Introduction to JavaScript

DroidScript - Language

A basic introduction to the JavaScript language with the most useful and commonly used parts of the JavaScript language. This manual is a resume offline of the web site http://droidscript.org/javascript/index.html

9/18/2022

INTRODUCTION TO THE JAVASCRIPT

Transcribed by Gustavo Puelma – gpuelma@gmail.com

INTRODUCTION TO THE JAVASCRIPT

PREFACE

This book is a basic introduction to the JavaScript language with the most useful and commonly used parts of the JavaScript language. This manual is a resume offline of the web site http://droidscript.org/javascript/index.html

JavaScript is now one of the most popular and useful computer languages on the planet. It can be used to create web pages, web servers, mobile Apps and even inside embedded micro-controllers!

Statements

JavaScript 'Apps' are computer programs written as lines of text. Each line of text is made up of one or more **statements**, which are separated by a ';' (semi-colon) character like this:-

```
statement1;
statement2;
statement3;
```

These statements contain instructions for the computer to perform (execute) and the ';' character at the end tells the computer where each statement ends and the next one starts.

We usually place statements on separate lines as this can help when looking for bugs (problems). Here's a simple JavaScript program made using two statements. It displays the word "Hello" in a popup box:

```
var s = "Hello";
alert( s );
```

Comments

You should also add comments to your programs, which are ignored by the computer but help you and others understand what the program is supposed to be doing. To do this you use the **'//'** (double forward slash) characters.

For example we can add a comment line to the program above:

```
//This shows a popup message.
var s = "Hello";
alert( s );
```

Variables

Variables are declared using the **'var'** keyword and allow us to store data such as text, numbers and objects in the computer's memory using a name of our choosing. We can then read and change the stored data later by using the name we have given to the variable.

We can store text and display it like this:

```
//This shows the message 'Hi'.
var myText = "Hi";
alert( myText );
```

Note: Text data should always be enclosed in quotes or the computer will think it's a variable name.

We can store a number and then display it like this:

```
//This shows the message '7'.
var myNum = 7;
alert( myNum );
```

We can change the contents of our variables like this:

```
//This shows the message '6'.
var myNum = 3;
myNum = myNum * 2;
alert( myNum );
```

Note: JavaScript is case sensitive, so myNum is not the same MyNum.

Expressions

A statement such as myNum = myNum + 2 is an **expression** and we can perform various maths and text operations on our variables using expressions.

For example:

```
//This shows the message 'Answer: 17'.
var myNum = 7;
var myText = "";
myNum = myNum * 2 + 3;
myText = "Answer: " + myNum;
alert( myText );
```

Note: When you add a number to text, the number is automatically converted to text.

Data Types

In JavaScript, we can store and use these data types:

String	- Text data.
Number	- Numbers.
Boolean	- True or false values.
Array	- Lists of data.
Object	- Complex data types.

We can store all of these types in variables and also use them in expressions, but the **Array** and **Object** data types are special **'reference'** types, which means variables containing these types actually hold a reference (or pointer) to the original data rather than the actual data.

When you copy reference types from one variable to another, you will not get a separate copy of the data in the computer's memory, but rather a copy of the reference.

Here are a few examples of using various data types:

```
//This shows the message 'true'.
var isGood = true;
alert( isGood );
//This calculates a circle's circumference.
var pi = 3.14;
var r = 5;
alert( 2 * pi * r );
//This shows the message 'Hello World'.
var s = "Hello";
alert( s + " World" );
```

```
//This shows the message 'First item: 3'.
var myArray = [3,4,5];
var firstItem = myArray[0];
alert( "First item: " + firstItem );
//This shows the message 'First item: fred'.
//(Arrays are reference types).
var myArray1 = ["a","b","c"];
var myArray2 = myArray1;
myArray1[0] = "fred";
alert( "First item: " + myArray2[0] );
```

Conditional Statements

We often want to take different actions in our program depending on certain conditions and the easiest way to do this is using an **'if'** statement like this:

```
if( x > 1000 ) splat = true;
```

The line of code above would set the variable 'splat' to true if the variable 'x' contains a value greater than 1000. We can also use the **'else'** statement in combination with **'if'** statement to do one thing or the other:

```
if( splat ) image = "splat.jpg";
else image = "bird.jpg";
```

If we want to check a whole lot of conditions, we can do something like this:

```
//Find the correct photo.
if( name=="Sam" && male ) image = "Samuel.jpg";
else if( name=="Sam" ) image = "Samanther.jpg";
else if( name=="Bill" ) image = "William.jpg";
else image = "Anyone.jpg";
```

Note: We use the double equals sign when we comparing variables and the single equals sign when we are assigning (setting) variables.

In order to execute more than one statement after an **'if'** or **'else'** clause, we can use the **'{}'** (brace) characters to group statements into a **'block'** of code:

For example, in a game we might have some statements like this to check if the character has crashed and the game is over:

```
if( splat )
{
    imageName = "splat.jpg";
    gameOver = true;
}
else
{
    imageName = "bird.jpg";
    gameOver = false;
    count++;
}
```

Operators

We can perform various mathematical, logical and text operations in JavaScript such as add, subtract, divide and multiply but you will often see a statement like this:

```
num += 2;
```

This may look confusing at first, but this '+=' operation simply adds 2 to the contents of variable 'num'. This is a shorter and more efficient way of writing the following:

```
num = num + 2;
```

Here are some examples of the most common operations and their meanings:

```
z = x * y;    //z = x multiplied by y
z = x / y;    //z = x divided by y
z = x % y;    //z = remainder of x divided by y
num++;        //add 1 to the contents of num
num--;        //subtract 1 from contents of num
num += 3;     //add 3 to the contents of num
num -= x;     //subtract x from contents of num
num *= 9;     //multiply the contents of num by 9
num /= 9;     //divide the contents of num by 9
b = 7;        //set b to the value 7
if( b == 4 )  //if b is equal to 4
if( b != c )  //if b is not equal to c
```

```
if( b > c )   //if b is greater than c
if( b < c )   //if b is less than c
if( b >= c )  //if b is greater or equal to c
if( b <= c )  //if b is less than or equal to c
if( b || c )  //if b or c is true
if( b && c )  //if b and c are true
if( b && !c ) //if b is true and c is false
```

Another very useful operation is the **?:** or **'Ternary'** operation. This allows us to conditionally get one of two possible values using a single line of code. For example:

```
adult = (age > 18 ? true : false);
```

This is the same as writing:

```
if( age > 18 ) adult = true;
else adult = false;
```

You can use this in expressions too. Here's another example:

```
meals = 7 * (name=="Sam" ? 6 : 3);
```

Loops

In JavaScript we often want to repeat a block of statements many times and the two most common ways of doing this are with the **'while'** and **'for'** statements.

A **'while'** statement will keep repeating a block of statements while a given expression is true. For example, the following code will keep asking the user if they are ready until they answer "yes".

```
var answer;
while( answer != "yes" )
{
    answer = prompt( "Are you ready?" );
}
```

A **'for'** statement is usually used to repeat a block of statements until a counter has reached a certain value. The following code will add the letter 'O' to the letter "G" ten times, followed by the letters "GLE".

```
var txt = "G";
for( var i=0; i<10; i++ )
{
    txt += "O";
}
txt += "GLE"
alert( txt );
```

In the above sample, the counter variable 'i' starts at zero and increases by 1 each time the block of code is complete, but only while 'i' contains a value less than 10. So the variable 'i' will range from zero to nine.

Functions

If we find that we need to execute the same (or a similar) group of statements from various parts of our program, then instead of copying and pasting the same group of statements all over our program, it is far more efficient and much neater to 'wrap' a block of code up using the **'function'** statement. We can then use a single line of code to 'call' our function and execute the block of code.

For example, if we needed to show a random number to the user and we expect to do that more than once in our program, then we could define a function called 'ShowRandom' and call it like this:

```
//Show a random number.
function ShowRandom()
{
    var num = Math.random();
    alert( num );
}

ShowRandom();
```

If we want the function to behave slightly differently each time we call it, then we can pass input values to the function. These are known as input **'parameters'**.

So for example we might want to change the range of the random numbers produced by our 'ShowRandom' function by providing a 'range' parameter like this:

```
//Show a random number in a given range.
function ShowRandom( range )
{
    var num = Math.random() * range;
    alert( num );
}

ShowRandom( 10 );
ShowRandom( 100 );
```

As well as using input values, we can also get an output ('return') value from our function too, by using the **'return'** statement. This allows our function to do calculations based on the input parameters and return a result afterwards.

We could for example create our own function to calculate the area of a circle like this:

```
//Get the area of a circle.
//(to two decimal places)
function GetAreaOfCircle( radius )
{
    var area = Math.PI * radius * radius;
    return area.toFixed(2);
}

//Show area of a circle with radius 4.8cm
var area = GetAreaOfCircle( 4.8 );
alert( "Area = " + area + " cm" );
```

In Javascript we don't have to place our function before the statement which uses it, so we can put it at the bottom of our script if we prefer, like this:

```
//Show volume of box 4cm x 4cm x 2cm
var vol = GetVolumeOfBox( 4 ,4, 2 );
alert( "Volume = " + vol );
```

```
//Get the volume of a box.
function GetVolumeOfBox( width, height, depth )
{
    var volume = width * height * depth;
    return  volume;
}
```

String Handling

We often want to manipulate strings (text) in JavaScript and there are a number of useful methods available to us for doing this. Note that the position of characters within a string (known as their 'index') always starts at zero.

One of the most useful string methods is the **'split'** method. This allows us to split the string into parts and store each part in an array. The split method takes a single parameter which tells the computer which character to look for while splitting the string.

For example if we want to get the first and third names in a comma separated list of names, we do the following:

```
//Get the first and third list items.
var names = "Fred,Bill,Amy,Sally";
var namesArray = names.split(",");
var firstName = namesArray[0];
var thirdName = namesArray[2];
```

Another very useful string method is the **'indexOf'** method. This allows us to find the index (position) of a string within another string.

For example, we might want to check if a file is a text file by searching for the string '.txt' in its file name like this:

```
//Check for text files.
var isTextFile = false;
if( fileName.indexOf(".txt" ) > -1 )
    isTextFile = true;
```

Here are some more examples of the many string manipulation methods available to us in JavaScript:

```
//Set our test string
var txt = "Big Friendly Giant";

//Get the number of characters in the string.
var numLetters = txt.length;

//Get a copy the first character.
var firstLetter = txt.slice( 0, 1 );

//Get a copy the last character.
var lastLetter = txt.slice( -1 );

//Get a copy the first word.
var firstWord = txt.substring( 0, 3 );

//Get a copy of the second word.
var secondWord = txt.substring( 4, 12 );

//Get the start index of the word 'Giant'.
var indexOfGiant = txt.indexOf( "Giant" );

//Get the start index of the word 'Big'.
var indexOfBig = txt.indexOf( "Big" );

//Get the index of the last space.
var indexOfLastSpace = txt.lastIndexOf( " " );

//Get a copy of string with word replaced.
var changed = txt.replace( "Big", "Little" );

//Get a lower case copy of the string.
var lowerCase = txt.toLowerCase();
```

```
//Get an upper case copy of the string.
var upperCase = txt.toUpperCase();
```

Statements

JavaScript applications consist of statements with an appropriate syntax. A single statement may span multiple lines. Multiple statements may occur on a single line if each statement is separated by a semicolon. This isn't a keyword, but a group of keywords.

Statements and declarations by category

For an alphabetical listing see the sidebar on the left.

Control flow

`Block` A block statement is used to group zero or more statements. The block is delimited by a pair of curly brackets.

`break` Terminates the current loop, switch, or label statement and transfers program control to the statement following the terminated statement.

`continue`

Terminates execution of the statements in the current iteration of the current or labeled loop, and continues execution of the loop with the next iteration.

`Empty` An empty statement is used to provide no statement, although the JavaScript syntax would expect one.

`if...else`

Executes a statement if a specified condition is true. If the condition is false, another statement can be executed.

`switch` Evaluates an expression, matching the expression's value to a case clause, and executes statements associated with that case.

`throw` Throws a user-defined exception.

`try...catch`

Marks a block of statements to try, and specifies a response, should an exception be thrown.

Declarations

`var` Declares a variable, optionally initializing it to a value.

Functions and classes

`function`

Declares a function with the specified parameters.

`return` Specifies the value to be returned by a function.

Iterations

`do...while`

Creates a loop that executes a specified statement until the test condition evaluates to false. The condition is evaluated after executing the statement, resulting in the specified statement executing at least once.

`for` Creates a loop that consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by a statement executed in the loop.

`for...in`

Iterates over the enumerable properties of an object, in arbitrary order. For each distinct property, statements can be executed.

`for...of`

Iterates over iterable objects (including arrays, array-like objects, iterators and generators), invoking a custom iteration hook with statements to be executed for the value of each distinct property.

`while` Creates a loop that executes a specified statement as long as the test condition evaluates to true. The condition is evaluated before executing the statement.

Operators

This chapter documents many of the JavaScript language operators, expressions and keywords.

Expressions and operators by category

For an alphabetical listing see the sidebar on the left.

Primary expressions

Basic keywords and general expressions in JavaScript.

`function`

: The `function` keyword defines a function expression.

`[]` Array initializer/literal syntax.

`{}` Object initializer/literal syntax.

`/ab+c/i`

: Regular expression literal syntax.

`( )` Grouping operator.

Left-hand-side expressions

Left values are the destination of an assignment.

`Property accessors`

: Member operators provide access to a property or method of an object (`object.property` and `object["property"]`).

`new` The `new` operator creates an instance of a constructor.

Increment and decrement

Postfix/prefix increment and postfix/prefix decrement operators.

`A++` Postfix increment operator.

`A--` Postfix decrement operator.

`++A` Prefix increment operator.

`--A` Prefix decrement operator.

Unary operators

A unary operation is operation with only one operand.

`delete` The `delete` operator deletes a property from an object.

`void` The `void` operator discards an expression's return value.

`typeof` The `typeof` operator determines the type of a given object.

`+` The unary plus operator converts its operand to Number type.

`-` The unary negation operator converts its operand to Number type and then negates it.

`~` Bitwise NOT operator.

`!` Logical NOT operator.

Arithmetic operators

Arithmetic operators take numerical values (either literals or variables) as their operands and return a single numerical value.

`+` Addition operator.

`-` Subtraction operator.

`/` Division operator.

`*` Multiplication operator.

`%` Remainder operator.

Relational operators

A comparison operator compares its operands and returns a `Boolean` value based on whether the comparison is true.

`in` The `in` operator determines whether an object has a given property.

`instanceof`

The `instanceof` operator determines whether an object is an instance of another object.

`<` Less than operator.

`>` Greater than operator.

`<=` Less than or equal operator.

`>=` Greater than or equal operator.

Equality operators

The result of evaluating an equality operator is always of type `Boolean` based on whether the comparison is true.

`==` Equality operator.

`!=` Inequality operator.

`===` Identity operator.

`!==` Nonidentity operator.

Bitwise shift operators

Operations to shift all bits of the operand.

`<<` Bitwise left shift operator.

`>>` Bitwise right shift operator.

`>>>` Bitwise unsigned right shift operator.

Binary bitwise operators

Bitwise operators treat their operands as a set of 32 bits (zeros and ones) and return standard JavaScript numerical values.

`&` Bitwise AND.

`|` Bitwise OR.

`^` Bitwise XOR.

Binary logical operators

Logical operators are typically used with boolean (logical) values, and when they are, they return a boolean value.

`&&` Logical AND.

`||` Logical OR.

Conditional (ternary) operator

```
(condition ? ifTrue : ifFalse)
```

The conditional operator returns one of two values based on the logical value of the condition.

