EXPRESSIONS AND OPERATORS

This article is in need of a technical review.

This chapter describes JavaScript expressions and operators, including assignment, comparison, arithmetic, bitwise, logical, string, and special operators.

Expressions

An *expression* is any valid unit of code that resolves to a value.

Conceptually, there are two types of expressions: those that assign a value to a variable and those that simply have a value.

The expression `x = 7` is an example of the first type. This expression uses the = *operator* to assign the value seven to the variable `x`. The expression itself evaluates to seven.

The code `3 + 4` is an example of the second expression type. This expression uses the + operator to add three and four together without assigning the result, seven, to a variable.

JavaScript has the following expression categories:

- Arithmetic: evaluates to a number, for example 3.14159. (Generally uses arithmetic operators.)
- String: evaluates to a character string, for example, "Fred" or "234". (Generally uses string operators.)
- Logical: evaluates to true or false. (Often involves logical operators.)
- Object: evaluates to an object. (See special operators for various ones that evaluate to objects.)

Operators

JavaScript has the following types of operators. This section describes the operators and contains information about operator precedence.

- Assignment operators
- Comparison operators
- Arithmetic operators
- Bitwise operators
- Logical operators
- String operators
- Special operators

JavaScript has both *binary* and *unary* operators, and one special ternary operator, the conditional operator. A binary operator requires two operands, one before the operator and one after the operator:

```
operand1 operator operand2
```

For example, `3+4` or `x*y`.

A unary operator requires a single operand, either before or after the operator:

```
operator operand
```

or

```
operand operator
```

For example, `x++` or `++x`.

Assignment operators

An assignment operator assigns a value to its left operand based on the value of its right operand. The simple assignment operator is equal (=), which assigns the value of its right operand to its left operand. That is, x = y assigns the value of y to x.

There are also compound assignment operators that are shorthand for the operations listed in the following table:

Table 3.1 Assignment operators

Compound assignment operator	Meaning
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x %= y`	`x = x % y`
`x <<= y`	`x = x << y`
`x >>= y`	`x = x >> y`
`x >>>= y`	`x = x >>> y`
`x &= y`	`x = x & y`
`x ^= y`	`x = x ^ y`
`x |= y`	`x = x | y`

Comparison operators

A comparison operator compares its operands and returns a logical value based on whether the comparison is true. The operands can be numerical, string, logical, or object values. Strings are compared based on standard lexicographical ordering, using Unicode values. In most cases, if the two operands are not of the same type, JavaScript attempts to convert them to an appropriate type for the comparison. This behavior generally results in comparing the operands numerically. The sole exceptions to type conversion within comparisons involve the === and !== operators, which perform strict equality and inequality comparisons. These operators do not attempt to convert the operands to compatible types before checking equality. The following table describes the comparison operators in terms of this sample code:

```
var var1 = 3, var2 = 4;
```

Table 3.2 Comparison operators

Operator	Description	Examples returning true

Equal (`==`)	Returns true if the operands are equal.	`3 == var1`
		`"3" == var1`
		`3 == '3'`

Not equal (`!=`)	Returns true if the operands are not equal.	`var1 != 4`
		`var2 != "3"`

Strict equal (`===`)	Returns true if the operands are equal and of the same type. See also `Object.is`and sameness in JS.	`3 === var1`

Strict not equal (`!==`)	Returns true if the operands are not equal and/or not of the same type.	`var1 !== "3"`
		`3 !== '3'`

Greater than (`>`)	Returns true if the left operand is greater than the right operand.	`var2 > var1`
		`"12" > 2`

Greater than or equal (`>=`)	Returns true if the left operand is greater than or equal to the right operand.	`var2 >= var1`
		`var1 >= 3`

Less than (`<`)	Returns true if the left operand is less than the right operand.	`var1 < var2`
		`"2" < "12"`

Less than or equal (`<=`)	Returns true if the left operand is less than or equal to the right operand.	`var1 <= var2`
		`var2 <= 5`

Arithmetic operators

Arithmetic operators take numerical values (either literals or variables) as their operands and return a single numerical value. The standard arithmetic operators are addition (+), subtraction (-), multiplication (*), and division (/). These operators work as they do in most other programming languages when used with floating point numbers (in particular, note that division by zero produces`Infinity`). For example:

```
console.log(1 / 2); /* prints 0.5 */ console.log(1 / 2 == 1.0 / 2.0); /* also this is true */
```

In addition, JavaScript provides the arithmetic operators listed in the following table.

