type. The Types column in Table 4-1 specifies operand types (before the arrow) and result type (after the arrow) for the operators.

JavaScript operators usually convert the type (see §3.9) of their operands as needed. The multiplication operator `*` expects numeric operands, but the expression `"3" * "5"` is legal because JavaScript can convert the operands to numbers. The value of this expression is the number 15, not the string "15", of course. Remember also that every JavaScript value is either "truthy" or "falsy," so operators that expect boolean operands will work with an operand of any type.

Some operators behave differently depending on the type of the operands used with them. Most notably, the `+` operator adds numeric operands but concatenates string operands. Similarly, the comparison operators such as `<` perform comparison in numerical or alphabetical order depending on the type of the operands. The descriptions of individual operators explain their type-dependencies and specify what type conversions they perform.

Notice that the assignment operators and a few of the other operators listed in Table 4-1 expect an operand of type `lval`. *lvalue* is a historical term that means "an expression that can legally appear on the left side of an assignment expression." In JavaScript, variables, properties of objects, and elements of arrays are lvalues.

4.7.3 Operator Side Effects

Evaluating a simple expression like `2 * 3` never affects the state of your program, and any future computation your program performs will be unaffected by that evaluation. Some expressions, however, have *side effects*, and their evaluation may affect the result of future evaluations. The assignment operators are the most obvious example: if you assign a value to a variable or property, that changes the value of any expression that uses that variable or property. The `++` and `--` increment and decrement operators are similar, since they perform an implicit assignment. The `delete` operator also has side effects: deleting a property is like (but not the same as) assigning `undefined` to the property.

No other JavaScript operators have side effects, but function invocation and object creation expressions will have side effects if any of the operators used in the function or constructor body have side effects.

4.7.4 Operator Precedence

The operators listed in Table 4-1 are arranged in order from high precedence to low precedence, with horizontal lines separating groups of operators at the same precedence level. Operator precedence controls the order in which operations are performed. Operators with higher precedence (nearer the top of the table) are performed before those with lower precedence (nearer to the bottom).

Consider the following expression:

```
w = x + y*z;
```

The multiplication operator `*` has a higher precedence than the addition operator `+`, so the multiplication is performed before the addition. Furthermore, the assignment operator `=` has the lowest precedence, so the assignment is performed after all the operations on the right side are completed.

Operator precedence can be overridden with the explicit use of parentheses. To force the addition in the previous example to be performed first, write:

```
w = (x + y)*z;
```

Note that property access and invocation expressions have higher precedence than any of the operators listed in Table 4-1. Consider this expression:

```
// my is an object with a property named functions whose value is an
// array of functions. We invoke function number x, passing it argument
// y, and then we ask for the type of the value returned.
typeof my.functions[x](y)
```

Although `typeof` is one of the highest-priority operators, the `typeof` operation is performed on the result of the property access, array index, and function invocation, all of which have higher priority than operators.

In practice, if you are at all unsure about the precedence of your operators, the simplest thing to do is to use parentheses to make the evaluation order explicit. The rules that are important to know are these: multiplication and division are performed before addition and subtraction, and assignment has very low precedence and is almost always performed last.

When new operators are added to JavaScript, they do not always fit naturally into this precedence scheme. The `??` operator (§4.13.2) 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.

4.7.5 Operator Associativity

In Table 4-1, the column labeled A specifies the *associativity* of the operator. A value of L specifies left-to-right associativity, and a value of R specifies right-to-left associativity. The associativity of an operator specifies the order in which operations of the same precedence are performed. Left-to-right associativity means that operations are performed from left to right. For example, the subtraction operator has left-to-right associativity, so:

```
w = x - y - z;
```

is the same as:

```
w = ((x - y) - z);
```

On the other hand, the following expressions:

```
y = a ** b ** c;
x = ~-y;
w = x = y = z;
q = a?b:c?d:e?f:g;
```

are equivalent to:

```
y = (a ** (b ** c));
x = ~(-y);
w = (x = (y = z));
q = a?b:(c?d:(e?f:g));
```

because the exponentiation, unary, assignment, and ternary conditional operators have right-to-left associativity.