Assignment operators

An assignment operator assigns a value to its left operand based on the value of its right operand.

`=` Assignment operator.

`*=` Multiplication assignment.

`/=` Division assignment.

`%=` Remainder assignment.

`+=` Addition assignment.

`-=` Subtraction assignment

`<<=` Left shift assignment.

`>>=` Right shift assignment.

`>>>=` Unsigned right shift assignment.

`&=` Bitwise AND assignment.

`^=` Bitwise XOR assignment.

`|=` Bitwise OR assignment.

Comma operator

`,` The comma operator allows multiple expressions to be evaluated in a single statement and returns the result of the last expression.

Global Objects

This chapter documents the most useful JavaScript built-in objects, including their methods and properties.

Standard objects by category

Value properties

These global properties return a simple value; they have no properties or methods.

- `Infinity`
- `NaN`
- `undefined`
- `null` literal

Function properties

These global functionsâ€”functions which are called globally rather than on an objectâ€”directly return their results to the caller.

- `eval()`
- `isFinite()`
- `isNaN()`
- `parseFloat()`
- `parseInt()`
- `decodeURI()`
- `decodeURIComponent()`
- `encodeURI()`
- `encodeURIComponent()`

Numbers and dates

These are the base objects representing numbers, dates, and mathematical calculations.

- `Number`
- `Math`
- `Date`

Text processing

These objects represent strings and support manipulating them.

- String
- RegExp

---

block

A **block statement** (or **compound statement** in other languages) is used to group zero or more statements. The block is delimited by a pair of curly brackets.

Syntax

```
{
  statement_1;
  statement_2;
  ...
  statement_n;
}
```

`statement_1`, `statement_2`, `statement_n`

Statements grouped within the block statement.

Description

This statement is commonly used with control flow statements (e.g. **if...else**, **for**, **while**). For example:

```
while (x < 10) {
  x++;
}
```

Note that the block statement does not end with a semicolon.

The block statement is often called **compound statement** in other languages. It allows you to use multiple statements where JavaScript expects only one statement. Combining statements into blocks

is a common practice in JavaScript. The opposite behavior is possible using an empty statement, where you provide no statement, although one is required.

No block scope

Important: Variables declared with `var` do **not** have block scope. Variables introduced with a block are scoped to the containing function or script, and the effects of setting them persist beyond the block itself. In other words, block statements do not introduce a scope. Although "standalone" blocks are valid syntax, you do not want to use standalone blocks in JavaScript, because they don't do what you think they do, if you think they do anything like such blocks in C or Java. For example:

```
var x = 1;
{
  var x = 2;
}
console.log(x); // logs 2
```

This logs 2 because the `var x` statement within the block is in the same scope as the `var x` statement before the block. In C or Java, the equivalent code would have outputted 1.

---

break

The **break statement** terminates the current loop, **switch**, or **label** statement and transfers program control to the statement following the terminated statement.

Syntax

```
break [label];
label
```

Optional. Identifier associated with the label of the statement. If the statement is not a loop or **switch**, this is required.

Description

The `break` statement includes an optional label that allows the program to break out of a labeled statement. The `break` statement needs to be nested within the referenced label. The labeled statement can be any **block** statement; it does not have to be preceded by a loop statement.

Examples

The following function has a `break` statement that terminates the **while** loop when `i` is 3, and then returns the value 3 * `x`.

```
function testBreak(x) {
  var i = 0;

  while (i < 6) {
    if (i == 3) {
      break;
    }
    i += 1;
  }

  return i * x;
}
```

The following code uses `break` statements with labeled blocks. A `break` statement must be nested within any label it references. Notice that `inner_block` is nested within `outer_block`.

```
outer_block: {
  inner_block: {
    console.log('1');
    break outer_block; // breaks out of both inner_block and outer_block
    console.log(':-('); // skipped
  }
  console.log('2'); // skipped
}
```

The following code also uses `break` statements with labeled blocks but generates a Syntax Error because its `break` statement is within `block_1` but references `block_2`. A `break` statement must always be nested within any label it references.

```
block_1: {
  console.log('1');
  break block_2; // SyntaxError: label not found
```

```
}

block_2: {
  console.log('2');
}
```

---

continue

The **continue statement** terminates execution of the statements in the current iteration of the current or labeled loop, and continues execution of the loop with the next iteration.

Syntax

```
continue [label];
```

`label`

Identifier associated with the label of the statement.

Description

In contrast to the **break** statement, `continue` does not terminate the execution of the loop entirely: instead,

- In a **while** loop, it jumps back to the condition.
- In a **for** loop, it jumps to the update expression.

The `continue` statement can include an optional label that allows the program to jump to the next iteration of a labeled loop statement instead of the current loop. In this case, the `continue` statement needs to be nested within this labeled statement.

Examples

Using `continue` with `while`

The following example shows a **while** loop that has a `continue` statement that executes when the value of `i` is 3. Thus, `n` takes on the values 1, 3, 7, and 12.

```
var i = 0;
```

```
var n = 0;

while (i < 5) {
  i++;

  if (i === 3) {
    continue;
  }

  n += i;
}
```

Using `continue` with a label

In the following example, a statement labeled `checkiandj` contains a statement labeled `checkj`. If `continue` is encountered, the program continues at the top of the `checkj` statement. Each time `continue` is encountered, `checkj` reiterates until its condition returns false. When false is returned, the remainder of the `checkiandj` statement is completed.

If `continue` had a label of `checkiandj`, the program would continue at the top of the `checkiandj` statement.

See also **label**.

```
var i = 0;
var j = 8;

checkiandj: while (i < 4) {
  console.log("i: " + i);
  i += 1;

  checkj: while (j > 4) {
    console.log("j: "+ j);
    j -= 1;

    if ((j % 2) == 0)
      continue checkj;
```

```
    console.log(j + " is odd.");
  }
  console.log("i = " + i);
  console.log("j = " + j);
}
```

Output:

```
"i: 0"

// start checkj
"j: 8"
"7 is odd."
"j: 7"
"j: 6"
"5 is odd."
"j: 5"
// end checkj

"i = 1"
"j = 4"

"i: 1"
"i = 2"
"j = 4"

"i: 2"
"i = 3"
"j = 4"

"i: 3"
"i = 4"
"j = 4"
```

Empty

An **empty statement** is used to provide no statement, although the JavaScript syntax would expect one.

Syntax

```
;
```

Description

The empty statement is a semicolon (;) indicating that no statement will be executed, even if JavaScript syntax requires one. The opposite behavior, where you want multiple statements, but JavaScript only allows a single one, is possible using a block statement; it combines several statements into a single one.

Examples

The empty statement is sometimes used with loop statements. See the following example with an empty loop body:

```
var arr = [1, 2, 3];

// Assign all array values to 0
for (i = 0; i < arr.length; arr[i++] = 0) /* empty statement */ ;

console.log(arr)
// [0, 0, 0]
```

Note: It is a good idea to comment the intentional use of the empty statement, as it is not really obvious to distinguish between a normal semicolon. In the following example the usage is probably not intentional:

```
if (condition);       // Caution, this "if" does nothing!
   killTheUniverse()  // So this gets always executed!!!
```

Another Example: An `if...else` statement without curly braces (`{}`). If `three` is `true`, nothing will happen, `four` does not matter, and also the `launchRocket()` function in the `else` case will not be executed.

```
if (one)
  doOne();
else if (two)
  doTwo();
else if (three)
  ; // nothing here
else if (four)
  doFour();
else
  launchRocket();
```

---

if...else

The **if statement** executes a statement if a specified condition is true. If the condition is false, another statement can be executed.

Syntax

```
if (condition)
   statement1
[else
   statement2]
```

`condition`

An expression that evaluates to true or false.

`statement1`

Statement that is executed if `condition` evaluates to true. Can be any statement, including further nested `if` statements. To execute multiple statements, use a block statement ({ ... }) to group those statements, to execute no statements, use an empty statement.

`statement2`

Statement that is executed if `condition` evaluates to false and the `else` clause exists. Can be any statement, including block statements and further nested `if` statements.

Description

Multiple `if...else` statements can be nested to create an `else if` clause. Note that there is no `elseif` (in one word) keyword in JavaScript.

```
if (condition1)
   statement1
else if (condition2)
   statement2
else if (condition3)
   statement3
...
else
   statementN
```

To see how this works, this is how it would look like if the nesting were properly indented:

```
if (condition1)
   statement1
else
   if (condition2)
      statement2
   else
      if (condition3)
...
```

To execute multiple statements within a clause, use a block statement (`{ ... }`) to group those statements. In general, it is a good practice to always use block statements, especially in code involving nested `if` statements:

```
if (condition) {
   statements1
} else {
   statements2
}
```

Do not confuse the primitive boolean values `true` and `false` with the true and false values of the Boolean object. Any value that is not `undefined`, `null`, `0`, `NaN`, or the empty string (`""`), and any object, including a Boolean object whose value is false, evaluates to true when passed to a conditional statement. For example:

```
var b = new Boolean(false);
if (b) // this condition evaluates to true
```

Examples

Using `if...else`

```
if (cipher_char === from_char) {
   result = result + to_char;
   x++;
} else {
   result = result + clear_char;
}
```

Using `else if`

Note that there is no `elseif` syntax in JavaScript. However, you can write it with a space between `else` and `if`:

```
if (x > 5) {

} else if (x > 50) {

} else {

}
```

Assignment within the conditional expression

It is advisable to not use simple assignments in a conditional expression, because the assignment can be confused with equality when glancing over the code. For example, do not use the following code:

```
if (x = y) {
   /* do the right thing */
}
```

If you need to use an assignment in a conditional expression, a common practice is to put additional parentheses around the assignment. For example:

```
if ((x = y)) {
   /* do the right thing */
}
```

---

switch

The **switch statement** evaluates an expression, matching the expression's value to a case clause, and executes statements associated with that case.

Syntax

```
switch (expression) {
  case value1:
    //Statements executed when the result of expression matches value1
    [break;]
  case value2:
    //Statements executed when the result of expression matches value2
    [break;]
  ...
  case valueN:
    //Statements executed when the result of expression matches valueN
    [break;]
  default:
    //Statements executed when none of the values match the value of the
expression
    [break;]
}
```

`expression`

An expression whose result is matched against each case clause.

`case valueN`

A case clause used to match against `expression`.

Description

A switch statement first evaluates its expression. It then looks for the first `case` clause whose expression evaluates to the same value as the result of the input expression (using strict comparison, `===`) and transfers control to that clause, executing the associated statements. (If multiple cases match the provided value, the first case that matches is selected, even if the cases are not equal to each other.) If no matching `case` clause is found, the program looks for the optional `default` clause, and if found, transfers control to that clause, executing the associated statements. If no `default` clause is found, the program continues execution at the statement following the end of `switch`. By convention, the `default` clause is the last clause, but it does not need to be so.

The optional `break` statement associated with each case label ensures that the program breaks out of switch once the matched statement is executed and continues execution at the statement following switch. If `break` is omitted, the program continues execution at the next statement in the `switch` statement.

Examples

Using `switch`

In the following example, if `expr` evaluates to "Bananas", the program matches the value with case "Bananas" and executes the associated statement. When `break` is encountered, the program breaks out of `switch` and executes the statement following `switch`. If `break` were omitted, the statement for case "Cherries" would also be executed.

```
switch (expr) {
  case "Oranges":
    console.log("Oranges are $0.59 a pound.");
    break;
  case "Apples":
    console.log("Apples are $0.32 a pound.");
    break;
  case "Bananas":
    console.log("Bananas are $0.48 a pound.");
```

```
    break;
  case "Cherries":
    console.log("Cherries are $3.00 a pound.");
    break;
  case "Mangoes":
  case "Papayas":
    console.log("Mangoes and papayas are $2.79 a pound.");
    break;
  default:
    console.log("Sorry, we are out of " + expr + ".");
}

console.log("Is there anything else you'd like?");
```

What happens if I forgot a break?

If you forget a break then script will run from the case where criteria is met, and will run the case after that regardless if criteria was met. See example here:

```
var foo = 0;
switch (foo) {
  case -1:
    console.log('negative 1');
    break;
  case 0: // foo is 0 so criteria met here so this block will run
    console.log(0);
    // NOTE: the forgotten break would have been here
  case 1: // no break statement in 'case 0:' so this case will run as well
    console.log(1);
    break; // it encounters this break so will not continue into 'case 2:'
  case 2:
    console.log(2);
    break;
  default:
    console.log('default');
}
```

Methods for multi-criteria case

Source for this technique is here:

Switch statement multiple cases in JavaScript (Stack Overflow)

Multi-case - single operation

This method takes advantage of the fact that if there is no break below a case statement it will continue to execute the next case statement regardless if the case meets the criteria. See the section title "What happens if I forgot a break?"

This is an example of a single operation sequential switch statement, where four different values perform exactly the same.

```
var Animal = 'Giraffe';
switch (Animal) {
  case 'Cow':
  case 'Giraffe':
  case 'Dog':
  case 'Pig':
    console.log('This animal will go on Noah\'s Ark.');
    break;
  case 'Dinosaur':
  default:
    console.log('This animal will not.');
}
```

Multi-case - chained operations

This is an example of a multiple-operation sequential switch statement, where, depending on the provided integer, you can receive different output. This shows you that it wil traverse in the order that you put the case statements, and it does not have to be numerically sequential. In JavaScript, you can even mix in definitions of strings into these case statements as well.

```
var foo = 1;
var output = 'Output: ';
switch (foo) {
```

```
  case 10:
    output += 'So ';
  case 1:
    output += 'What ';
    output += 'Is ';
  case 2:
    output += 'Your ';
  case 3:
    output += 'Name';
  case 4:
    output += '?';
    console.log(output);
    break;
  case 5:
    output += '!';
    console.log(output);
    break;
  default:
    console.log('Please pick a number from 0 to 6!');
}
```

---

throw

The **`throw` statement** throws a user-defined exception. Execution of the current function will stop (the statements after `throw` won't be executed), and control will be passed to the first `catch` block in the call stack. If no `catch` block exists among caller functions, the program will terminate.

Syntax

```
throw expression;
```

`expression`

The expression to throw.

Description

Use the `throw` statement to throw an exception. When you throw an exception, `expression` specifies the value of the exception. Each of the following throws an exception:

```
throw "Error2"; // generates an exception with a string value
throw 42;       // generates an exception with the value 42
throw true;     // generates an exception with the value true
```

Also note that the `throw` statement is affected by automatic semicolon insertion (ASI) as no line terminator between the `throw` keyword and the expression is allowed.