Table 3.3 Arithmetic operators

Operator	Description	Example

Table 3.3 Arithmetic operators

Operator	Description	Example
`%` (Modulus)	Binary operator. Returns the integer remainder of dividing the two operands.	12 % 5 returns 2.
`++` (Increment)	Unary operator. Adds one to its operand. If used as a prefix operator (`++x`), returns the value of its operand after adding one; if used as a postfix operator (`x++`), returns the value of its operand before adding one.	If `x` is 3, then `++x` sets `x` to 4 and returns 4, whereas `x++`returns 3 and, only then, sets`x` to 4.
`--` (Decrement)	Unary operator. Subtracts one from its operand. The return value is analogous to that for the increment operator.	If `x` is 3, then `--x` sets `x` to 2 and returns 2, whereas `x--`returns 3 and, only then, sets`x` to 2.
`-` (Unary negation)	Unary operator. Returns the negation of its operand.	If `x` is 3, then `-x` returns -3.

Bitwise operators

Bitwise operators treat their operands as a set of 32 bits (zeros and ones), rather than as decimal, hexadecimal, or octal numbers. For example, the decimal number nine has a binary representation of 1001. Bitwise operators perform their operations on such binary representations, but they return standard JavaScript numerical values.

The following table summarizes JavaScript's bitwise operators.

Table 3.4 Bitwise operators

Operator	Usage	Description
Bitwise AND	`a & b`	Returns a one in each bit position for which the corresponding bits of both operands are ones.
Bitwise OR	`a | b`	Returns a one in each bit position for which the corresponding bits of either or both operands are ones.
Bitwise XOR	`a ^ b`	Returns a one in each bit position for which the corresponding bits of either but not both operands are ones.
Bitwise NOT	`~ a`	Inverts the bits of its operand.
Left shift	`a << b`	Shifts `a` in binary representation `b` bits to the left, shifting in zeros from the right.
Sign-propagating right shift	`a >> b`	Shifts `a` in binary representation `b` bits to the right, discarding bits shifted off.

Table 3.4 Bitwise operators

Operator	Usage	Description
Zero-fill right shift	`a >>> b`	Shifts `a` in binary representation `b` bits to the right, discarding bits shifted off, and shifting in zeros from the left.

Bitwise logical operators

Conceptually, the bitwise logical operators work as follows:

- The operands are converted to thirty-two-bit integers and expressed by a series of bits (zeros and ones).
- Each bit in the first operand is paired with the corresponding bit in the second operand: first bit to first bit, second bit to second bit, and so on.
- The operator is applied to each pair of bits, and the result is constructed bitwise.

For example, the binary representation of nine is 1001, and the binary representation of fifteen is 1111. So, when the bitwise operators are applied to these values, the results are as follows:

Table 3.5 Bitwise operator examples

Expression	Result	Binary Description
`15 & 9`	`9`	`1111 & 1001 = 1001`
`15 | 9`	`15`	`1111 | 1001 = 1111`
`15 ^ 9`	`6`	`1111 ^ 1001 = 0110`
`~15`	`-16`	`~00000000...00001111 = 11111111...11110000`
`~9`	`-10`	`~00000000...00001001 = 11111111...11110110`

Note that all 32 bits are inverted using the Bitwise NOT operator, and that values with the most significant (left-most) bit set to 1 represent negative numbers (two's-complement representation).

Bitwise shift operators

The bitwise shift operators take two operands: the first is a quantity to be shifted, and the second specifies the number of bit positions by which the first operand is to be shifted. The direction of the shift operation is controlled by the operator used.

Shift operators convert their operands to thirty-two-bit integers and return a result of the same type as the left operand.

The shift operators are listed in the following table.

Table 3.6 Bitwise shift operators

Operator	Description	Example
`<<` (Left shift)	This operator shifts the first operand the specified number of bits to the left. Excess bits shifted off to	`9<<2` yields 36, because 1001 shifted 2 bits to the left becomes 100100, which is 36.