4.7.6 Order of Evaluation

Operator precedence and associativity specify the order in which operations are performed in a complex expression, but they do not specify the order in which the subexpressions are evaluated. JavaScript always evaluates expressions in strictly left-to-right order. In the expression `w = x + y * z`, for example, the subexpression `w` is evaluated first, followed by `x`, `y`, and `z`. Then the values of `y` and `z` are multiplied, added to the value of `x`, and assigned to the variable or property specified by expression `w`. Adding parentheses to the expressions can change the relative order of the multiplication, addition, and assignment, but not the left-to-right order of evaluation.

Order of evaluation only makes a difference if any of the expressions being evaluated has side effects that affect the value of another expression. If expression `x` increments a variable that is used by expression `z`, then the fact that `x` is evaluated before `z` is important.

4.8 Arithmetic Expressions

This section covers the operators that perform arithmetic or other numerical manipulations on their operands. The exponentiation, multiplication, division, and subtraction operators are straightforward and are covered first. The addition operator gets a subsection of its own because it can also perform string concatenation and has some unusual type conversion rules. The unary operators and the bitwise operators are also covered in subsections of their own.

Most of these arithmetic operators (except as noted as follows) can be used with BigInt (see §3.2.5) operands or with regular numbers, as long as you don't mix the two types.

The basic arithmetic operators are `**` (exponentiation), `*` (multiplication), `/` (division), `%` (modulo: remainder after division), `+` (addition), and `-` (subtraction). As noted, we'll discuss the `+` operator in a section of its own. The other five basic operators simply evaluate their operands, convert the values to numbers if necessary, and then compute the power, product, quotient, remainder, or difference. Non-numeric operands that cannot convert to numbers convert to the `NaN` value. If either operand is (or converts to) `NaN`, the result of the operation is (almost always) `NaN`.

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. `**` is JavaScript's newest arithmetic operator: it was added to the language with ES2016. The `Math.pow()` function has been available since the earliest versions of JavaScript, however, and it performs exactly the same operation as the `**` operator.

The `/` operator divides its first operand by its second. If you are used to programming languages that distinguish between integer and floating-point 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.

The `%` operator computes the first operand modulo the second operand. In other words, it returns the remainder after whole-number division of the first operand by the second operand. The sign of the result is the same as the sign of the first operand. For example, `5 % 2` evaluates to `1`, and `-5 % 2` evaluates to `-1`.

While the modulo operator is typically used with integer operands, it also works for floating-point values. For example, `6.5 % 2.1` evaluates to `0.2`.

4.8.1 The + Operator

The binary `+` operator adds numeric operands or concatenates string operands:

```
1 + 2                        // => 3
"hello" + " " + "there"      // => "hello there"
"1" + "2"                    // => "12"
```

When the values of both operands are numbers, or are both strings, then it is obvious what the `+` operator does. In any other case, however, type conversion is necessary, and the operation to be performed depends on the conversion performed. The conversion rules for `+` give priority to string concatenation: if either of the operands is a string or an object that converts to a string, the other operand is converted to a string and concatenation is performed. Addition is performed only if neither operand is string-like.

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 described in §3.9.3. 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]": concatenation after object-to-strir
```

```
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. In the second line, parentheses alter this order of operations: the number 2 is concatenated with the string to produce a new string. Then the number 1 is concatenated with the new string to produce the final result.

4.8.2 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. When used before the operand, where it is known as the pre-increment operator, it increments the operand and evaluates to the incremented value of that operand. When used after the operand, where it is known as the post-increment operator, 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.

This operator, in both its pre- and post-increment forms, is most commonly used to increment a counter that controls a `for` loop (§5.4.3).

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.

4.8.3 Bitwise Operators

The bitwise operators perform low-level manipulation of the bits in the binary representation of numbers. Although they do not perform traditional arithmetic operations, they are categorized as arithmetic operators here because they operate on numeric operands and return a numeric value. Four of these operators perform Boolean algebra on the individual bits of the operands, behaving as if each bit in each operand were a boolean value (1=true, 0=false). The other three bitwise operators are used to shift bits left and right. These operators are not commonly used in JavaScript programming, and if you are not familiar with the binary representation of integers, including the two's complement representation of negative integers, you can probably skip this section.