Examples

Throw an object

You can specify an object when you throw an exception. You can then reference the object's properties in the `catch` block. The following example creates an object of type `UserException` and uses it in a `throw` statement.

```
function UserException(message) {
   this.message = message;
   this.name = "UserException";
}
function getMonthName(mo) {
   mo = mo-1; // Adjust month number for array index (1=Jan, 12=Dec)
   var months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
      "Aug", "Sep", "Oct", "Nov", "Dec"];
   if (months[mo] !== undefined) {
      return months[mo];
   } else {
      throw new UserException("InvalidMonthNo");
   }
}

try {
   // statements to try
   var myMonth = 15; // 15 is out of bound to raise the exception
   monthName = getMonthName(myMonth);
} catch (e) {
   monthName = "unknown";
   logMyErrors(e.message, e.name); // pass exception object to err handler
```

```
}
```

Another example of throwing an object

The following example tests an input string for a U.S. zip code. If the zip code uses an invalid format, the throw statement throws an exception by creating an object of type `ZipCodeFormatException`.

```
/*
 * Creates a ZipCode object.
 *
 * Accepted formats for a zip code are:
 *    12345
 *    12345-6789
 *    123456789
 *    12345 6789
 *
 * If the argument passed to the ZipCode constructor does not
 * conform to one of these patterns, an exception is thrown.
 */

function ZipCode(zip) {
   zip = new String(zip);
   pattern = /[0-9]{5}([- ]?[0-9]{4})?/;
   if (pattern.test(zip)) {
      // zip code value will be the first match in the string
      this.value = zip.match(pattern)[0];
      this.valueOf = function() {
         return this.value
      };
      this.toString = function() {
         return String(this.value)
      };
   } else {
      throw new ZipCodeFormatException(zip);
   }
}
```

```
function ZipCodeFormatException(value) {
   this.value = value;
   this.message = "does not conform to the expected format for a zip
code";
   this.toString = function() {
      return this.value + this.message;
   };
}

/*
 * This could be in a script that validates address data
 * for US addresses.
 */

const ZIPCODE_INVALID = -1;
const ZIPCODE_UNKNOWN_ERROR = -2;

function verifyZipCode(z) {
   try {
      z = new ZipCode(z);
   } catch (e) {
      if (e instanceof ZipCodeFormatException) {
         return ZIPCODE_INVALID;
      } else {
         return ZIPCODE_UNKNOWN_ERROR;
      }
   }
   return z;
}

a = verifyZipCode(95060);         // returns 95060
b = verifyZipCode(9560);          // returns -1
c = verifyZipCode("a");           // returns -1
d = verifyZipCode("95060");       // returns 95060
e = verifyZipCode("95060 1234");  // returns 95060 1234
```

Rethrow an exception

You can use `throw` to rethrow an exception after you catch it. The following example catches an exception with a numeric value and rethrows it if the value is over 50. The rethrown exception propagates up to the enclosing function or to the top level so that the user sees it.

```
try {
   throw n; // throws an exception with a numeric value
} catch (e) {
   if (e <= 50) {
      // statements to handle exceptions 1-50
   } else {
      // cannot handle this exception, so rethrow
      throw e;
   }
}
```

---

try...catch

The **`try...catch` statement** marks a block of statements to try, and specifies a response, should an exception be thrown.

Syntax

```
try {
   try_statements
}
[catch (exception_var_1 if condition_1) { // non-standard
   catch_statements_1
}]
...
[catch (exception_var_2) {
   catch_statements_2
}]
[finally {
   finally_statements
}]
```

`try_statements`

The statements to be executed.

`catch_statements_1`, `catch_statements_2`

Statements that are executed if an exception is thrown in the `try` block.

`exception_var_1`, `exception_var_2`

An identifier to hold an exception object for the associated `catch` clause.

`condition_1`

A conditional expression.

`finally_statements`

Statements that are executed after the `try` statement completes. These statements execute regardless of whether or not an exception was thrown or caught.

Description

The `try` statement consists of a `try` block, which contains one or more statements, and at least one `catch` clause or a `finally` clause, or both. That is, there are three forms of the `try` statement:

1. `try...catch`
2. `try...finally`
3. `try...catch...finally`

A `catch` clause contain statements that specify what to do if an exception is thrown in the `try` block. That is, you want the `try` block to succeed, and if it does not succeed, you want control to pass to the `catch` block. If any statement within the `try` block (or in a function called from within the `try` block) throws an exception, control immediately shifts to the `catch` clause. If no exception is thrown in the `try` block, the `catch` clause is skipped.

The `finally` clause executes after the `try` block and `catch` clause(s) execute but before the statements following the `try` statement. It always executes, regardless of whether or not an exception was thrown or caught.

You can nest one or more `try` statements. If an inner `try` statement does not have a `catch` clause, the enclosing `try` statement's `catch` clause is entered.

You also use the `try` statement to handle JavaScript exceptions. See the JavaScript Guide for more information on JavaScript exceptions.

Unconditional `catch` clause

When a single, unconditional `catch` clause is used, the `catch` block is entered when any exception is thrown. For example, when the exception occurs in the following code, control transfers to the `catch` clause.

```
try {

   throw "myException"; // generates an exception
}
catch (e) {
   // statements to handle any exceptions
   logMyErrors(e); // pass exception object to error handler
}
```

Conditional `catch` clauses

You can also use one or more conditional `catch` clauses to handle specific exceptions. In this case, the appropriate `catch` clause is entered when the specified exception is thrown. In the following example, code in the `try` block can potentially throw three exceptions: **TypeError**, **RangeError**, and **EvalError**. When an exception occurs, control transfers to the appropriate `catch` clause. If the exception is not one of the specified exceptions and an unconditional `catch` clause is found, control transfers to that `catch` clause.

If you use an unconditional `catch` clause with one or more conditional `catch` clauses, the unconditional `catch` clause must be specified last. Otherwise, the unconditional `catch` clause will intercept all types of exception before they can reach the conditional ones.

Reminder: this functionality is not part of the ECMAScript specification.

```
try {
    myroutine(); // may throw three types of exceptions
} catch (e if e instanceof TypeError) {
    // statements to handle TypeError exceptions
```

```
} catch (e if e instanceof RangeError) {
    // statements to handle RangeError exceptions
} catch (e if e instanceof EvalError) {
    // statements to handle EvalError exceptions
} catch (e) {
    // statements to handle any unspecified exceptions
    logMyErrors(e); // pass exception object to error handler
}
```

And here is how to do implement the same "Conditional catch clauses" using only simple JavaScript conforming to the ECMAScript specification (obviously it's more verbose, but works everywhere):

```
try {
    myroutine(); // may throw three types of exceptions
} catch (e) {
    if (e instanceof TypeError) {
        // statements to handle TypeError exceptions
    } else if (e instanceof RangeError) {
        // statements to handle RangeError exceptions
    } else if (e instanceof EvalError) {
        // statements to handle EvalError exceptions
    } else {
       // statements to handle any unspecified exceptions
       logMyErrors(e); // pass exception object to error handler
    }
}
```

The exception identifier

When an exception is thrown in the `try` block, *`exception_var`* (e.g. the `e` in `catch (e)`) holds the value specified by the `throw` statement. You can use this identifier to get information about the exception that was thrown.

This identifier is local to the `catch` clause. That is, it is created when the `catch` clause is entered, and after the `catch` clause finishes executing, the identifier is no longer available.

The `finally` clause

The `finally` clause contains statements to execute after the `try` block and `catch` clause(s) execute, but before the statements following the `try` statement. The `finally` clause executes regardless of whether or not an exception is thrown. If an exception is thrown, the statements in the `finally` clause execute even if no `catch` clause handles the exception.

You can use the `finally` clause to make your script fail gracefully when an exception occurs; for example, you may need to release a resource that your script has tied up. The following example opens a file and then executes statements that use the file (server-side JavaScript allows you to access files). If an exception is thrown while the file is open, the `finally` clause closes the file before the script fails. The code in `finally` also executes upon explicitly returning from `try` or `catch` block.

```
openMyFile()
try {
   // tie up a resource
   writeMyFile(theData);
}
finally {
   closeMyFile(); // always close the resource
}
```

Examples

Nested try-blocks

First let's see what happens with this:

```
try {
  try {
    throw new Error("oops");
  }
  finally {
    console.log("finally");
  }
}
catch (ex) {
  console.error("outer", ex.message);
```

```
}

// Output:
// "finally"
// "outer" "oops"
```

Now, if we already caught the exception in the inner try-block by adding a catch block

```
try {
  try {
    throw new Error("oops");
  }
  catch (ex) {
    console.error("inner", ex.message);
  }
  finally {
    console.log("finally");
  }
}
catch (ex) {
  console.error("outer", ex.message);
}

// Output:
// "inner" "oops"
// "finally"
```

And now, lets re-throw the error.

```
try {
  try {
    throw new Error("oops");
  }
  catch (ex) {
    console.error("inner", ex.message);
    throw ex;
  }
```

```
  finally {
    console.log("finally");
  }
}
catch (ex) {
  console.error("outer", ex.message);
}

// Output:
// "inner" "oops"
// "finally"
// "outer" "oops"
```

Any given exception will be caught only once by the nearest enclosing catch-block, unless it is re-thrown. Of course, any new exceptions raised in the "inner" block (because code in catch-block may do something that throws), will be caught by the "outer" block.

Returning from a finally block

If the `finally` block returns a value, this value becomes the return value of the entire `try-catch-finally` production, regardless of any `return` statements in the `try` and `catch` blocks. This includes exceptions thrown inside of the catch block:

```
try {
  try {
    throw new Error("oops");
  }
  catch (ex) {
    console.error("inner", ex.message);
    throw ex;
  }
  finally {
    console.log("finally");
    return;
  }
}
catch (ex) {
```

```
  console.error("outer", ex.message);
}

// Output:
// "inner" "oops"
// "finally"
```

The outer "oops" is not thrown because of the return in the finally block. The same would apply to any value returned from the catch block.

---

var

The **`variable` statement** declares a variable, optionally initializing it to a value.

Syntax

```
var varname1 [= value1 [, varname2 [, varname3 ... [, varnameN]]]];
```

`varnameN`

Variable name. It can be any legal identifier.

`valueN`

Initial value of the variable. It can be any legal expression.

Description

Variable declarations, wherever they occur, are processed before any code is executed. The scope of a variable declared with `var` is its current *execution context*, which is either the enclosing function or, for variables declared outside any function, global.

Assigning a value to an undeclared variable implicitly creates it as a global variable (it becomes a property of the global object) when the assignment is executed. The differences between declared and undeclared variables are:

1. Declared variables are constrained in the execution context in which they are declared. Undeclared variables are always global.

```
function x() {
  y = 1;   // Throws a ReferenceError in strict mode
```

```
  var z = 2;
}
x();
console.log(y); // logs "1"
console.log(z); // Throws a ReferenceError: z is not defined outside x
```

2. Declared variables are created before any code is executed. Undeclared variables do not exist until the code assigning to them is executed.

```
console.log(a);                // Throws a ReferenceError.
console.log('still going...'); // Never executes.
var a;
console.log(a);                    // logs "undefined" or "" depending on
browser.
console.log('still going...'); // logs "still going...".
```

3. Declared variables are a non-configurable property of their execution context (function or global). Undeclared variables are configurable (e.g. can be deleted).

```
var a = 1;
b = 2;

delete this.a; // Throws a TypeError in strict mode. Fails silently
otherwise.
delete this.b;

console.log(a, b); // Throws a ReferenceError.
// The 'b' property was deleted and no longer exists.
```

Because of these three differences, failure to declare variables will very likely lead to unexpected results. Thus **it is recommended to always declare variables, regardless of whether they are in a function or global scope.** And in ECMAScript 5 strict mode, assigning to an undeclared variable throws an error.

var hoisting

Because variable declarations (and declarations in general) are processed before any code is executed, declaring a variable anywhere in the code is equivalent to declaring it at the top. This also means that a variable can appear to be used before it's declared. This behavior is called "hoisting", as it appears that the variable declaration is moved to the top of the function or global code.

```
bla = 2
var bla;
// ...

// is implicitly understood as:

var bla;
bla = 2;
```

For that reason, it is recommended to always declare variables at the top of their scope (the top of global code and the top of function code) so it's clear which variables are function scoped (local) and which are resolved on the scope chain.

Examples

Declaring and initializing two variables

```
var a = 0, b = 0;
```

Assigning two variables with single string value

```
var a = "A";
var b = a;

// Equivalent to:

var a, b = a = "A";
```

Be mindful of the order:

```
var x = y, y = 'A';
console.log(x + y); // undefinedA
```

Here, x and y are declared before any code is executed, the assignments occur later. At the time "x = y" is evaluated, y exists so no ReferenceError is thrown and its value is 'undefined'. So, x is assigned the undefined value. Then, y is assigned a value of 'A'. Consequently, after the first line, x === undefined && y === 'A', hence the result.

Initialization of several variables

```
var x = 0;

function f(){
  var x = y = 1; // x is declared locally. y is not!
}
f();

console.log(x, y); // 0, 1
// x is the global one as expected
// y leaked outside of the function, though!
```

Implicit globals and outer function scope

Variables that appear to be implicit globals may be references to variables in an outer function scope:

```
var x = 0;  // x is declared global, then assigned a value of 0

console.log(typeof z); // undefined, since z doesn't exist yet

function a() { // when a is called,
  var y = 2;   // y is declared local to function a, then assigned a value
of 2

  console.log(x, y);   // 0 2

  function b() {       // when b is called
    x = 3;  // assigns 3 to existing global x, doesn't create a new global
var
```

```
    y = 4;  // assigns 4 to existing outer y, doesn't create a new global
var
    z = 5;  // creates a new global variable z and assigns a value of 5.
  }         // (Throws a ReferenceError in strict mode.)

  b();     // calling b creates z as a global variable
  console.log(x, y, z);  // 3 4 5
}

a();                    // calling a also calls b

console.log(x, z);      // 3 5
console.log(typeof y); // undefined as y is local to function a
```

function

The **function declaration** defines a function with the specified parameters.

You can also define functions using the **Function** constructor and a **function expression**.

Syntax

```
function name([param,[, param,[..., param]]]) {
   [statements]
}
```

`Name` The function name.

`Param` The name of an argument to be passed to the function. A function can have up to 255 arguments.

`statements`

The statements which comprise the body of the function.

Description

A function created with a function declaration is a `Function` object and has all the properties, methods and behavior of `Function` objects. See **Function** for detailed information on functions.

A function can also be created using an expression (see **function expression**).

By default, functions return `undefined`. To return any other value, the function must have a **return** statement that specifies the value to return.

Conditionally created functions

Functions can be conditionally declared, that is, a function statement can be nested within an `if` statement. Most browsers other than Mozilla will treat such conditional declarations as an unconditional declaration and create the function whether the condition is true or not, see this article for an overview. Therefore they should not be used, for conditional creation use function expressions.

Function declaration hoisting

Function declarations in JavaScript are hoisting the function definition. You can use the function before you declared it:

```
hoisted(); // logs "foo"

function hoisted() {
  console.log("foo");
}
```

Note that **function expressions** are not hoisted:

```
notHoisted(); // TypeError: notHoisted is not a function

var notHoisted = function() {
   console.log("bar");
};
```

Examples

Using `function`

The following code declares a function that returns the total amount of sales, when given the number of units sold of products `a`, `b`, and `c`.

```
function calc_sales(units_a, units_b, units_c) {
   return units_a*79 + units_b * 129 + units_c * 699;
}
```

return

The **`return` statement** ends function execution and specifies a value to be returned to the function caller.

Syntax

```
return [[expression]];
```

`expression`

The expression to return. If omitted, `undefined` is returned instead.

Description

When a `return` statement is called in a function, the execution of this function is stopped. If specified, a given value is returned to the function caller. If the expression is omitted, `undefined` is returned instead. The following return statements all break the function execution:

```
return;
return true;
return false;
return x;
return x + y / 3;
```

Automatic Semicolon Insertion

The `return` statement is affected by automatic semicolon insertion (ASI). No line terminator is allowed between the `return` keyword and the expression.

```
return
a + b;
```

is transformed by ASI into:

```
return;
```

```
a + b;
```

The console will warn "unreachable code after return statement".

Starting with Gecko 40 , a warning is shown in the console if unreachable code is found after a return statement.