Table 3.6 Bitwise shift operators

Operator	Description	Example
	the left are discarded. Zero bits are shifted in from the right.	
`>>` (Sign-propagating right shift)	This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Copies of the leftmost bit are shifted in from the left.	`9>>2` yields 2, because 1001 shifted 2 bits to the right becomes 10, which is 2. Likewise, `-9>>2` yields -3, because the sign is preserved.
`>>>` (Zero-fill right shift)	This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Zero bits are shifted in from the left.	`19>>>2` yields 4, because 10011 shifted 2 bits to the right becomes 100, which is 4. For non-negative numbers, zero-fill right shift and sign-propagating right shift yield the same result.

Logical operators

Logical operators are typically used with Boolean (logical) values; when they are, they return a Boolean value. However, the && and || operators actually return the value of one of the specified operands, so if these operators are used with non-Boolean values, they may return a non-Boolean value. The logical operators are described in the following table.

Table 3.6 Logical operators

Operator	Usage	Description
`&&`	`expr1 && expr2`	(Logical AND) Returns `expr1` if it can be converted to false; otherwise, returns `expr2`. Thus, when used with Boolean values, `&&` returns true if both operands are true; otherwise, returns false.
`||`	`expr1 || expr2`	(Logical OR) Returns `expr1` if it can be converted to true; otherwise, returns `expr2`. Thus, when used with Boolean values, `||` returns true if either operand is true; if both are false, returns false.
`!`	`!expr`	(Logical NOT) Returns false if its single operand can be converted to true; otherwise, returns true.

Examples of expressions that can be converted to false are those that evaluate to null, 0, NaN, the empty string (""), or undefined.

The following code shows examples of the && (logical AND) operator.

```
var a1 =  true && true;     // t && t returns true var a2 =  true && false;    // t && f returns false var a3 = false && true;
// f && t returns false var a4 = false && (3 == 4); // f && f returns false var a5 = "Cat" && "Dog";    // t && t returns Dog
var a6 = false && "Cat";    // f && t returns false var a7 = "Cat" && false;    // t && f returns false
```

The following code shows examples of the || (logical OR) operator.

```
var o1 =  true || true;     // t || t returns true var o2 = false || true;     // f || t returns true var o3 =  true || false;
// t || f returns true var o4 = false || (3 == 4); // f || f returns false var o5 = "Cat" || "Dog";    // t || t returns Cat
var o6 = false || "Cat";    // f || t returns Cat var o7 = "Cat" || false;    // t || f returns Cat
```

The following code shows examples of the ! (logical NOT) operator.

```
var n1 = !true;  // !t returns false var n2 = !false; // !f returns true var n3 = !"Cat"; // !t returns false
```

Short-circuit evaluation

As logical expressions are evaluated left to right, they are tested for possible "short-circuit" evaluation using the following rules:

- `false` && *anything* is short-circuit evaluated to false.
- `true` || *anything* is short-circuit evaluated to true.

The rules of logic guarantee that these evaluations are always correct. Note that the *anything* part of the above expressions is not evaluated, so any side effects of doing so do not take effect.

String operators

In addition to the comparison operators, which can be used on string values, the concatenation operator (+) concatenates two string values together, returning another string that is the union of the two operand strings. For example, `"my " + "string"`returns the string `"my string"`.

The shorthand assignment operator += can also be used to concatenate strings. For example, if the variable `mystring` has the value "alpha", then the expression `mystring += "bet"` evaluates to "alphabet" and assigns this value to `mystring`.

Special operators

JavaScript provides the following special operators:

- Conditional operator
- Comma operator
- `delete`
- `in`
- `instanceof`
- `new`
- `this`
- `typeof`
- `void`

Conditional operator

The conditional operator is the only JavaScript operator that takes three operands. The operator can have one of two values based on a condition. The syntax is:

```
condition ? val1 : val2
```

If `condition` is true, the operator has the value of `val1`. Otherwise it has the value of `val2`. You can use the conditional operator anywhere you would use a standard operator.

For example,

```
var status = (age >= 18) ? "adult" : "minor";
```

This statement assigns the value "adult" to the variable `status` if `age` is eighteen or more. Otherwise, it assigns the value "minor" to`status`.

Comma operator

The comma operator (`,`) simply evaluates both of its operands and returns the value of the second operand. This operator is primarily used inside a `for` loop, to allow multiple variables to be updated each time through the loop.