The bitwise operators expect integer operands and behave as if those values were represented as 32-bit integers rather than 64-bit floating-point values. These operators convert their operands to numbers, if necessary, and then coerce the numeric values to 32-bit integers by dropping any fractional part and any bits beyond the 32nd. The shift operators require a right-side operand between 0 and 31. After converting this operand to an unsigned 32-bit integer, they drop any bits beyond the 5th, which yields a number in the appropriate range. Surprisingly, `NaN`, `Infinity`, and `-Infinity` all convert to 0 when used as operands of these bitwise operators.

All of these bitwise operators except `>>>` can be used with regular number operands or with BigInt (see §3.2.5) operands.

Bitwise AND (`&`)

The `&` operator performs a Boolean AND operation on each bit of its integer arguments. A bit is set in the result only if the corresponding bit is set in both operands. For example, `0x1234 & 0x00FF` evaluates to `0x0034`.

Bitwise OR (`|`)

The `|` operator performs a Boolean OR operation on each bit of its integer arguments. A bit is set in the result if the corresponding bit is set in one or both of the operands. For example, `0x1234 | 0x00FF` evaluates to `0x12FF`.

Bitwise XOR (`^`)

The `^` operator performs a Boolean exclusive OR operation on each bit of its integer arguments. Exclusive OR means that either operand one is `true` or operand two is `true`, but not both. A bit is set in this operation's result if a corresponding bit is set in one (but not both) of the two operands. For example, `0xFF00 ^ 0xF0F0` evaluates to `0x0FF0`.

Bitwise NOT (`~`)

The `~` operator is a unary operator that appears before its single integer operand. It operates by reversing all bits in the operand. Because of the way signed integers are represented in JavaScript, applying the `~` operator to a value is equivalent to changing its sign and subtracting 1. For example, `~0x0F` evaluates to `0xFFFFFFF0`, or −16.

Shift left (`<<`)

The `<<` operator moves all bits in its first operand to the left by the number of places specified in the second operand, which should be an integer between 0 and 31. For example, in the operation `a << 1`, the first bit (the ones bit) of `a` becomes the second bit (the twos bit), the second bit of `a` becomes the third, etc. A zero is used for the new first bit, and the value of the 32nd bit is lost. Shifting a value left by one position is equivalent to multiplying by 2, shifting two positions is equivalent to multiplying by 4, and so on. For example, `7 << 2` evaluates to 28.

Shift right with sign (`>>`)

The `>>` operator moves all bits in its first operand to the right by the number of places specified in the second operand (an integer between 0 and 31). Bits that are shifted off the right are lost. The bits filled in on the left depend on the sign bit of the original operand, in order to preserve the sign of the result. If the first operand is positive, the result has zeros placed in the high bits; if the first operand is negative, the result has ones placed in the high bits. Shifting a positive value right one place is equivalent to dividing by 2 (discarding the remainder), shifting right two places is equivalent to integer division by 4, and so on. `7 >> 1` evaluates to 3, for example, but note that and `−7 >> 1` evaluates to −4.

Shift right with zero fill (`>>>`)

The `>>>` operator is just like the `>>` operator, except that the bits shifted in on the left are always zero, regardless of the sign of the first operand. This is useful when you want to treat signed 32-bit values as if they are unsigned integers. `−1 >> 4` evaluates to −1, but `−1 >>> 4` evaluates to `0x0FFFFFFF`, for example. This is the only one of the JavaScript bitwise operators that cannot be used with BigInt values. BigInt does not represent negative numbers by setting the high bit the way that 32-bit integers do, and this operator only makes sense for that particular two's complement representation.

4.9 Relational Expressions

This section describes JavaScript's relational operators. These operators test for a relationship (such as "equals," "less than," or "property of") between two values and return `true` or `false` depending on whether that relationship exists. Relational expressions always evaluate to a boolean value, and that value is often used to control the flow of program execution in `if`, `while`, and `for` statements (see Chapter 5). The subsections that follow document the equality and inequality operators, the comparison operators, and JavaScript's other two relational operators, `in` and `instanceof`.

4.9.1 Equality and Inequality Operators

The `==` and `===` operators check whether two values are the same, using two different definitions of sameness. Both operators accept operands of any type, and both return `true` if their operands are the same and `false` if they are different. The `===` operator is known as the strict equality operator (or sometimes the identity operator), and it checks whether its two operands are "identical" using a strict definition of sameness. The `==` operator is known as the equality operator; it checks whether its two operands are "equal" using a more relaxed definition of sameness that allows type conversions.