Examples

return

The following function returns the square of its argument, `x`, where `x` is a number.

```
function square(x) {
   return x * x;
}
```

Interrupt a function

A function immediately stops at the point where `return` is called.

```
function counter() {
  for (var count = 1; ; count++) {  // infinite loop
    console.log(count + "A"); // until 5
      if (count === 5) {
        return;
      }
      console.log(count + "B");  // until 4
    }
  console.log(count + "C");  // never appears
}

counter();

// Output:
// 1A
// 1B
```

```
// 2A
// 2B
// 3A
// 3B
// 4A
// 4B
// 5A
```

Returning a function

See also the article about Closures.

```
function magic(x) {
  return function calc(x) { return x * 42 };
}

var answer = magic();
answer(1337); // 56154
```

---

do...while

The **`do...while` statement** creates a loop that executes a specified statement until the test condition evaluates to false. The condition is evaluated after executing the statement, resulting in the specified statement executing at least once.

Syntax

```
do
   statement
while (condition);
```

`statement`

A statement that is executed at least once and is re-executed each time the condition evaluates to true. To execute multiple statements within the loop, use a **block** statement ({ ... }) to group those statements.

`condition`

An expression evaluated after each pass through the loop. If `condition` evaluates to true, the `statement` is re-executed. When `condition` evaluates to false, control passes to the statement following the `do...while`.

Examples

Using `do...while`

In the following example, the `do...while` loop iterates at least once and reiterates until `i` is no longer less than 5.

```
var i = 0;
do {
   i += 1;
   console.log(i);
} while (i < 5);
```

---

for

The **for statement** creates a loop that consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by a statement or a set of statements executed in the loop.

Syntax

```
for ([initialization]; [condition]; [final-expression])
   statement
```

`initialization`

An expression (including assignment expressions) or variable declaration. Typically used to initialize a counter variable. This expression may optionally declare new variables with the `var` keyword. These variables are not local to the loop, i.e. they are in the same scope the `for` loop is in. The result of this expression is discarded.

`condition`

An expression to be evaluated before each loop iteration. If this expression evaluates to true, `statement` is executed. This conditional test is optional. If omitted, the condition always

evaluates to true. If the expression evaluates to false, execution skips to the first expression following the `for` construct.

`final-expression`

An expression to be evaluated at the end of each loop iteration. This occurs before the next evaluation of `condition`. Generally used to update or increment the counter variable.

`statement`

A statement that is executed as long as the condition evaluates to true. To execute multiple statements within the loop, use a block statement (`{ ... }`) to group those statements. To execute no statement within the loop, use an empty statement (`;`).

Examples

Using `for`

The following `for` statement starts by declaring the variable `i` and initializing it to `0`. It checks that `i` is less than nine, performs the two succeeding statements, and increments `i` by `1` after each pass through the loop.

```
for (var i = 0; i < 9; i++) {
   console.log(i);
   // more statements
}
```

Optional `for` expressions

All three expressions in the head of the `for` loop are optional.

For example, in the *initialization* block it is not required to initialize variables:

```
var i = 0;
for (; i < 9; i++) {
    console.log(i);
    // more statements
}
```

Like the *initialization* block, the *condition* block is also optional. If you are omitting this expression, you must make sure to break the loop in the body in order to not create an infinite loop.

```
for (var i = 0;; i++) {
   console.log(i);
   if (i > 3) break;
   // more statements
}
```

You can also omit all three blocks. Again, make sure to use a `break` statement to end the loop and also modify (increase) a variable, so that the condition for the break statement is true at some point.

```
var i = 0;

for (;;) {
  if (i > 3) break;
  console.log(i);
  i++;
}
```

Using `for` with an empty statement

The following `for` cycle calculates the offset position of a node in the *[final-expression]* section, and therefore it does not require the use of a `statement` or `block statement` section, an empty statement is used instead.

```
function showOffsetPos (sId) {
  var nLeft = 0, nTop = 0;

  for (var oItNode = document.getElementById(sId); // initialization
       oItNode; // condition
       nLeft += oItNode.offsetLeft, nTop += oItNode.offsetTop, oItNode =
oItNode.offsetParent) // final-expression
       /* empty statement */ ;

  console.log("Offset position of \"" + sId + "\" element:\n left: " +
nLeft + "px;\n top: " + nTop + "px;");
```

```
}

// Example call:

showOffsetPos("content");

// Output:
// "Offset position of "content" element:
// left: 0px;
// top: 153px;"
```

Note: In this case, when you do not use the `statement` section, **a semicolon is put immediately after the declaration of the cycle**.

---

for...in

The **`for...in` statement** iterates over the enumerable properties of an object, in arbitrary order. For each distinct property, statements can be executed.

Syntax

```
for (variable in object) {...
}
```

`variable`

A different property name is assigned to *variable* on each iteration.

`object`

Object whose enumerable properties are iterated.

Description

A `for...in` loop only iterates over enumerable properties. Objects created from builtâ€“in constructors like `Array` and `Object` have inherited nonâ€“enumerable properties from `Object.prototype` and `String.prototype`, such as **String**'s **indexOf()** method or **Object**'s **toString()** method. The loop will iterate over all enumerable properties of the object itself and those the object inherits from its constructor's prototype (properties closer to the object in the prototype chain override prototypes' properties).

Deleted, added or modified properties

A `for...in` loop iterates over the properties of an object in an arbitrary order (see the **delete** operator for more on why one cannot depend on the seeming orderliness of iteration, at least in a cross-browser setting). If a property is modified in one iteration and then visited at a later time, its value in the loop is its value at that later time. A property that is deleted before it has been visited will not be visited later. Properties added to the object over which iteration is occurring may either be visited or omitted from iteration. In general it is best not to add, modify or remove properties from the object during iteration, other than the property currently being visited. There is no guarantee whether or not an added property will be visited, whether a modified property (other than the current one) will be visited before or after it is modified, or whether a deleted property will be visited before it is deleted.

Array iteration and `for...in`

Note: `for...in` should not be used to iterate over an **Array** where the index order is important.

Array indexes are just enumerable properties with integer names and are otherwise identical to general Object properties. There is no guarantee that `for...in` will return the indexes in any particular order and it will return all enumerable properties, including those with nonâ€“integer names and those that are inherited.

Because the order of iteration is implementation-dependent, iterating over an array may not visit elements in a consistent order. Therefore it is better to use a **for** loop with a numeric index (or **Array.prototype.forEach()** or the **for...of** loop) when iterating over arrays where the order of access is important.

Iterating over own properties only

If you only want to consider properties attached to the object itself, and not its prototypes, use **getOwnPropertyNames()** or perform a **hasOwnProperty()** check (**propertyIsEnumerable** can also be used). Alternatively, if you know there won't be any outside code interference, you can extend built-in prototypes with a check method.

Examples

The following function takes as its argument an object. It then iterates over all the object's enumerable properties and returns a string of the property names and their values.

```
var obj = {a:1, b:2, c:3};

for (var prop in obj) {
  console.log("obj." + prop + " = " + obj[prop]);
}

// Output:
// "obj.a = 1"
// "obj.b = 2"
// "obj.c = 3"
```

The following function illustrates the use of **hasOwnProperty()**: the inherited properties are not displayed.

```
var triangle = {a:1, b:2, c:3};

function ColoredTriangle() {
  this.color = "red";
}

ColoredTriangle.prototype = triangle;

var obj = new ColoredTriangle();

for (var prop in obj) {
  if( obj.hasOwnProperty( prop ) ) {
    console.log("obj." + prop + " = " + obj[prop]);
  }
}

// Output:
// "obj.color = red"
```

Compatibility: Initializer expressions

Prior to SpiderMonkey 40 , it was possible to use an initializer expression (`i=0`) in a `for...in` loop:

```
var obj = {a:1, b:2, c:3};
for(var i=0 in obj) {
  console.log(obj[i]);
}
// 1
// 2
// 3
```

This non-standard behavior is now ignored in version 40 and later and will present a **SyntaxError** ("for-in loop head declarations may not have initializers") warning in the console (and).

Other engines like v8 (Chrome), Chakra (IE/Edge), and JSC (WebKit/Safari) are investigating to remove the non-standard behavior as well.

---

for...of

The **`for...of` statement** creates a loop Iterating over iterable objects (including **Array**, **Map**, **Set**, arguments object and so on), invoking a custom iteration hook with statements to be executed for the value of each distinct property.

Syntax

```
for (variable of object) {
  statement
}
```

`variable`

On each iteration a value of a different property is assigned to *variable*.

`object`

Object whose enumerable properties are iterated.

Examples

Difference between `for...of` and `for...in`

The following example shows the difference between a `for...of` loop and a `for...in` loop. While `for...in` iterates over property names, `for...of` iterates over property values:

```
let arr = [3, 5, 7];
arr.foo = "hello";

for (let i in arr) {
   console.log(i); // logs "0", "1", "2", "foo"
}

for (let i of arr) {
   console.log(i); // logs "3", "5", "7"
}
```

Using `Array.prototype.forEach()`

To get the same property values the `for...of` loop would return, you can also use the **Array.prototype.forEach()** method:

```
let arr = [3, 5, 7];
arr.foo = "hello";

arr.forEach(function (element, index) {
    console.log(element); // logs "3", "5", "7"
    console.log(index);   // logs "0", "1", "2"
});

// or with Object.keys()

Object.keys(arr).forEach(function (element, index) {
    console.log(arr[element]); // logs "3", "5", "7", "hello"
    console.log(arr[index]);   // logs "3", "5", "7", undefined
```

```
});
```

Iterating over DOM collections

Iterating over DOM collections like : the following example adds a `read` class to paragraphs that are direct descendants of an article:

```
// Note: This will only work in platforms that have
// implemented NodeList.prototype[Symbol.iterator]
let articleParagraphs = document.querySelectorAll("article > p");

for (let paragraph of articleParagraphs) {
  paragraph.classList.add("read");
}
```

Iterating over generators

You can also iterate over generators:

```
function* fibonacci() { // a generator function
    let [prev, curr] = [0, 1];
    while (true) {
        [prev, curr] = [curr, prev + curr];
        yield curr;
    }
}

for (let n of fibonacci()) {
    console.log(n);
    // truncate the sequence at 1000
    if (n >= 1000) {
        break;
    }
}
```

while

The **while statement** creates a loop that executes a specified statement as long as the test condition evaluates to true. The condition is evaluated before executing the statement.

Syntax

```
while (condition) {
  statement
}
```

`condition`

An expression evaluated before each pass through the loop. If this condition evaluates to true, `statement` is executed. When condition evaluates to false, execution continues with the statement after the `while` loop.

`statement`

A statement that is executed as long as the condition evaluates to true. To execute multiple statements within the loop, use a block statement (`{ ... }`) to group those statements.

Examples

The following `while` loop iterates as long as `n` is less than three.

```
var n = 0;
var x = 0;

while (n < 3) {
  n++;
  x += n;
}
```

Each iteration, the loop increments `n` and adds it to `x`. Therefore, `x` and `n` take on the following values:

- After the first pass: `n` = 1 and `x` = 1
- After the second pass: `n` = 2 and `x` = 3
- After the third pass: `n` = 3 and `x` = 6

After completing the third pass, the condition `n < 3` is no longer true, so the loop terminates.

function

The **`function`** keyword can be used to define a function inside an expression.

Syntax

```
function [name]([param1[, param2[, ..., paramN]]]) {
   statements
}
```

Parameters

`Name` The function name. Can be omitted, in which case the function is *anonymous*. The name is only local to the function body.

`paramN` The name of an argument to be passed to the function.

`statements`

The statements which comprise the body of the function.

Description

A function expression is very similar to and has almost the same syntax as a function statement (see function statement for details). The main difference between a function expression and a function statement is the *function name,* which can be omitted in function expressions to create *anonymous* functions. See also the chapter about functions for more information.

Examples

The following example defines an unnamed function and assigns it to `x`. The function returns the square of its argument:

```
var x = function(y) {
   return y * y;
};
```

Named function expression

If you want to refer to the current function inside the function body, you need to create a named function expression. This name is then local only to the function body (scope). This also avoids using the non-standard `arguments.callee` property.

```
var math = {
  'factorial': function factorial(n) {
    if (n <= 1)
      return 1;
    return n * factorial(n - 1);
  }
};
```

Array

Summary

The JavaScript **`Array`** object is a global object that is used in the construction of arrays; which are high-level, list-like objects.

Create an Array

```
var fruits = ["Apple", "Banana"];

console.log(fruits.length);
// 2
```

Access (index into) an Array item

```
var first = fruits[0];
// Apple

var last = fruits[fruits.length - 1];
// Banana
```

Loop over an Array

```
fruits.forEach(function (item, index, array) {
  console.log(item, index);
});
// Apple 0
// Banana 1
```

Add to the end of an Array

```
var newLength = fruits.push("Orange");
// ["Apple", "Banana", "Orange"]
```

Remove from the end of an Array

```
var last = fruits.pop(); // remove Orange (from the end)
// ["Apple", "Banana"];
```

Remove from the front of an Array

```
var first = fruits.shift(); // remove Apple from the front
// ["Banana"];
```

Add to the front of an Array

```
var newLength = fruits.unshift("Strawberry") // add to the front
// ["Strawberry", "Banana"];
```

Find the index of an item in the Array

```
fruits.push("Mango");
// ["Strawberry", "Banana", "Mango"]

var pos = fruits.indexOf("Banana");
// 1
```

Remove an item by Index Position

```
var removedItem = fruits.splice(pos, 1); // this is how to remove an item
// ["Strawberry", "Mango"]
```

Copy an Array

```
var shallowCopy = fruits.slice(); // this is how to make a copy
// ["Strawberry", "Mango"]
```

Syntax

```
[element0, element1, ..., elementN]
new Array(element0, element1[, ...[, elementN]])
new Array(arrayLength)
elementN
```

A JavaScript array is initialized with the given elements, except in the case where a single argument is passed to the `Array` constructor and that argument is a number. (See below.) Note that this special case only applies to JavaScript arrays created with the `Array` constructor, not array literals created with the bracket syntax.

`arrayLength`

If the only argument passed to the `Array` constructor is an integer between 0 and 2^{32}-1 (inclusive), this returns a new JavaScript array with length set to that number. If the argument is any other number, a **RangeError** exception is thrown.

Description

Arrays are list-like objects whose prototype has methods to perform traversal and mutation operations. Neither the length of a JavaScript array nor the types of its elements are fixed. Since an array's size length grow or shrink at any time, JavaScript arrays are not guaranteed to be dense. In general, these are convenient characteristics; but if these features are not desirable for your particular use, you might consider using typed arrays.

Some people think that you shouldn't use an array as an associative array. In any case, you can use plain **objects** instead, although doing so comes with its own caveats. See the post Lightweight JavaScript dictionaries with arbitrary keys as an example.