For example, if `a` is a 2-dimensional array with 10 elements on a side, the following code uses the comma operator to increment two variables at once. The code prints the values of the diagonal elements in the array:

```
for (var i = 0, j = 9; i <= j; i++, j--)   document.writeln("a[" + i + "][" + j + "]= " + a[i][j]);
```

`delete`

The `delete` operator deletes an object, an object's property, or an element at a specified index in an array. The syntax is:

```
delete objectName; delete objectName.property; delete objectName[index]; delete property; // legal only within a with
statement
```

where `objectName` is the name of an object, `property` is an existing property, and `index` is an integer representing the location of an element in an array.

The fourth form is legal only within a `with` statement, to delete a property from an object.

You can use the `delete` operator to delete variables declared implicitly but not those declared with the `var` statement.

If the `delete` operator succeeds, it sets the property or element to `undefined`. The `delete` operator returns true if the operation is possible; it returns false if the operation is not possible.

```
x = 42; var y = 43; myobj = new Number(); myobj.h = 4;    // create property h delete x;       // returns true (can delete if
declared implicitly) delete y;       // returns false (cannot delete if declared with var) delete Math.PI; // returns false
(cannot delete predefined properties) delete myobj.h; // returns true (can delete user-defined properties) delete myobj;   //
returns true (can delete if declared implicitly)
```

Deleting array elements

When you delete an array element, the array length is not affected. For example, if you delete `a[3]`, `a[4]` is still `a[4]` and `a[3]` is undefined.

When the `delete` operator removes an array element, that element is no longer in the array. In the following example, `trees[3]` is removed with `delete`. However, `trees[3]` is still addressable and returns `undefined`.

```
var trees = new Array("redwood", "bay", "cedar", "oak", "maple"); delete trees[3]; if (3 in trees) {   // this does not get
executed }
```

If you want an array element to exist but have an undefined value, use the `undefined` keyword instead of the `delete` operator. In the following example, `trees[3]` is assigned the value `undefined`, but the array element still exists:

```
var trees = new Array("redwood", "bay", "cedar", "oak", "maple"); trees[3] = undefined; if (3 in trees) {   // this gets
executed }
```

`in`

The `in` operator returns true if the specified property is in the specified object. The syntax is:

```
propNameOrNumber in objectName
```

where `propNameOrNumber` is a string or numeric expression representing a property name or array index, and `objectName` is the name of an object.

The following examples show some uses of the in operator.

```
// Arrays var trees = new Array("redwood", "bay", "cedar", "oak", "maple"); 0 in trees;        // returns true 3 in trees;
// returns true 6 in trees;        // returns false "bay" in trees;    // returns false (you must specify the index number,
// not the value at that index) "length" in trees; // returns true (length is an Array property)  // Predefined objects "PI"
in Math;          // returns true var myString = new String("coral"); "length" in myString;  // returns true  // Custom
objects var mycar = {make: "Honda", model: "Accord", year: 1998}; "make" in mycar;  // returns true "model" in mycar; //
returns true
```

instanceof

The instanceof operator returns true if the specified object is of the specified object type. The syntax is:

```
objectName instanceof objectType
```

where objectName is the name of the object to compare to objectType, and objectType is an object type, such as Date or Array.

Use instanceof when you need to confirm the type of an object at runtime. For example, when catching exceptions, you can branch to different exception-handling code depending on the type of exception thrown.

For example, the following code uses instanceof to determine whether theDay is a Date object. Because theDay is a Date object, the statements in the if statement execute.

```
var theDay = new Date(1995, 12, 17); if (theDay instanceof Date) {   // statements to execute }
```

new

You can use the new operator to create an instance of a user-defined object type or of one of the predefined object types Array,Boolean, Date, Function, Image, Number, Object, Option, RegExp, or String. On the server, you can also use it with DbPool, Lock,File, or SendMail. Use new as follows:

```
var objectName = new objectType([param1, param2, ..., paramN]);
```

You can also create objects using object initializers, as described in using object initializers.

See the new operator page in the Core JavaScript Reference for more information.

this

Use the this keyword to refer to the current object. In general, this refers to the calling object in a method. Use this as follows:

```
this["propertyName"]
this.propertyName
```

Example 1.