The `!=` and `!==` operators test for the exact opposite of the `==` and `===` operators. The `!=` inequality operator returns `false` if two values are equal to each other according to `==` and returns `true` otherwise. The `!==` operator returns `false` if two values are strictly equal to each other and returns `true` otherwise. As you'll see in §4.10, the `!` operator computes the Boolean NOT operation. This makes it easy to remember that `!=` and `!==` stand for "not equal to" and "not strictly equal to."

THE `=`, `==`, AND `===` OPERATORS

JavaScript supports `=`, `==`, and `===` operators. Be sure you understand the differences between these assignment, equality, and strict equality operators, and be careful to use the correct one when coding! Although it is tempting to read all three operators as "equals," it may help to reduce confusion if you read "gets" or "is assigned" for `=`, "is equal to" for `==`, and "is strictly equal to" for `===`.

The `==` operator is a legacy feature of JavaScript and is widely considered to be a source of bugs. You should almost always use `===` instead of `==`, and `!==` instead of `!=`.

As mentioned in §3.8, JavaScript objects are compared by reference, not by value. An object is equal to itself, but not to any other object. If two distinct objects have the same number of properties, with the same names and values, they are still not equal. Similarly, two arrays that have the same elements in the same order are not equal to each other.

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 (see the sidebar in §3.3) 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.

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 using the algorithm described in §3.9.3 and try the comparison again. 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 different-looking 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`.

4.9.2 Comparison Operators

The comparison operators test the relative order (numerical or alphabetical) of their two operands:

Less than (<)

The `<` operator evaluates to `true` if its first operand is less than its second operand; otherwise, it evaluates to `false`.

Greater than (>)

The `>` operator evaluates to `true` if its first operand is greater than its second operand; otherwise, it evaluates to `false`.

Less than or equal (<=)

The `<=` operator evaluates to `true` if its first operand is less than or equal to its second operand; otherwise, it evaluates to `false`.

Greater than or equal (>=)

The `>=` operator evaluates to `true` if its first operand is greater than or equal to its second operand; otherwise, it evaluates to `false`.

The operands of these comparison operators may be of any type. Comparison can be performed only on numbers and strings, however, so operands that are not numbers or strings are converted.

Comparison and conversion occur as follows:

- If either operand evaluates to an object, that object is converted to a primitive value, as described at the end of §3.9.3; 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.

Remember that JavaScript strings are sequences of 16-bit integer values, and that string comparison is just a numerical comparison of the values in the two strings. The numerical encoding order defined by Unicode may not match the traditional collation order used in any particular language or locale. Note in particular that string comparison is case-sensitive, and all capital ASCII letters are "less than" all lowercase ASCII letters. This rule can cause confusing results if you do not expect it. For example, according to the `<` operator, the string "Zoo" comes before the string "aardvark".

For a more robust string-comparison algorithm, try the `String.localeCompare()` method, which also takes locale-specific definitions of alphabetical order into account. For case-insensitive comparisons, you can convert the strings to all lowercase or all uppercase using `String.toLowerCase()` or `String.toUpperCase()`. And, for a more general and better localized string comparison tool, use the Intl.Collator class described in §11.7.3.

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:

```
1 + 2         // => 3: addition.
"1" + "2"     // => "12": concatenation.
"1" + 2       // => "12": 2 is converted to "2".
11 < 3        // => false: numeric comparison.
"11" < "3"    // => true: string comparison.
"11" < 3      // => false: numeric comparison, "11" converted to 11.
"one" < 3     // => false: numeric comparison, "one" converted to NaN.
```

Finally, note that the `<=` (less than or equal) and `>=` (greater than or equal) operators do not rely on the equality or strict equality operators for determining whether two values are "equal." Instead, the less-than-or-equal operator is simply defined as "not greater than," and the greater-than-or-equal operator is defined as "not less than." The one exception occurs when either operand is (or converts to) `NaN`, in which case, all four comparison operators return `false`.