Accessing array elements

JavaScript arrays are zero-indexed: the first element of an array is at index `0`, and the last element is at the index equal to the value of the array's **length** property minus 1.

```
var arr = ['this is the first element', 'this is the second element'];
console.log(arr[0]);              // logs 'this is the first element'
console.log(arr[1]);              // logs 'this is the second element'
console.log(arr[arr.length - 1]); // logs 'this is the second element'
```

Array elements are object properties in the same way that `toString` is a property, but trying to access an element of an array as follows throws a syntax error, because the property name is not valid:

```
console.log(arr.0); // a syntax error
```

There is nothing special about JavaScript arrays and the properties that cause this. JavaScript properties that begin with a digit cannot be referenced with dot notation; and must be accessed using bracket notation. For example, if you had an object with a property named `'3d'`, it can only be referenced using bracket notation. E.g.:

```
var years = [1950, 1960, 1970, 1980, 1990, 2000, 2010];
console.log(years.0);   // a syntax error
console.log(years[0]);  // works properly
renderer.3d.setTexture(model, 'character.png');     // a syntax error
renderer['3d'].setTexture(model, 'character.png');  // works properly
```

Note that in the `3d` example, `'3d'` had to be quoted. It's possible to quote the JavaScript array indexes as well (e.g., `years['2']` instead of `years[2]`), although it's not necessary. The 2 in `years[2]` is coerced into a string by the JavaScript engine through an implicit `toString` conversion. It is for this reason that `'2'` and `'02'` would refer to two different slots on the `years` object and the following example could be `true`:

```
console.log(years['2'] != years['02']);
```

Similarly, object properties which happen to be reserved words(!) can only be accessed as string literals in bracket notation(but it can be accessed by dot notation in firefox 40.0a2 at least):

```
var promise = {
  'var'  : 'text',
  'array': [1, 2, 3, 4]
};
```

```
console.log(promise['array']);
```

Relationship between `length` and numerical properties

A JavaScript array's **length** property and numerical properties are connected. Several of the built-in array methods (e.g., **join**, **slice**, **indexOf**, etc.) take into account the value of an array's **length** property when they're called. Other methods (e.g., **push**, **splice**, etc.) also result in updates to an array's **length** property.

```
var fruits = [];
fruits.push('banana', 'apple', 'peach');

console.log(fruits.length); // 3
```

When setting a property on a JavaScript array when the property is a valid array index and that index is outside the current bounds of the array, the engine will update the array's **length** property accordingly:

```
fruits[5] = 'mango';
console.log(fruits[5]); // 'mango'
console.log(Object.keys(fruits));  // ['0', '1', '2', '5']
console.log(fruits.length); // 6
```

Increasing the **length**.

```
fruits.length = 10;
console.log(Object.keys(fruits)); // ['0', '1', '2', '5']
console.log(fruits.length); // 10
```

Decreasing the **length** property does, however, delete elements.

```
fruits.length = 2;
console.log(Object.keys(fruits)); // ['0', '1']
console.log(fruits.length); // 2
```

This is explained further on the **Array.length** page.

Creating an array using the result of a match

The result of a match between a regular expression and a string can create a JavaScript array. This array has properties and elements which provide information about the match. Such an array is returned by **RegExp.exec**, **String.match**, and **String.replace**. To help explain these properties and elements, look at the following example and then refer to the table below:

```
// Match one d followed by one or more b's followed by one d
// Remember matched b's and the following d
// Ignore case

var myRe = /d(b+)(d)/i;
var myArray = myRe.exec('cdbBdbsbz');
```

The properties and elements returned from this match are as follows:

Property/Element	Description	Example
`input`	A read-only property that reflects the original string against which the regular expression was matched.	cdbBdbsbz
`index`	A read-only property that is the zero-based index of the match in the string.	1
`[0]`	A read-only element that specifies the last matched characters.	dbBd
`[1], ...[n]`	Read-only elements that specify the parenthesized substring matches, if included in the regular expression. The number of possible parenthesized substrings is unlimited.	[1]: bB [2]: d

Properties

`Array.length`

The `Array` constructor's length property whose value is 1.

Array.prototype

Allows the addition of properties to all array objects.

Methods

Array.isArray()

Returns true if a variable is an array, if not false.

`Array` instances

All `Array` instances inherit from **Array.prototype**. The prototype object of the `Array` constructor can be modified to affect all `Array` instances.

Properties

Methods

Mutator methods

Accessor methods

Iteration methods

`Array` generic methods

Array generics are non-standard, deprecated and will get removed near future. Note that you can not rely on them cross-browser. However, there is a shim available on GitHub.

Sometimes you would like to apply array methods to strings or other array-like objects (such as function **arguments**). By doing this, you treat a string as an array of characters (or otherwise treat a non-array as an array). For example, in order to check that every character in the variable *str* is a letter, you would write:

```
function isLetter(character) {
  return character >= 'a' && character <= 'z';
}

if (Array.prototype.every.call(str, isLetter)) {
  console.log("The string '" + str + "' contains only letters!");
}
```

This notation is rather wasteful and JavaScript 1.6 introduced a generic shorthand:

```
if (Array.every(str, isLetter)) {
  console.log("The string '" + str + "' contains only letters!");
}
```

Generics are also available on **String**.

These are **not** part of ECMAScript standards (though the ES6 **Array.from()** can be used to achieve this). The following is a shim to allow its use in all browsers:

```
// Assumes Array extras already present (one may use polyfills for these
as well)
(function() {
  'use strict';

  var i,
    // We could also build the array of methods with the following, but
the
    //   getOwnPropertyNames() method is non-shimable:
    // Object.getOwnPropertyNames(Array).filter(function(methodName) {
    //   return typeof Array[methodName] === 'function'
    // });
    methods = [
      'join', 'reverse', 'sort', 'push', 'pop', 'shift', 'unshift',
      'splice', 'concat', 'slice', 'indexOf', 'lastIndexOf',
      'forEach', 'map', 'reduce', 'reduceRight', 'filter',
      'some', 'every', 'find', 'findIndex', 'entries', 'keys',
      'values', 'copyWithin', 'includes'
    ],
    methodCount = methods.length,
    assignArrayGeneric = function(methodName) {
      if (!Array[methodName]) {
        var method = Array.prototype[methodName];
        if (typeof method === 'function') {
          Array[methodName] = function() {
            return method.call.apply(method, arguments);
```

```
          };
        }
      }
    };

  for (i = 0; i < methodCount; i++) {
    assignArrayGeneric(methods[i]);
  }
}());
```

Examples

Creating an array

The following example creates an array, `msgArray`, with a length of 0, then assigns values to `msgArray[0]` and `msgArray[99]`, changing the length of the array to 100.

```
var msgArray = [];
msgArray[0] = 'Hello';
msgArray[99] = 'world';

if (msgArray.length === 100) {
  console.log('The length is 100.');
}
```

Creating a two-dimensional array

The following creates a chess board as a two dimensional array of strings. The first move is made by copying the 'p' in (6,4) to (4,4). The old position (6,4) is made blank.

```
var board = [
  ['R','N','B','Q','K','B','N','R'],
  ['P','P','P','P','P','P','P','P'],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' ',' '],
```

```
  ['p','p','p','p','p','p','p','p'],
  ['r','n','b','q','k','b','n','r'] ];

console.log(board.join('\n') + '\n\n');

// Move King's Pawn forward 2
board[4][4] = board[6][4];
board[6][4] = ' ';
console.log(board.join('\n'));
```

Here is the output:

```
R,N,B,Q,K,B,N,R
P,P,P,P,P,P,P,P
 , , , , , , ,
 , , , , , , ,
 , , , , , , ,
 , , , , , , ,
p,p,p,p,p,p,p,p
r,n,b,q,k,b,n,r

R,N,B,Q,K,B,N,R
P,P,P,P,P,P,P,P
 , , , , , , ,
 , , , , , , ,
 , , , ,p, , ,
 , , , , , , ,
p,p,p,p, ,p,p,p
r,n,b,q,k,b,n,r
```

Object initializer

Objects can be initialized using `new Object()`, `Object.create()`, or using the *literal* notation (*initializer* notation). An object initializer is a list of zero or more pairs of property names and associated values of an object, enclosed in curly braces (`{}`).

Syntax

```
var o = {};
var o = { a: "foo", b: 42, c: {} };

var a = "foo", b = 42, c = {};
var o = { a: a, b: b, c: c };

var o = {
  property: function ([parameters]) {},
  get property() {},
  set property(value) {},
};
```

New notations in ECMAScript 6

Please see the compatibility table for support for these notations. In non-supporting environments, these notations will lead to syntax errors.

```
// Shorthand property names (ES6)
var a = "foo", b = 42, c = {};
var o = { a, b, c };

// Shorthand method names (ES6)
var o = {
  property([parameters]) {},
  get property() {},
  set property(value) {},
  * generator() {}
};

// Computed property names (ES6)
var prop = "foo";
var o = {
  [prop]: "hey",
  ["b" + "ar"]: "there",
};
```

Description

An object initializer is an expression that describes the initialization of an **Object**. Objects consist of *properties*, which are used to describe an object. Values of object properties can either contain primitive data types or other objects.

Creating objects

An empty object with no properties can be created like this:

```
var object = {};
```

However, the advantage of the *literal* or *initializer* notation is, that you are able to quickly create objects with properties inside the curly braces. You simply notate a list of key: value pairs delimited by comma. The following code creates an object with three properties and the keys are "foo", "age" and "baz". The values of these keys are a string "bar", a number 42 and the third property has another object as its value.

```
var object = {
  foo: "bar",
  age: 42,
  baz: { myProp: 12 },
}
```

Accessing properties

Once you have created an object, you might want to read or change them. Object properties can be accessed by using the dot notation or the bracket notation. See property accessors for detailed information.

```
object.foo; // "bar"
object["age"]; // 42

object.foo = "baz";
```

Property definitions

We have already learned how to notate properties using the initializer syntax. Oftentimes, there are variables in your code that you would like to put into an object. You will see code like this:

```
var a = "foo",
    b = 42,
    c = {};

var o = {
  a: a,
  b: b,
  c: c
};
```

With ECMAScript 6, there is a shorter notation available to achieve the same:

```
var a = "foo",
    b = 42,
    c = {};

// Shorthand property names (ES6)
var o = { a, b, c };
```

Duplicate property names

When using the same name for your properties, the second property will overwrite the first.

```
var a = {x: 1, x: 2};
console.log(a); // { x: 2}
```

In ECMAScript 5 strict mode code, duplicate property names were considered a **SyntaxError**. With the introduction of computed property names making duplication possible at runtime, ECMAScript 6 has removed this restriction.

```
function haveES6DuplicatePropertySemantics(){
  "use strict";
  try {
    ({ prop: 1, prop: 2 });
```

```
    // No error thrown, duplicate property names allowed in strict mode
    return true;
  } catch (e) {
    // Error thrown, duplicates prohibited in strict mode
    return false;
  }
}
```

Method definitions

A property of an object can also refer to a function or a getter or setter method.

```
var o = {
  property: function ([parameters]) {},
  get property() {},
  set property(value) {},
};
```

In ECMAScript 6, a shorthand notation is available, so that the keyword "function" is no longer necessary.

```
// Shorthand method names (ES6)
var o = {
  property([parameters]) {},
  get property() {},
  set property(value) {},
  * generator() {}
};
```

In ECMAScript 6 There is a way to concisely define properties whose values are generator functions:

```
var o = {
  * generator() {
    ...........
  }
};
```

In ECMAScript 5 you would write it like this (but note that ES5 has no generators):

```
var o = {
  generatorMethod: function *() {
    ..........
  }
};
```

For more information and examples about methods, see method definitions.

Computed property names

Starting with ECMAScript 6, the object initializer syntax also supports computed property names. That allows you to put an expression in brackets `[]`, that will be computed as the property name. This is symmetrically to the bracket notation of the property accessor syntax, which you might have used to read and set properties already. Now you can use the same syntax in object literals, too:

```
// Computed property names (ES6)
var i = 0;
var a = {
  ["foo" + ++i]: i,
  ["foo" + ++i]: i,
  ["foo" + ++i]: i
};

console.log(a.foo1); // 1
console.log(a.foo2); // 2
console.log(a.foo3); // 3

var param = 'size';
var config = {
  [param]: 12,
  ["mobile" + param.charAt(0).toUpperCase() + param.slice(1)]: 4
};

console.log(config); // { size: 12, mobileSize: 4 }
```

Prototype mutation

A property definition of the form `__proto__: value` or `"__proto__": value` does not create a property with the name `__proto__`. Instead, if the provided value is an object or `null`, it changes the `[[Prototype]]` of the created object to that value. (If the value is not an object or null, the object is not changed.)

```
var obj1 = {};
assert(Object.getPrototypeOf(obj1) === Object.prototype);

var obj2 = { __proto__: null };
assert(Object.getPrototypeOf(obj2) === null);

var protoObj = {};
var obj3 = { "__proto__": protoObj };
assert(Object.getPrototypeOf(obj3) === protoObj);

var obj4 = { __proto__: "not an object or null" };
assert(Object.getPrototypeOf(obj4) === Object.prototype);
assert(!obj4.hasOwnProperty("__proto__"));
```

Only a single prototype mutation is permitted in an object literal: multiple prototype mutations are a syntax error.

Property definitions that do not use "colon" notation are not prototype mutations: they are property definitions that behave identically to similar definitions using any other name.

```
var __proto__ = "variable";

var obj1 = { __proto__ };
assert(Object.getPrototypeOf(obj1) === Object.prototype);
assert(obj1.hasOwnProperty("__proto__"));
assert(obj1.__proto__ === "variable");

var obj2 = { __proto__() { return "hello"; } };
assert(obj2.__proto__() === "hello");
```

```
var obj3 = { ["__prot" + "o__"]: 17 };
assert(obj3.__proto__ === 17);
```

Object literal notation vs JSON

The object literal notation is not the same as the **J**ava**S**cript **O**bject **N**otation (JSON). Although they look similar, there are differences between them:

- JSON permits *only* property definition using `"property": value` syntax. The property name must be double-quoted, and the definition cannot be a shorthand.
- In JSON the values can only be strings, numbers, arrays, `true`, `false`, `null`, or another (JSON) object.
- A function value (see "Methods" above) can not be assigned to a value in JSON.
- Objects like **Date** will be a string after **JSON.parse()**.
- **JSON.parse()** will reject computed property names and an error will be thrown.

RegExp

The **`RegExp`** constructor creates a regular expression object for matching text with a pattern.

For an introduction to regular expressions, read the Regular Expressions chapter in the JavaScript Guide.

Constructor

Literal and constructor notations are possible:

```
/pattern/flags
new RegExp(pattern[, flags])
```

Parameters

`pattern`

The text of the regular expression.

`Flags` If specified, flags can have any combination of the following values:

`g`

global match

`i`

ignore case

`m`

multiline; treat beginning and end characters (^ and $) as working over multiple lines (i.e., match the beginning or end of *each* line (delimited by \n or \r), not only the very beginning or end of the whole input string)

Description

There are 2 ways to create a `RegExp` object: a literal notation and a constructor. To indicate strings, the parameters to the literal notation do not use quotation marks while the parameters to the constructor function do use quotation marks. So the following expressions create the same regular expression:

```
/ab+c/i;
new RegExp('ab+c', 'i');
new RegExp(/ab+c/, 'i');
```

The literal notation provides compilation of the regular expression when the expression is evaluated. Use literal notation when the regular expression will remain constant. For example, if you use literal notation to construct a regular expression used in a loop, the regular expression won't be recompiled on each iteration.

The constructor of the regular expression object, for example, `new RegExp('ab+c')`, provides runtime compilation of the regular expression. Use the constructor function when you know the regular expression pattern will be changing, or you don't know the pattern and are getting it from another source, such as user input.

Starting with ECMAScript 6, `new RegExp(/ab+c/, 'i')` no longer throws a **TypeError** ("can't supply flags when constructing one RegExp from another") when the first argument is a `RegExp` and the second `flags` argument is present. A new `RegExp` from the arguments is created instead.

When using the constructor function, the normal string escape rules (preceding special characters with \ when included in a string) are necessary. For example, the following are equivalent:

```
var re = /\w+/;
var re = new RegExp('\\w+');
```

Special characters meaning in regular expressions

- Character Classes
- Character Sets
- Boundaries
- Grouping and back references
- Quantifiers

Character Classes

Character	Meaning
.	(The dot, the decimal point) matches any single character *except* line terminators: `\n`, `\r`, `\u2028` or `\u2029`. Inside character class, the dot loses its special meaning and matches a literal dot. Note that the `m` multiline flag doesn't change the dot behavior. So to match a pattern across multiple lines, the character set `[^]` can be used (if you don't mean an old version of IE, of course), it will match any character including newlines. For example, `/.y/` matches "my" and "ay", but not "yes", in "yes make my day".
`\d`	Matches a digit character in the basic Latin alphabet. Equivalent to `[0-9]`. For example, `/\d/` or `/[0-9]/` matches "2" in "B2 is the suite number".
`\D`	Matches any character that is not a digit in the basic Latin alphabet. Equivalent to `[^0-9]`. For example, `/\D/` or `/[^0-9]/` matches "B" in "B2 is the suite number".