Suppose a function called validate validates an object's value property, given the object and the high and low values:

```
function validate(obj, lowval, hival){   if ((obj.value < lowval) || (obj.value > hival))     alert("Invalid Value!"); }
```

You could call validate in each form element's onChange event handler, using this to pass it the form element, as in the following example:

```
<B>Enter a number between 18 and 99:</B> <INPUT TYPE="text" NAME="age" SIZE=3    onChange="validate(this, 18, 99);">
```

Example 2.

When combined with the form property, this can refer to the current object's parent form. In the following example, the formmyForm contains a Text object and a button.

When the user clicks the button, the value of the `Text` object is set to the form's name. The button's `onClick` event handler uses `this.form` to refer to the parent form, `myForm`.

```
<FORM NAME="myForm"> Form name:<INPUT TYPE="text" NAME="text1" VALUE="Beluga"/> <INPUT NAME="button1" TYPE="button"
VALUE="Show Form Name"    onClick="this.form.text1.value = this.form.name;"/> </FORM>
```

`typeof`

The `typeof` operator is used in either of the following ways:

```
1. typeof operand
2. typeof (operand)
```

The `typeof` operator returns a string indicating the type of the unevaluated operand. `operand` is the string, variable, keyword, or object for which the type is to be returned. The parentheses are optional.

Suppose you define the following variables:

```
var myFun = new Function("5 + 2"); var shape = "round"; var size = 1; var today = new Date();
```

The `typeof` operator returns the following results for these variables:

```
typeof myFun;     // returns "function" typeof shape;     // returns "string" typeof size;      // returns "number" typeof
today;     // returns "object" typeof dontExist; // returns "undefined"
```

For the keywords `true` and `null`, the `typeof` operator returns the following results:

```
typeof true; // returns "boolean" typeof null; // returns "object"
```

For a number or string, the `typeof` operator returns the following results:

```
typeof 62;            // returns "number" typeof 'Hello world'; // returns "string"
```

For property values, the `typeof` operator returns the type of value the property contains:

```
typeof document.lastModified; // returns "string" typeof window.length;         // returns "number" typeof Math.LN2;
// returns "number"
```

For methods and functions, the `typeof` operator returns results as follows:

```
typeof blur;        // returns "function" typeof eval;        // returns "function" typeof parseInt;    // returns "function"
typeof shape.split; // returns "function"
```

For predefined objects, the `typeof` operator returns results as follows:

```
typeof Date;     // returns "function" typeof Function; // returns "function" typeof Math;     // returns "object" typeof
Option;   // returns "function" typeof String;   // returns "function"
```

`void`

The `void` operator is used in either of the following ways:

```
1. void (expression)
2. void expression
```

The `void` operator specifies an expression to be evaluated without returning a value. `expression` is a JavaScript expression to evaluate. The parentheses surrounding the expression are optional, but it is good style to use them.

You can use the `void` operator to specify an expression as a hypertext link. The expression is evaluated but is not loaded in place of the current document.
The following code creates a hypertext link that does nothing when the user clicks it. When the user clicks the link, `void(0)`evaluates to undefined, which has no effect in JavaScript.

```
<A HREF="javascript:void(0)">Click here to do nothing</A>
```

The following code creates a hypertext link that submits a form when the user clicks it.

```
<A HREF="javascript:void(document.form.submit())"> Click here to submit</A>
```

Operator precedence

The *precedence* of operators determines the order they are applied when evaluating an expression. You can override operator precedence by using parentheses.
The following table describes the precedence of operators, from highest to lowest.

Table 3.7 Operator precedence

Operator type	Individual operators
member	`. []`
call / create instance	`() new`
negation/increment	`! ~ - + ++ -- typeof void delete`
multiply/divide	`* / %`
addition/subtraction	`+ -`
bitwise shift	`<< >> >>>`
relational	`< <= > >= in instanceof`
equality	`== != === !==`
bitwise-and	`&`
bitwise-xor	`^`
bitwise-or	`|`
logical-and	`&&`

Table 3.7 Operator precedence

Operator type	Individual operators
logical-or	`||`
conditional	`?:`
assignment	`= += -= *= /= %= <<= >>= >>>= &= ^= |=`
comma	`,`