4.9.3 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
```

4.9.4 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. Chapter 9 explains that, 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`.

In order to understand how the `instanceof` operator works, you must understand the “prototype chain.” This is JavaScript’s inheritance mechanism, and it is described in §6.3.2. To evaluate the expression `o instanceof f`, JavaScript evaluates `f.prototype`, and then looks for that value in the prototype chain of `o`. If it finds it, then `o` is an instance of `f` (or of a subclass of `f`) and the operator returns `true`. If `f.prototype` is not one of the values in the prototype chain of `o`, then `o` is not an instance of `f` and `instanceof` returns `false`.

4.10 Logical Expressions

The logical operators `&&`, `||`, and `!` perform Boolean algebra and are often used in conjunction with the relational operators to combine two relational expressions into one more complex expression. These operators are described in the subsections that follow. In order to fully understand them, you may want to review the concept of "truthy" and "falsy" values introduced in §3.4.

4.10.1 Logical AND (&&)

The `&&` operator can be understood at three different levels. At the simplest level, when used with boolean operands, `&&` performs the Boolean AND operation on the two values: it returns `true` if and only if both its first operand *and* its second operand are `true`. If one or both of these operands is `false`, it returns `false`.

`&&` is often used as a conjunction to join two relational expressions:

```
x === 0 && y === 0   // true if, and only if, x and y are both 0
```

Relational expressions always evaluate to `true` or `false`, so when used like this, the `&&` operator itself returns `true` or `false`. Relational operators have higher precedence than `&&` (and `||`), so expressions like these can safely be written without parentheses.

But `&&` does not require that its operands be boolean values. Recall that all JavaScript values are either "truthy" or "falsy." (See §3.4 for details. The falsy values are `false`, `null`, `undefined`, `0`, `-0`, `NaN`, and `""`. All other values, including all objects, are truthy.) The second level at which `&&` can be understood is as a Boolean AND operator for truthy and falsy values. If both operands are truthy, the operator returns a truthy value. Otherwise, one or both operands must be falsy, and the operator returns a falsy value. In JavaScript, any expression or statement that expects a boolean value will work with a truthy or falsy value, so the fact that `&&` does not always return `true` or `false` does not cause practical problems.

Notice that this description says that the operator returns "a truthy value" or "a falsy value" but does not specify what that value is. For that, we need to describe `&&` at the third and final level. 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
```

It is important to understand that `&&` may or may not evaluate its right-side operand. In this code example, the variable `p` is set to `null`, and the expression `p.x` would, if evaluated, cause a TypeError. But the code uses `&&` in an idiomatic way so that `p.x` is evaluated only if `p` is truthy—not `null` or `undefined`.

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
```

In general, you must be careful whenever you write an expression with side effects (assignments, increments, decrements, or function invocations) on the righthand side of `&&`. Whether those side effects occur depends on the value of the lefthand side.

Despite the somewhat complex way that this operator actually works, it is most commonly used as a simple Boolean algebra operator that works on truthy and falsy values.

4.10.2 Logical OR (||)

The `||` operator performs the Boolean OR operation on its two operands. If one or both operands is truthy, it returns a truthy value. If both operands are falsy, it returns a falsy value.

Although the `||` operator is most often used simply as a Boolean OR operator, it, like the `&&` operator, has more complex behavior. It starts by evaluating its first operand, the expression on its left. If the value of this first operand is truthy, it short-circuits and returns that truthy value without ever evaluating the expression on the right. If, on the other hand, the value of the first operand is falsy, then `||` evaluates its second operand and returns the value of that expression.

As with the `&&` operator, you should avoid right-side operands that include side effects, unless you purposely want to use the fact that the right-side expression may not be evaluated.

An idiomatic usage of this operator is to select the first truthy value in a set of alternatives:

```
// If maxWidth is truthy, use that. Otherwise, look for a value in
// the preferences object. If that is not truthy, use a hardcoded constant.
let max = maxWidth || preferences.maxWidth || 500;
```

Note that if 0 is a legal value for `maxWidth`, then this code will not work correctly, since 0 is a falsy value. See the `??` operator (§4.13.2) for an alternative.

Prior to ES6, this idiom is often used in functions to supply default values for parameters:

```
// Copy the properties of o to p, and return p
function copy(o, p) {
    p = p || {};  // If no object passed for p, use a newly created object.
    // function body goes here
}
```

In ES6 and later, however, this trick is no longer needed because the default parameter value could simply be written in the function definition itself: `function copy(o, p={}) { ... }`.

4.10.3 Logical NOT (!)

The `!` operator is a unary operator; it is placed before a single operand. Its purpose is to invert the boolean value of its operand. For example, if `x` is truthy, `!x` evaluates to `false`. If `x` is falsy, then `!x` is `true`.

Unlike the `&&` and `||` operators, the `!` operator converts its operand to a boolean value (using the rules described in Chapter 3) before inverting the converted value. This means that `!` always returns `true` or `false` and that you can convert any value `x` to its equivalent boolean value by applying this operator twice: `!!x` (see §3.9.2).

As a unary operator, `!` has high precedence and binds tightly. If you want to invert the value of an expression like `p && q`, you need to use parentheses: `!(p && q)`. It is worth noting two laws of Boolean algebra here that we can express using JavaScript syntax:

```
// DeMorgan's Laws
!(p && q) === (!p || !q)  // => true: for all values of p and q
!(p || q) === (!p && !q)  // => true: for all values of p and q
```

4.11 Assignment Expressions

JavaScript uses the `=` operator to assign a value to a variable or property. For example:

```
i = 0;      // Set the variable i to 0.
o.x = 1;    // Set the property x of object o to 1.
```

The `=` operator expects its left-side operand to be an lvalue: a variable or object property (or array element). It expects its right-side operand to be an arbitrary value of any type. The value of an assignment expression is the value of the right-side operand. As a side effect, the `=` operator assigns the value on the right to the variable or property on the left so that future references to the variable or property evaluate to the value.

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
```

If you do this, be sure you are clear on the difference between the `=` and `===` operators! Note that `=` has very low precedence, and parentheses are usually necessary when the value of an assignment is to be used in a larger expression.

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
```

4.11.1 Assignment with Operation

Besides the normal `=` assignment operator, JavaScript supports a number of other assignment operators that provide shortcuts by combining assignment with some other operation. For example, the `+=` operator performs addition and assignment. The following expression:

```
total += salesTax;
```

is equivalent to this one:

```
total = total + salesTax;
```

As you might expect, the `+=` operator works for numbers or strings. For numeric operands, it performs addition and assignment; for string operands, it performs concatenation and assignment.

Similar operators include `-=`, `*=`, `&=`, and so on. Table 4-2 lists them all.

Table 4-2. Assignment operators

Operator	Example	Equivalent
`+=`	`a += b`	`a = a + b`
`-=`	`a -= b`	`a = a - b`
`*=`	`a *= b`	`a = a * b`
`/=`	`a /= b`	`a = a / b`
`%=`	`a %= b`	`a = a % b`
`**=`	`a **= b`	`a = a ** b`
`<<=`	`a <<= b`	`a = a << b`
`>>=`	`a >>= b`	`a = a >> b`
`>>>=`	`a >>>= b`	`a = a >>> b`
`&=`	`a &= b`	`a = a & b`
`|=`	`a |= b`	`a = a | b`
`^=`	`a ^= b`	`a = a ^ b`

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:

```
data[i++] *= 2;
data[i++] = data[i++] * 2;
```

4.12 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
```

Dynamic evaluation of strings of source code is a powerful language feature that is almost never necessary in practice. If you find yourself using `eval()`, you should think carefully about whether you really need to use it. In particular, `eval()` can be a security hole, and you should never pass any string derived from user input to `eval()`. With a language as complicated as JavaScript, there is no way to sanitize user input to make it safe to use with `eval()`. Because of these security issues, some web servers use the HTTP "Content-Security-Policy" header to disable `eval()` for an entire website.

The subsections that follow explain the basic use of `eval()` and explain two restricted versions of it that have less impact on the optimizer.

IS EVAL() A FUNCTION OR AN OPERATOR?

`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. This issue could have been avoided if `eval()` was an operator (and a reserved word). We'll learn (in §4.12.2 and §4.12.3) about restrictions placed on `eval()` to make it more operator-like.

4.12.1 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.

Note that the string of code you pass to `eval()` must make syntactic sense on its own: you cannot use it to paste code fragments into a function. It makes no sense to write `eval("return;")`, for example, because `return` is only legal within functions, and the fact that the evaluated string uses the same variable environment as the calling function does not make it part of that function. If your string would make sense as a standalone script (even a very short one like `x=0`), it is legal to pass to `eval()`. Otherwise, `eval()` will throw a SyntaxError.