`\w`	Matches any alphanumeric character from the basic Latin alphabet, including the underscore. Equivalent to `[A-Za-z0-9_]`. For example, `/\w/` matches "a" in "apple", "5" in "$5.28", and "3" in "3D".
`\W`	Matches any character that is not a word character from the basic Latin alphabet. Equivalent to `[^A-Za-z0-9_]`. For example, `/\W/` or `/[^A-Za-z0-9_]/` matches "%" in "50%".
`\s`	Matches a single white space character, including space, tab, form feed, line feed and other Unicode spaces. Equivalent to `[ \f\n\r\t\vâ€‹\u00a0\u1680â€‹\u180e\u2000â€‹-\u200aâ€‹\u2028\u2029\u202f\u205fâ€‹\u3000\ufeff]`. For example, `/\s\w*/` matches " bar" in "foo bar".
`\S`	Matches a single character other than white space. Equivalent to `[^ \f\n\r\t\vâ€‹\u00a0\u1680â€‹\u180e\u2000â€‹-\u200aâ€‹\u2028\u2029\u202f\u205fâ€‹\u3000\ufeff]`. For example, `/\S\w*/` matches "foo" in "foo bar".
`\t`	Matches a horizontal tab.
`\r`	Matches a carriage return.
`\n`	Matches a linefeed.
`\v`	Matches a vertical tab.
`\f`	Matches a form-feed.
`[\b]`	Matches a backspace. (Not to be confused with `\b`)
`\0`	Matches a NUL character. Do not follow this with another digit.
`\c`*X*	Where *x* is a letter from A - Z. Matches a control character in a string.

For example, `/\cM/` matches control-M in a string.

`\x`*`hh`* Matches the character with the code *`hh`* (two hexadecimal digits).

`\u`*`hhhh`* Matches the character with the Unicode value *`hhhh`* (four hexadecimal digits).

For characters that are usually treated literally, indicates that the next character is special and not to be interpreted literally.

For example, `/b/` matches the character "b". By placing a backslash in front of "b", that is by using `/\b/`, the character becomes special to mean match a word boundary.

`\` *or*

For characters that are usually treated specially, indicates that the next character is not special and should be interpreted literally.

For example, "*" is a special character that means 0 or more occurrences of the preceding character should be matched; for example, `/a*/` means match 0 or more "a"s. To match `*` literally, precede it with a backslash; for example, `/a\*/` matches "a*".

Character Sets

Character	**Meaning**
`[xyz]`	A character set. Matches any one of the enclosed characters. You can specify a range of characters by using a hyphen. For example, `[abcd]` is the same as `[a-d]`. They match the "b" in "brisket" and the "c" in "chop".
`[^xyz]`	A negated or complemented character set. That is, it matches anything that is not enclosed in the brackets. You can specify a range of characters by using a hyphen.

For example, `[^abc]` is the same as `[^a-c]`. They initially match "o" in "bacon" and "h" in "chop".

Alternation

Character	Meaning
x|*y*	Matches either *x* or *y*. For example, `/green|red/` matches "green" in "green apple" and "red" in "red apple".

Boundaries

Character	Meaning
^	Matches beginning of input. If the multiline flag is set to true, also matches immediately after a line break character. For example, `/^A/` does not match the "A" in "an A", but does match the first "A" in "An A".
$	Matches end of input. If the multiline flag is set to true, also matches immediately before a line break character. For example, `/t$/` does not match the "t" in "eater", but does match it in "eat".
\b	Matches a zero-width word boundary, such as between a letter and a space. (Not to be confused with `[\b]`) For example, `/\bno/` matches the "no" in "at noon"; `/ly\b/` matches the "ly" in "possibly yesterday".
\B	Matches a zero-width non-word boundary, such as between two letters or between two spaces. For example, `/\Bon/` matches "on" in "at noon", and `/ye\B/` matches "ye" in "possibly yesterday".

Grouping and back references

Character	Meaning
`(x)`	Matches *x* and remembers the match. These are called capturing groups. For example, `/(foo)/` matches and remembers "foo" in "foo bar". The capturing groups are numbered according to the order of left parentheses of capturing groups, starting from 1. The matched substring can be recalled from the resulting array's elements `[1], ..., [n]` or from the predefined `RegExp` object's properties `$1, ..., $9`. Capturing groups have a performance penalty. If you don't need the matched substring to be recalled, prefer non-capturing parentheses (see below).
`\n`	Where *n* is a positive integer. A back reference to the last substring matching the n parenthetical in the regular expression (counting left parentheses). For example, `/apple(,)\sorange\1/` matches "apple, orange," in "apple, orange, cherry, peach". A more complete example follows this table.
`(?:x)`	Matches *x* but does not remember the match. These are called non-capturing groups. The matched substring can not be recalled from the resulting array's elements `[1], ..., [n]` or from the predefined `RegExp` object's properties `$1, ..., $9`.

Quantifiers

Character	Meaning
`x*`	Matches the preceding item *x* 0 or more times. For example, `/bo*/` matches "boooo" in "A ghost booooed" and "b" in "A bird warbled", but nothing in "A goat grunted".
`x+`	Matches the preceding item *x* 1 or more times. Equivalent to `{1,}`.

For example, `/a+/` matches the "a" in "candy" and all the "a"'s in "caaaaaaandy".

`x*?`

`x+?`

Matches the preceding item *x* like `*` and `+` from above, however the match is the smallest possible match.

For example, `/".*?"/` matches '"foo"' in '"foo" "bar"' and does not match '"foo" "bar"' as without the `?` behind the `*`.

`x?`

Matches the preceding item *x* 0 or 1 time.

For example, `/e?le?/` matches the "el" in "angel" and the "le" in "angle."

`x{n}`

If used immediately after any of the quantifiers `*`, `+`, `?`, or `{}`, makes the quantifier non-greedy (matching the minimum number of times), as opposed to the default, which is greedy (matching the maximum number of times).

Where *n* is a positive integer. Matches exactly *n* occurrences of the preceding item *x*.

For example, `/a{2}/` doesn't match the "a" in "candy", but it matches all of the "a"'s in "caandy", and the first two "a"'s in "caaandy".

`x{n,}`

Where *n* is a positive integer. Matches at least *n* occurrences of the preceding item *x*.

For example, `/a{2,}/` doesn't match the "a" in "candy", but matches all of the a's in "caandy" and in "caaaaaaandy".

`x{n,m}`

Where *n* and *m* are positive integers. Matches at least *n* and at most *m* occurrences of the preceding item *x*.

For example, `/a{1,3}/` matches nothing in "cndy", the "a" in "candy", the two "a"'s in "caandy", and the first three "a"'s in "caaaaaaandy". Notice that when matching "caaaaaaandy", the match is "aaa", even though the original string had more "a"'s in it.

Assertions

Character	Meaning
x(?=*y*)	Matches *x* only if *x* is followed by *y*. For example, /Jack(?=Sprat)/ matches "Jack" only if it is followed by "Sprat". /Jack(?=Sprat|Frost)/ matches "Jack" only if it is followed by "Sprat" or "Frost". However, neither "Sprat" nor "Frost" is part of the match results.
x(?!*y*)	Matches *x* only if *x* is not followed by *y*. For example, /\d+(?!\.)/ matches a number only if it is not followed by a decimal point. /\d+(?!\.)/.exec('3.141') matches "141" but not "3.141".

Properties

RegExp.prototype

Allows the addition of properties to all objects.

`RegExp.length`

The value of `RegExp.length` is 2.

RegExp.lastIndex

The index at which to start the next match.

Methods

The global `RegExp` object has no methods of its own, however, it does inherit some methods through the prototype chain.

`RegExp` prototype objects and instances

Properties

Methods

Examples

Using a regular expression to change data format

The following script uses the **replace()** method of the **String** instance to match a name in the format *first last* and output it in the format *last, first*. In the replacement text, the script uses `$1` and `$2` to indicate the results of the corresponding matching parentheses in the regular expression pattern.

```
var re = /(\w+)\s(\w+)/;
var str = 'John Smith';
var newstr = str.replace(re, '$2, $1');
console.log(newstr);
```

This displays "Smith, John".

Using regular expression to split lines with different line endings/ends of line/line breaks

The default line ending varies depending on the platform (Unix, Windows, etc.). The line splitting provided in this example works on all platforms.

```
var text = 'Some text\nAnd some more\r\nAnd yet\rThis is the end';
var lines = text.split(/\r\n|\r|\n/);
console.log(lines); // logs [ 'Some text', 'And some more', 'And yet',
'This is the end' ]
```

Note that the order of the patterns in the regular expression matters.

Using regular expression on multiple lines

```
var s = 'Please yes\nmake my day!';
s.match(/yes.*day/);
// Returns null
s.match(/yes[^]*day/);
// Returns 'yes\nmake my day'
```

Using a regular expression with the "sticky" flag

This example demonstrates how one could use the sticky flag on regular expressions to match individual lines of multiline input.

```
var text = 'First line\nSecond line';
var regex = /(\S+) line\n?/y;

var match = regex.exec(text);
console.log(match[1]);         // logs 'First'
console.log(regex.lastIndex); // logs '11'

var match2 = regex.exec(text);
console.log(match2[1]);        // logs 'Second'
console.log(regex.lastIndex); // logs '22'

var match3 = regex.exec(text);
console.log(match3 === null); // logs 'true'
```

One can test at run-time whether the sticky flag is supported, using `try { â€¦ } catch { â€¦ }`. For this, either an `eval(â€¦)` expression or the `RegExp(`*`regex-string, flags-string`*`)` syntax must be used (since the */regex/flags* notation is processed at compile-time, so throws an exception before the `catch` block is encountered). For example:

```
var supports_sticky;
try { RegExp('', 'y'); supports_sticky = true; }
catch(e) { supports_sticky = false; }
console.log(supports_sticky); // logs 'true'
```

Regular expression and Unicode characters

As mentioned above, `\w` or `\W` only matches ASCII based characters; for example, "a" to "z", "A" to "Z", "0" to "9" and "_". To match characters from other languages such as Cyrillic or Hebrew, use `\uhhhh`, where "hhhh" is the character's Unicode value in hexadecimal. This example demonstrates how one can separate out Unicode characters from a word.

```
var text = 'ÐžÐ±Ñ€Ð°Ð·ÐµÑ† text Ð½Ð° Ñ€ÑƒÑ Ñ Ð°Ð¾Ð¼ Ñ Ð·Ñ‹Ð°Ðµ';
```

```
var regex = /[\u0400-\u04FF]+/g;

var match = regex.exec(text);
console.log(match[0]);        // logs 'ДžÐ±Ñ€Ð°Ð·ÐµÑ†'
console.log(regex.lastIndex); // logs '7'

var match2 = regex.exec(text);
console.log(match2[0]);       // logs 'Ð½Ð°' [did not log 'text']
console.log(regex.lastIndex); // logs '15'

// and so on
```

Here's an external resource for getting the complete Unicode block range for different scripts: Regexp-unicode-block.

Extracting sub-domain name from URL

```
var url = 'http://xxx.domain.com';
console.log(/[^.]+/.exec(url)[0].substr(7)); // logs 'xxx'
```

[1] Behind a flag.

Gecko-specific notes

Starting with Gecko 34 , in the case of a capturing group with quantifiers preventing its exercise, the matched text for a capturing group is now `undefined` instead of an empty string:

```
// Firefox 33 or older
'x'.replace(/x(.)?/g, function(m, group) {
  console.log("'group:" + group + "'");
}); // 'group:'

// Firefox 34 or newer
'x'.replace(/x(.)?/g, function(m, group) {
  console.log("'group:" + group + "'");
}); // 'group:undefined'
```

Note that due to web compatibility, `RegExp.$N` will still return an empty string instead of `undefined` ().

Grouping

The grouping operator `( )` controls the precedence of evaluation in expressions.

Syntax

```
( )
```

Description

The grouping operator consists of a pair of parentheses around an expression or sub-expression to override the normal operator precedence so that expressions with lower precedence can be evaluated before an expression with higher priority.

Examples

Overriding multiplication and division first, then addition and subtraction to evaluate addition first.

```
var a = 1;
var b = 2;
var c = 3;

// default precedence
a + b * c     // 7
// evaluated by default like this
a + (b * c)   // 7

// now overriding precedence
// addition before multiplication
(a + b) * c   // 9

// which is equivalent to
a * c + b * c // 9
```

Property accessors

Property accessors provide access to an object's properties by using the dot notation or the bracket notation.

Syntax

```
object.property
object["property"]
```

Description

One can think of an object as an *associative array* (a.k.a. *map*, *dictionary*, *hash*, *lookup table*). The *keys* in this array are the names of the object's properties. It's typical when speaking of an object's properties to make a distinction between properties and methods. However, the property/method distinction is little more than a convention. A method is simply a property that can be called, for example if it has a reference to a Function instance as its value.

There are two ways to access properties: dot notation and bracket notation.

Dot notation

```
get = object.property;
object.property = set;
```

In this code, `property` must be a valid JavaScript identifier, i.e. a sequence of alphanumerical characters, also including the underscore ("_") and dollar sign ("$"), that cannot start with a number. For example, `object.$1` is valid, while `object.1` is not.

```
document.createElement('pre');
```

Here, the method named "createElement" is retrieved from `document` and is called.

Bracket notation

```
get = object[property_name];
object[property_name] = set;
```

`property_name` is a string. The string does not have to be a valid identifier; it can have any value, e.g. "1foo", "!bar!", or even " " (a space).

```
document['createElement']('pre');
```

This does the exact same thing as the previous example.

Property names

Property names must be strings. This means that non-string objects cannot be used as keys in the object. Any non-string object, including a number, is typecasted into a string via the toString method.

```
var object = {};
object['1'] = 'value';
console.log(object[1]);
```

This outputs "value", since 1 is type-casted into '1'.

```
var foo = {unique_prop: 1}, bar = {unique_prop: 2}, object = {};
object[foo] = 'value';
console.log(object[bar]);
```

This also outputs "value", since both foo and bar are converted to the same string. In the SpiderMonkey JavaScript engine, this string would be "['object Object']".

Method binding

A method is not bound to the object that it is a method of. Specifically, `this` is not fixed in a method, i.e., `this` does not necessarily refer to an object containing the method. `this` is instead "passed" by the function call. See method binding.

Note on `eval`

JavaScript novices often make the mistake of using eval where the bracket notation can be used instead. For example, the following syntax is often seen in many scripts.

```
x = eval('document.forms.form_name.elements.' + strFormControl +
'.value');
```

`eval` is slow and should be avoided whenever possible. Also, `strFormControl` would have to hold an identifier, which is not required for names and IDs of form controls. It is better to use bracket notation instead:

```
x = document.forms["form_name"].elements[strFormControl].value;
```

---

new

The **`new` operator** creates an instance of a user-defined object type or of one of the built-in object types that has a constructor function.

Syntax

```
new constructor[([arguments])]
```

Parameters

`constructor`

A function that specifies the type of the object instance.

`arguments`

A list of values that the `constructor` will be called with.

Description

Creating a user-defined object requires two steps:

1. Define the object type by writing a function.
2. Create an instance of the object with `new`.

To define an object type, create a function for the object type that specifies its name and properties. An object can have a property that is itself another object. See the examples below.