4.12.2 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 variable changed: prints "localchanged global
console.log(g(), y); // Global variable changed: prints "local globalchange
```

Notice that the ability to do a global eval is not just an accommodation to the needs of the optimizer; it is actually a tremendously useful feature that allows you to execute strings of code as if they were independent, top-level scripts. As noted at the beginning of this section, it is rare to truly need to evaluate a string of code. But if you do find it necessary, you are more likely to want to do a global eval than a local eval.

4.12.3 Strict eval()

Strict mode (see §5.6.3) imposes further restrictions on the behavior of the `eval()` function and even on the use of the identifier "eval". When `eval()` is called from strict-mode code, or when the string of code to be evaluated itself begins with a "use strict" directive, then `eval()` does a local eval with a private variable environment. This means that in strict mode, evaluated code can query and set local variables, but it cannot define new variables or functions in the local scope.

Furthermore, strict mode makes `eval()` even more operator-like by effectively making "eval" into a reserved word. You are not allowed to overwrite the `eval()` function with a new value. And you are not allowed to declare a variable, function, function parameter, or catch block parameter with the name "eval".

4.13 Miscellaneous Operators

JavaScript supports a number of other miscellaneous operators, described in the following sections.

4.13.1 The Conditional Operator (?:)

The conditional operator is the only ternary operator (three operands) in JavaScript and is sometimes actually called the *ternary operator*. This operator is sometimes written `?:`, although it does not appear quite that way in code. Because this operator has three operands, the first goes before the `?`, the second goes between the `?` and the `:`, and the third goes after the `:`. It is used like this:

```
x > 0 ? x : -x     // The absolute value of x
```

The operands of the conditional operator may be of any type. The first operand is evaluated and interpreted as a boolean. If the value of the first operand is truthy, then the second operand is evaluated, and its value is returned. Otherwise, if the first operand is falsy, then the third operand is evaluated and its value is returned. Only one of the second and third operands is evaluated; never both.

While you can achieve similar results using the `if` statement (§5.3.1), the `?:` operator often provides a handy shortcut. Here is a typical usage, which checks to be sure that a variable is defined (and has a meaningful, truthy value) and uses it if so or provides a default value if not:

```
greeting = "hello " + (username ? username : "there");
```

This is equivalent to, but more compact than, the following `if` statement:

```
greeting = "hello ";
if (username) {
    greeting += username;
} else {
    greeting += "there";
}
```

4.13.2 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
```

`??` is a useful alternative to `||` (§4.10.2) when you want to select the first *defined* operand rather than the first truthy operand. Although `||` is nominally a logical OR operator, it is also used idiomatically to select the first non-falsy operand with code like this:

```
// If maxWidth is truthy, use that. Otherwise, look for a value in
// the preferences object. If that is not truthy, use a hardcoded constant.
let max = maxWidth || preferences.maxWidth || 500;
```

The problem with this idiomatic use is that zero, the empty string, and `false` are all falsy values that may be perfectly valid in some circumstances. In this code example, if `maxWidth` is zero, that value will be ignored. But if we change the `||` operator to `??`, we end up with an expression where zero is a valid value:

```
// If maxWidth is defined, use that. Otherwise, look for a value in
// the preferences object. If that is not defined, use a hardcoded constant
let max = maxWidth ?? preferences.maxWidth ?? 500;
```

Here are more examples showing how `??` works when the first operand is falsy. If that operand is falsy but defined, then `??` returns it. It is only when the first operand is “nullish” (i.e., `null` or `undefined`) that this operator evaluates and returns the second operand:

```
let options = { timeout: 0, title: "", verbose: false, n: null };
options.timeout ?? 1000     // => 0: as defined in the object
options.title ?? "Untitled" // => "": as defined in the object
options.verbose ?? true     // => false: as defined in the object
options.quiet ?? false       // => false: property is not defined
options.n ?? 10              // => 10: property is null
```

Note that the `timeout`, `title`, and `verbose` expressions here would have different values if we used `||` instead of `??`.

The `??` operator is similar to the `&&` and `||` operators but does not have higher precedence or lower precedence than they do. If you use it in an expression with either of those operators, you must use explicit parentheses to specify which operation you want to perform first:

```
(a ?? b) || c    // ?? first, then ||
a ?? (b || c)    // || first, then ??
a ?? b || c      // SyntaxError: parentheses are required
```

The `??` operator is defined by ES2020, and as of early 2020, is newly supported by current or beta versions of all major browsers. This operator is formally called the "nullish coalescing" operator, but I avoid that term because this operator selects one of its operands but does not "coalesce" them in any way that I can see.

4.13.3 The typeof Operator

`typeof` is a unary operator that is placed before its single operand, which can be of any type. Its value is a string that specifies the type of the operand. Table 4-3 specifies the value of the `typeof` operator for any JavaScript value.

Table 4-3. Values returned by the typeof operator

x	typeof x
`undefined`	`"undefined"`
`null`	`"object"`
`true` or `false`	`"boolean"`
any number or `NaN`	`"number"`
any BigInt	`"bigint"`
any string	`"string"`
any symbol	`"symbol"`
any function	`"function"`
any nonfunction object	`"object"`

You might use the `typeof` operator in an expression like this:

```
// If the value is a string, wrap it in quotes, otherwise, convert
(typeof value === "string") ? "'" + value + "'" : value.toString()
```

Note that `typeof` returns "object" if the operand value is `null`. If you want to distinguish `null` from objects, you'll have to explicitly test for this special-case value.

Although JavaScript functions are a kind of object, the `typeof` operator considers functions to be sufficiently different that they have their own return value.

Because `typeof` evaluates to "object" for all object and array values other than functions, it is useful only to distinguish objects from other, primitive types. In order to distinguish one class of object from another, you must use other techniques, such as the `instanceof` operator (see

§4.9.4), the `class` attribute (see §14.4.3), or the `constructor` property (see §9.2.2 and §14.3).

4.13.4 The delete Operator

`delete` is a unary operator that attempts to delete the object property or array element specified as its operand. Like the assignment, increment, and decrement operators, `delete` is typically used for its property deletion side effect and not for the value it returns. Some examples:

```
let o = { x: 1, y: 2}; // Start with an object
delete o.x;            // Delete one of its properties
"x" in o               // => false: the property does not exist anymore

let a = [1,2,3];       // Start with an array
delete a[2];           // Delete the last element of the array
2 in a                 // => false: array element 2 doesn't exist anymore
a.length               // => 3: note that array length doesn't change, thou
```

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 (§4.9.3). Deleting an array element leaves a “hole” in the array and does not change the array’s length. The resulting array is *sparse* (§7.3).

`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 (§14.1) 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 (§4.4). 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.
// Can't delete a variable; returns false, or SyntaxError in strict mode.
delete o;
// Undeletable property: returns false, or TypeError in strict mode.
delete Object.prototype;
```

We'll see the `delete` operator again in §6.4.

4.13.5 The await Operator

`await` was introduced in ES2017 as a way to make asynchronous programming more natural in JavaScript. You will need to read Chapter 13 to understand this operator. Briefly, however, `await` expects a Promise object (representing an asynchronous computation) as its sole operand, and it makes your program behave as if it were waiting for the asynchronous computation to complete (but it does this without actually blocking, and it does not prevent other asynchronous operations from proceeding at the same time). The value of the `await` operator is the fulfillment value of the Promise object. Importantly, `await` is only legal within functions that have been declared asynchronous with the `async` keyword. Again, see Chapter 13 for full details.

4.13.6 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 (see §8.1.3) 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
```

4.13.7 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 (§5.4.3) that has multiple loop variables:

```
// The first comma below is part of the syntax of the let statement
// The second comma is the comma operator: it lets us squeeze 2
// expressions (i++ and j--) into a statement (the for loop) that expects 1
for(let i=0,j=10; i < j; i++,j--) {
    console.log(i+j);
}
```

4.14 Summary

This chapter covers a wide variety of topics, and there is lots of reference material here that you may want to reread in the future as you continue to learn JavaScript. Some key points to remember, however, are these:

- Expressions are the phrases of a JavaScript program.
- Any expression can be *evaluated* to a JavaScript value.
- Expressions can also have side effects (such as variable assignment) in addition to producing a value.
- Simple expressions such as literals, variable references, and property accesses can be combined with operators to produce larger expressions.
- JavaScript defines operators for arithmetic, comparisons, Boolean logic, assignment, and bit manipulation, along with some miscellaneous operators, including the ternary conditional operator.
- The JavaScript `+` operator is used to both add numbers and concatenate strings.
- The logical operators `&&` and `||` have special "short-circuiting" behavior and sometimes only evaluate one of their arguments. Common JavaScript idioms require you to understand the special behavior of these operators.