When the code `new Foo(...)` is executed, the following things happen:

1. A new object is created, inheriting from `Foo.prototype`.

2. The constructor function *Foo* is called with the specified arguments and `this` bound to the newly created object. `new Foo` is equivalent to `new Foo()`, i.e. if no argument list is specified, *Foo* is called without arguments.
3. The object returned by the constructor function becomes the result of the whole `new` expression. If the constructor function doesn't explicitly return an object, the object created in step 1 is used instead. (Normally constructors don't return a value, but they can choose to do so if they want to override the normal object creation process.)

You can always add a property to a previously defined object. For example, the statement `car1.color = "black"` adds a property `color` to `car1`, and assigns it a value of "`black`". However, this does not affect any other objects. To add the new property to all objects of the same type, you must add the property to the definition of the `Car` object type.

You can add a shared property to a previously defined object type by using the `Function.prototype` property. This defines a property that is shared by all objects created with that function, rather than by just one instance of the object type. The following code adds a color property with value `null` to all objects of type `car`, and then overwrites that value with the string "`black`" only in the instance object `car1`. For more information, see prototype.

```
function Car() {}
car1 = new Car();

console.log(car1.color);    // undefined

Car.prototype.color = null;
console.log(car1.color);    // null

car1.color = "black";
console.log(car1.color);   // black
```

Examples

Object type and object instance

Suppose you want to create an object type for cars. You want this type of object to be called `car`, and you want it to have properties for make, model, and year. To do this, you would write the following function:

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
```

Now you can create an object called `mycar` as follows:

```
var mycar = new Car("Eagle", "Talon TSi", 1993);
```

This statement creates `mycar` and assigns it the specified values for its properties. Then the value of `mycar.make` is the string "Eagle", `mycar.year` is the integer 1993, and so on.

You can create any number of `car` objects by calls to `new`. For example:

```
var kenscar = new Car("Nissan", "300ZX", 1992);
```

Object property that is itself another object

Suppose you define an object called `person` as follows:

```
function Person(name, age, sex) {
  this.name = name;
  this.age = age;
  this.sex = sex;
}
```

And then instantiate two new `person` objects as follows:

```
var rand = new Person("Rand McNally", 33, "M");
var ken = new Person("Ken Jones", 39, "M");
```

Then you can rewrite the definition of `car` to include an `owner` property that takes a person object, as follows:

```
function Car(make, model, year, owner) {
  this.make = make;
  this.model = model;
```

```
  this.year = year;
  this.owner = owner;
}
```

To instantiate the new objects, you then use the following:

```
var car1 = new Car("Eagle", "Talon TSi", 1993, rand);
var car2 = new Car("Nissan", "300ZX", 1992, ken);
```

Instead of passing a literal string or integer value when creating the new objects, the above statements pass the objects `rand` and `ken` as the parameters for the owners. To find out the name of the owner of `car2`, you can access the following property:

```
car2.owner.name
```

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 (/).

Addition (+)

The addition operator produces the sum of numeric operands or string concatenation.

Syntax

```
Operator: x + y
```

Examples

```
// Number + Number -> addition
1 + 2 // 3

// Boolean + Number -> addition
true + 1 // 2

// Boolean + Boolean -> addition
```

```
false + false // 0

// Number + String -> concatenation
5 + "foo" // "5foo"

// String + Boolean -> concatenation
"foo" + false // "foofalse"

// String + String -> concatenation
"foo" + "bar" // "foobar"
```

Subtraction (-)

The subtraction operator subtracts the two operands, producing their difference.

Syntax

```
Operator: x - y
```

Examples

```
5 - 3 // 2
3 - 5 // -2
"foo" - 3 // NaN
```

Division (/)

The division operator produces the quotient of its operands where the left operand is the dividend and the right operand is the divisor.

Syntax

```
Operator: x / y
```

Examples

```
1 / 2      // returns 0.5 in JavaScript
```

```
1 / 2      // returns 0 in Java
// (neither number is explicitly a floating point number)

1.0 / 2.0  // returns 0.5 in both JavaScript and Java

2.0 / 0    // returns Infinity in JavaScript
2.0 / 0.0  // returns Infinity too
2.0 / -0.0 // returns -Infinity in JavaScript
```

Multiplication (*)

The multiplication operator produces the product of the operands.

Syntax

```
Operator: x * y
```

Examples

```
2 * 2 // 4
-2 * 2 // -4
Infinity * 0 // NaN
Infinity * Infinity // Infinity
"foo" * 2 // NaN
```

Remainder (%)

The remainder operator returns the remainder left over when one operand is divided by a second operand. It always takes the sign of the dividend, not the divisor. It uses a built-in `modulo` function to produce the result, which is the integer remainder of dividing `var1` by `var2` â€" for example â€" `var1` modulo `var2`. There is a proposal to get an actual modulo operator in a future version of ECMAScript, the difference being that the modulo operator result would take the sign of the divisor, not the dividend.

Syntax

```
Operator: var1 % var2
```

Examples

```
12 % 5 // 2
-1 % 2 // -1
NaN % 2 // NaN
1 % 2 // 1
2 % 3 // 2
-4 % 2 // -0
5.5 % 2 // 1.5
```

Exponentiation (**)

The exponentiation operator returns the result of raising first operand to the power second operand. that is, $var1^{var2}$, in the preceding statement, where `var1` and `var2` are variables. Exponentiation operator is right associative. `a ** b ** c` is equal to `a ** (b ** c)`.

Syntax

Operator: `var1 ** var2`

Notes

In most languages like PHP and Python and others that have an exponentiation operator (typically ^ or **), the exponentiation operator is defined to have a higher precedence than unary operators such as unary + and unary -, but there are a few exceptions. For example, in Bash or in the current ES7 exponentiation operator draft spec, the ** operator is defined to have a lower precedence than unary operators.

```
-2 ** 2 // equals 4 in ES7 or in Bash, equals -4 in other languages.
```

Examples

```
2 ** 3        // 8
3 ** 2        // 9
3 ** 2.5      // 15.588457268119896
10 ** -1      // 0.1
NaN ** 2      // NaN
```

```
2 ** 3 ** 2    // 512
2 ** (3 ** 2)  // 512
(2 ** 3) ** 2  // 64
```

Increment (++)

The increment operator increments (adds one to) its operand and returns a value.

- If used postfix, with operator after operand (for example, x++), then it returns the value before incrementing.
- If used prefix with operator before operand (for example, ++x), then it returns the value after incrementing.

Syntax

Operator: x++ or ++x

Examples

```
// Postfix
var x = 3;
y = x++; // y = 3, x = 4

// Prefix
var a = 2;
b = ++a; // a = 3, b = 3
```

Decrement (--)

The decrement operator decrements (subtracts one from) its operand and returns a value.

- If used postfix (for example, x--), then it returns the value before decrementing.
- If used prefix (for example, --x), then it returns the value after decrementing.

Syntax

Operator: x-- or --x

Examples

```
// Postfix
var x = 3;
y = x--; // y = 3, x = 2

// Prefix
var a = 2;
b = --a; // a = 1, b = 1
```

Unary negation (-)

The unary negation operator precedes its operand and negates it.

Syntax

Operator: -x

Examples

```
var x = 3;
y = -x; // y = -3, x = 3
```

Unary plus (+)

The unary plus operator precedes its operand and evaluates to its operand but attempts to converts it into a number, if it isn't already. Although unary negation (-) also can convert non-numbers, unary plus is the fastest and preferred way of converting something into a number, because it does not perform any other operations on the number. It can convert string representations of integers and floats, as well as the non-string values `true`, `false`, and `null`. Integers in both decimal and hexadecimal ("0x"-prefixed) formats are supported. Negative numbers are supported (though not for hex). If it cannot parse a particular value, it will evaluate to NaN.

Syntax

Operator: +x

Examples

```
+3     // 3
+"3"   // 3
+true  // 1
+false // 0
+null  // 0
```

---

delete

The **`delete` operator** removes a property from an object.

Syntax

```
delete expression
```

where *expression* should evaluate to a property reference, e.g.:

```
delete object.property
delete object['property']
```

Parameters

`object`

The name of an object, or an expression evaluating to an object.

`property`

The property to delete.

Return value

Throws in strict mode if the property is an own non-configurable property (returns `false` in non-strict). Returns `true` in all other cases.

Description

Unlike what common belief suggests, the `delete` operator has **nothing** to do with directly freeing memory (it only does indirectly via breaking references. See the memory management page for more details).

If the `delete` operator succeeds, it removes the property from the object entirely. However, if a property with the same name exists on the object's prototype chain, the object will inherit that property from the prototype.

`delete` is only effective on an object's properties. It has no effect on variable or function names. While sometimes mis-characterized as global variables, assignments that don't specify an object (e.g. `x = 5`) are actually property assignments on the global object.

`delete` can't remove certain properties of predefined objects (like Object, Array, Math etc). These are described in ECMAScript 5 and later as non-configurable.

Temporal dead zone

The "temporal dead zone" (TDZ), specified in ECMAScript 6 for `const` and `let` declarations, also applies to the `delete` operator. Thus, code like the following will throw a **ReferenceError**.

```
function foo() {
  delete x;
  let x;
}

function bar() {
  delete y;
  const y;
}
```

Examples

```
x = 42;         // creates the property x on the global object
var y = 43;     // creates the property y on the global object, and marks
it as non-configurable
myobj = {
```

```
  h: 4,
  k: 5
};

// x is a property of the global object and can be deleted
delete x;       // returns true

// y is not configurable, so it cannot be deleted
delete y;       // returns false

// delete doesn't affect certain predefined properties
delete Math.PI; // returns false

// user-defined properties can be deleted
delete myobj.h; // returns true

// myobj is a property of the global object, not a variable,
// so it can be deleted
delete myobj;   // returns true

function f() {
  var z = 44;

  // delete doesn't affect local variable names
  delete z;     // returns false
}
```

If the object inherits a property from a prototype, and doesn't have the property itself, the property can't be deleted by referencing the object. You can, however, delete it directly on the prototype.

```
function Foo(){}
Foo.prototype.bar = 42;
var foo = new Foo();

// returns true, but with no effect,
// since bar is an inherited property
delete foo.bar;
```

```
// logs 42, property still inherited
console.log(foo.bar);

// deletes property on prototype
delete Foo.prototype.bar;

// logs "undefined", property no longer inherited
console.log(foo.bar);
```

Deleting array elements

When you delete an array element, the array length is not affected. This holds even if you delete the last element of the array.

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`.

```
var trees = ["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` value instead of the `delete` operator. In the following example, `trees[3]` is assigned the value undefined, but the array element still exists:

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

void

The **`void` operator** evaluates the given *`expression`* and then returns **undefined**.

Syntax

```
void expression
```

Description

This operator allows inserting expressions that produce side effects into places where an expression that evaluates to **undefined** is desired.

The `void` operator is often used merely to obtain the `undefined` primitive value, usually using "`void(0)`" (which is equivalent to "`void 0`"). In these cases, the global variable **undefined** can be used instead (assuming it has not been assigned to a non-default value).

Immediately Invoked Function Expressions

When using an immediately-invoked function expression, `void` can be used to force the `function` keyword to be treated as an expression instead of a declaration.

```
void function iife() {
    var bar = function () {};
    var baz = function () {};
    var foo = function () {
        bar();
        baz();
     };
    var biz = function () {};

    foo();
    biz();
}();
```

JavaScript URIs

When a browser follows a `javascript:` URI, it evaluates the code in the URI and then replaces the contents of the page with the returned value, unless the returned value is **undefined**. The `void` operator can be used to return **undefined**. For example:

```
<a href="javascript:void(0);">
```

```
  Click here to do nothing
</a>

<a href="javascript:void(document.body.style.backgroundColor='green');">
  Click here for green background
</a>
```

Note, however, that the `javascript:` pseudo protocol is discouraged over other alternatives, such as unobtrusive event handlers.

---

typeof

The **`typeof` operator** returns a string indicating the type of the unevaluated operand.

Syntax

The `typeof` operator is followed by its operand:

```
typeof operand
```

Parameters

operand is an expression representing the object or primitive whose type is to be returned.

Description

The following table summarizes the possible return values of `typeof`. For more information about types and primitives, see also the JavaScript data structure page.

Examples

```
// Numbers
typeof 37 === 'number';
typeof 3.14 === 'number';
typeof Math.LN2 === 'number';
typeof Infinity === 'number';
typeof NaN === 'number'; // Despite being "Not-A-Number"
typeof Number(1) === 'number'; // but never use this form!
```

```
// Strings
typeof "" === 'string';
typeof "bla" === 'string';
typeof (typeof 1) === 'string'; // typeof always return a string
typeof String("abc") === 'string'; // but never use this form!

// Booleans
typeof true === 'boolean';
typeof false === 'boolean';
typeof Boolean(true) === 'boolean'; // but never use this form!

// Symbols
typeof Symbol() === 'symbol'
typeof Symbol('foo') === 'symbol'
typeof Symbol.iterator === 'symbol'

// Undefined
typeof undefined === 'undefined';
typeof blabla === 'undefined'; // an undefined variable

// Objects
typeof {a:1} === 'object';

// use Array.isArray or Object.prototype.toString.call
// to differentiate regular objects from arrays
typeof [1, 2, 4] === 'object';

typeof new Date() === 'object';

// The following is confusing. Don't use!
typeof new Boolean(true) === 'object';
typeof new Number(1) === 'object';
typeof new String("abc") === 'object';

// Functions
```

```
typeof function(){} === 'function';
typeof class C {} === 'function';
typeof Math.sin === 'function';
```

`null`

```
// This stands since the beginning of JavaScript
typeof null === 'object';
```

In the first implementation of JavaScript, JavaScript values were represented as a type tag and a value. The type tag for objects was 0. `null` was represented as the NULL pointer (0x00 in most platforms). Consequently, null had 0 as type tag, hence the bogus `typeof` return value. (reference)

A fix was proposed for ECMAScript (via an opt-in), but was rejected. It would have resulted in `typeof null === 'null'`.

Regular expressions

Callable regular expressions were a non-standard addition in some browsers.

```
typeof /s/ === 'function'; // Chrome 1-12 Non-conform to ECMAScript 5.1
typeof /s/ === 'object';   // Firefox 5+  Conform to ECMAScript 5.1
```

IE-specific notes

On IE 6, 7, and 8 a lot of host objects are objects and not functions. For example:

```
typeof alert === 'object'
```

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.

Description

The logical operators are described in the following table:

Operator	Usage	Description
Logical AND (&&)	*expr1* && *expr2*	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.
Logical OR (||)	*expr1* || *expr2*	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.
Logical NOT (!)	!*expr*	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, the empty string (""), or undefined.

Even though the && and || operators can be used with operands that are not Boolean values, they can still be considered Boolean operators since their return values can always be converted to Boolean values.

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. Also note that the anything part of the above expression is any single logical expression (as indicated by the parentheses).

For example, the following two functions are equivalent.

```
function shortCircuitEvaluation() {
  doSomething() || doSomethingElse()
}

function equivalentEvaluation() {
  var flag = doSomething();
  if (!flag) {
    doSomethingElse();
  }
}
```

However, the following expressions are not equivalent due to operator precedence, and stresses the importance of requiring the right hand operator to be a single expression (grouped if needed by parentheses).

```
false && true  || true      // returns true
false && (true || true)     // returns false
```

Logical AND (&&)

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

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

Logical OR (||)

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

```
o1 = true  || true       // t || t returns true
```

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

Logical NOT (!)

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

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

Conversion rules

Converting AND to OR

the following operation involving Booleans:

```
bCondition1 && bCondition2
```

is always equal to:

```
!(!bCondition1 || !bCondition2)
```

Converting OR to AND

the following operation involving Booleans:

```
bCondition1 || bCondition2
```

is always equal to:

```
!(!bCondition1 && !bCondition2)
```

Converting between NOTs

the following operation involving Booleans:

```
!!bCondition
```

is always equal to:

```
bCondition
```

Removing nested parentheses

As logical expressions are evaluated left to right, it is always possible to remove parentheses from a complex expression following some rules.

Removing nested AND

The following composite operation involving Booleans:

```
bCondition1 || (bCondition2 && bCondition3)
```

is always equal to:

```
bCondition1 || bCondition2 && bCondition3
```

Removing nested OR

The following composite operation involving Booleans:

```
bCondition1 && (bCondition2 || bCondition3)
```

is always equal to:

```
!(!bCondition1 || !bCondition2 && !bCondition3)
```

Bitwise Operators

Bitwise operators treat their operands as a sequence of 32 bits (zeroes 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:

Signed 32-bit integers

The operands of all bitwise operators are converted to signed 32-bit integers in two's complement format. Two's complement format means that a number's negative counterpart (e.g. 5 vs. -5) is all the number's bits inverted (bitwise NOT of the number, a.k.a. ones' complement of the number) plus one. For example, the following encodes the integer 314:

```
00000000000000000000000100111010
```

The following encodes `~314`, i.e. the ones' complement of `314`:

```
11111111111111111111111011000101
```

Finally, the following encodes `-314`, i.e. the two's complement of `314`:

```
11111111111111111111111011000110
```

The two's complement guarantees that the left-most bit is 0 when the number is positive and 1 when the number is negative. Thus, it is called the *sign bit*.

The number `0` is the integer that is composed completely of 0 bits.

```
0 (base 10) = 00000000000000000000000000000000 (base 2)
```

The number `-1` is the integer that is composed completely of 1 bits.

```
-1 (base 10) = 11111111111111111111111111111111 (base 2)
```

The number `-2147483648` (hexadecimal representation: `-0x80000000`) is the integer that is composed completely of 0 bits except the first (left-most) one.

```
-2147483648 (base 10) = 10000000000000000000000000000000 (base 2)
```

The number `2147483647` (hexadecimal representation: `0x7fffffff`) is the integer that is composed completely of 1 bits except the first (left-most) one.

```
2147483647 (base 10) = 01111111111111111111111111111111 (base 2)
```

The numbers `-2147483648` and `2147483647` are the minimum and the maximum integers representable through a 32bit signed number.

Bitwise logical operators

Conceptually, the bitwise logical operators work as follows:

- The operands are converted to 32-bit integers and expressed by a series of bits (zeroes 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.

& (Bitwise AND)

Performs the AND operation on each pair of bits. `a` AND `b` yields 1 only if both `a` and `b` are 1. The truth table for the AND operation is:

```
.    9 (base 10) = 00000000000000000000000000001001 (base 2)
    14 (base 10) = 00000000000000000000000000001110 (base 2)
                   --------------------------------
14 & 9 (base 10) = 00000000000000000000000000001000 (base 2) = 8 (base 10)
```

Bitwise ANDing any number x with 0 yields 0. Bitwise ANDing any number x with -1 yields x.

| (Bitwise OR)

Performs the OR operation on each pair of bits. `a` OR `b` yields 1 if either `a` or `b` is 1. The truth table for the OR operation is:

```
.    9 (base 10) = 00000000000000000000000000001001 (base 2)
    14 (base 10) = 00000000000000000000000000001110 (base 2)
                   --------------------------------
```

```
14 | 9 (base 10) = 00000000000000000000000000001111 (base 2) = 15 (base
10)
```

Bitwise ORing any number x with 0 yields x. Bitwise ORing any number x with -1 yields -1.

^ (Bitwise XOR)

Performs the XOR operation on each pair of bits. a XOR b yields 1 if a and b are different. The truth table for the XOR operation is:

```
.    9 (base 10) = 00000000000000000000000000001001 (base 2)
    14 (base 10) = 00000000000000000000000000001110 (base 2)
                   --------------------------------
14 ^ 9 (base 10) = 00000000000000000000000000000111 (base 2) = 7 (base 10)
```

Bitwise XORing any number x with 0 yields x. Bitwise XORing any number x with -1 yields ~x.

~ (Bitwise NOT)

Performs the NOT operator on each bit. NOT a yields the inverted value (a.k.a. one's complement) of a. The truth table for the NOT operation is:

```
 9 (base 10) = 00000000000000000000000000001001 (base 2)
               --------------------------------
~9 (base 10) = 11111111111111111111111111110110 (base 2) = -10 (base 10)
```

Bitwise NOTing any number x yields -(x + 1). For example, ~5 yields -6.

Example with indexOf:

```
var str = 'rawr';
var searchFor = 'a';

// this is alternative way of typing if (-1*str.indexOf('a') <= -1)
if (~str.indexOf(searchFor)) {
  // searchFor is in the string
} else {
  // searchFor is not in the string
```

```
}

// here are the values returned by (~str.indexOf(searchFor))
// r == -1
// a == -2
// w == -3
```

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 32-bit integers in big-endian order and return a result of the same type as the left operand. The right operand should be less than 32, but if not only the low five bits will be used.

<< (Left shift)

This operator shifts the first operand the specified number of bits to the left. Excess bits shifted off to the left are discarded. Zero bits are shifted in from the right.

For example, `9 << 2` yields 36:

```
.    9 (base 10): 00000000000000000000000000001001 (base 2)
                  --------------------------------
9 << 2 (base 10): 00000000000000000000000000100100 (base 2) = 36 (base 10)
```

Bitwise shifting any number **x** to the left by **y** bits yields **x * 2^y**.

>> (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. Since the new leftmost bit has the same value as the previous leftmost bit, the sign bit (the leftmost bit) does not change. Hence the name "sign-propagating".

For example, `9 >> 2` yields 2:

```
.    9 (base 10): 00000000000000000000000000001001 (base 2)
                  --------------------------------
9 >> 2 (base 10): 00000000000000000000000000000010 (base 2) = 2 (base 10)
```

Likewise, `-9 >> 2` yields -3, because the sign is preserved:

```
.    -9 (base 10): 11111111111111111111111111110111 (base 2)
                   --------------------------------
-9 >> 2 (base 10): 11111111111111111111111111111101 (base 2) = -3 (base
10)
```

>>> (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. The sign bit becomes 0, so the result is always non-negative.

For non-negative numbers, zero-fill right shift and sign-propagating right shift yield the same result. For example, `9 >>> 2` yields 2, the same as `9 >> 2`:

```
.     9 (base 10): 00000000000000000000000000001001 (base 2)
                   --------------------------------
9 >>> 2 (base 10): 00000000000000000000000000000010 (base 2) = 2 (base 10)
```

However, this is not the case for negative numbers. For example, `-9 >>> 2` yields 1073741821, which is different than `-9 >> 2` (which yields -3):

```
.     -9 (base 10): 11111111111111111111111111110111 (base 2)
                    --------------------------------
-9 >>> 2 (base 10): 00111111111111111111111111111101 (base 2) = 1073741821
(base 10)
```

Examples

Flags and bitmasks

The bitwise logical operators are often used to create, manipulate, and read sequences of *flags*, which are like binary variables. Variables could be used instead of these sequences, but binary flags take much less memory (by a factor of 32).

Suppose there are 4 flags:

- flag A: we have an ant problem
- flag B: we own a bat
- flag C: we own a cat
- flag D: we own a duck

These flags are represented by a sequence of bits: DCBA. When a flag is *set*, it has a value of 1. When a flag is *cleared*, it has a value of 0. Suppose a variable `flags` has the binary value 0101:

```
var flags = 5;   // binary 0101
```

This value indicates:

- flag A is true (we have an ant problem);
- flag B is false (we don't own a bat);
- flag C is true (we own a cat);
- flag D is false (we don't own a duck);

Since bitwise operators are 32-bit, 0101 is actually 00000000000000000000000000000101, but the preceding zeroes can be neglected since they contain no meaningful information.

A *bitmask* is a sequence of bits that can manipulate and/or read flags. Typically, a "primitive" bitmask for each flag is defined:

```
var FLAG_A = 1; // 0001
var FLAG_B = 2; // 0010
var FLAG_C = 4; // 0100
var FLAG_D = 8; // 1000
```

New bitmasks can be created by using the bitwise logical operators on these primitive bitmasks. For example, the bitmask 1011 can be created by ORing FLAG_A, FLAG_B, and FLAG_D:

```
var mask = FLAG_A | FLAG_B | FLAG_D; // 0001 | 0010 | 1000 => 1011
```

Individual flag values can be extracted by ANDing them with a bitmask, where each bit with the value of one will "extract" the corresponding flag. The bitmask *masks* out the non-relevant flags by ANDing with zeroes (hence the term "bitmask"). For example, the bitmask 0101 can be used to see if flag C is set:

```
// if we own a cat
if (flags & FLAG_C) { // 0101 & 0100 => 0100 => true
   // do stuff
}
```

A bitmask with multiple set flags acts like an "either/or". For example, the following two are equivalent:

```
// if we own a bat or we own a cat
// (0101 & 0010) || (0101 & 0100) => 0000 || 0100 => true
if ((flags & FLAG_B) || (flags & FLAG_C)) {
   // do stuff
}
// if we own a bat or cat
var mask = FLAG_B | FLAG_C; // 0010 | 0100 => 0110
if (flags & mask) { // 0101 & 0110 => 0100 => true
   // do stuff
}
```

Flags can be set by ORing them with a bitmask, where each bit with the value one will set the corresponding flag, if that flag isn't already set. For example, the bitmask 1100 can be used to set flags C and D:

```
// yes, we own a cat and a duck
var mask = FLAG_C | FLAG_D; // 0100 | 1000 => 1100
flags |= mask;   // 0101 | 1100 => 1101
```

Flags can be cleared by ANDing them with a bitmask, where each bit with the value zero will clear the corresponding flag, if it isn't already cleared. This bitmask can be created by NOTing primitive bitmasks. For example, the bitmask 1010 can be used to clear flags A and C:

```
// no, we don't have an ant problem or own a cat
```

```
var mask = ~(FLAG_A | FLAG_C); // ~0101 => 1010
flags &= mask;   // 1101 & 1010 => 1000
```

The mask could also have been created with `~FLAG_A & ~FLAG_C` (De Morgan's law):

```
// no, we don't have an ant problem, and we don't own a cat
var mask = ~FLAG_A & ~FLAG_C;
flags &= mask;   // 1101 & 1010 => 1000
```

Flags can be toggled by XORing them with a bitmask, where each bit with the value one will toggle the corresponding flag. For example, the bitmask 0110 can be used to toggle flags B and C:

```
// if we didn't have a bat, we have one now,
// and if we did have one, bye-bye bat
// same thing for cats
var mask = FLAG_B | FLAG_C;
flags = flags ^ mask;   // 1100 ^ 0110 => 1010
```

Finally, the flags can all be flipped with the NOT operator:

```
// entering parallel universe...
flags = ~flags;    // ~1010 => 0101
```

Conversion snippets

Convert a binary `String` to a decimal `Number`:

```
var sBinString = "1011";
var nMyNumber = parseInt(sBinString, 2);
alert(nMyNumber); // prints 11, i.e. 1011
```

Convert a decimal `Number` to a binary `String`:

```
var nMyNumber = 11;
var sBinString = nMyNumber.toString(2);
alert(sBinString); // prints 1011, i.e. 11
```

Automatize the creation of a mask

If you have to create many masks from some `Boolean` values, you can automatize the process:

```
function createMask () {
  var nMask = 0, nFlag = 0, nLen = arguments.length > 32 ? 32 :
arguments.length;
  for (nFlag; nFlag < nLen; nMask |= arguments[nFlag] << nFlag++);
  return nMask;
}
var mask1 = createMask(true, true, false, true); // 11, i.e.: 1011
var mask2 = createMask(false, false, true); // 4, i.e.: 0100
var mask3 = createMask(true); // 1, i.e.: 0001
// etc.

alert(mask1); // prints 11, i.e.: 1011
```

Reverse algorithm: an array of booleans from a mask

If you want to create an `Array` of `Booleans` from a mask you can use this code:

```
function arrayFromMask (nMask) {
  // nMask must be between -2147483648 and 2147483647
  if (nMask > 0x7fffffff || nMask < -0x80000000) {
    throw new TypeError("arrayFromMask - out of range");
  }
  for (var nShifted = nMask, aFromMask = []; nShifted;
       aFromMask.push(Boolean(nShifted & 1)), nShifted >>>= 1);
  return aFromMask;
}

var array1 = arrayFromMask(11);
var array2 = arrayFromMask(4);
var array3 = arrayFromMask(1);

alert("[" + array1.join(", ") + "]");
// prints "[true, true, false, true]", i.e.: 11, i.e.: 1011
```

You can test both algorithms at the same timeâ€¦

```
var nTest = 19; // our custom mask
var nResult = createMask.apply(this, arrayFromMask(nTest));

alert(nResult); // 19
```

For didactic purpose only (since there is the `Number.toString(2)` method), we show how it is possible to modify the `arrayFromMask` algorithm in order to create a `String` containing the binary representation of a `Number`, rather than an `Array` of `Booleans`:

```
function createBinaryString (nMask) {
  // nMask must be between -2147483648 and 2147483647
  for (var nFlag = 0, nShifted = nMask, sMask = ""; nFlag < 32;
       nFlag++, sMask += String(nShifted >>> 31), nShifted <<= 1);
  return sMask;
}
var string1 = createBinaryString(11);
var string2 = createBinaryString(4);
var string3 = createBinaryString(1);
alert(string1);
// prints 00000000000000000000000000001011, i.e. 11
```

Conditional Operator

The **conditional (ternary) operator** is the only JavaScript operator that takes three operands. This operator is frequently used as a shortcut for the `if` statement.

Syntax

```
condition ? expr1 : expr2
```

Parameters

`condition`

An expression that evaluates to `true` or `false`.

`expr1`, `expr2`

Expressions with values of any type.

Description

If `condition` is `true`, the operator returns the value of `expr1`; otherwise, it returns the value of `expr2`. For example, to display a different message based on the value of the `isMember` variable, you could use this statement:

```
"The fee is " + (isMember ? "$2.00" : "$10.00")
```

You can also assign variables depending on a ternary result:

```
var elvisLives = Math.PI > 4 ? "Yep" : "Nope";
```

Multiple ternary evaluations are also possible (note: the conditional operator is right associative):

```
var firstCheck = false,
    secondCheck = false,
    access = firstCheck ? "Access denied" : secondCheck ? "Access denied"
: "Access granted";

console.log( access ); // logs "Access granted"
```

You can also use ternary evaluations in free space in order to do different operations:

```
var stop = false, age = 16;

age > 18 ? location.assign("continue.html") : stop = true;
```

You can also do more than one single operation per case, separating them with a comma:

```
var stop = false, age = 23;

age > 18 ? (
    alert("OK, you can go."),
    location.assign("continue.html")
) : (
    stop = true,
    alert("Sorry, you are much too young!")
);
```

You can also do more than one operation during the assignation of a value. In this case, ***the last comma-separated value of the parenthesis*** **will be the value to be assigned**.

```
var age = 16;

var url = age > 18 ? (
    alert("OK, you can go."),
    // alert returns "undefined", but it will be ignored because
    // isn't the last comma-separated value of the parenthesis
    "continue.html" // the value to be assigned if age > 18
) : (
    alert("You are much too young!"),
    alert("Sorry :-("),
    // etc. etc.
    "stop.html" // the value to be assigned if !(age > 18)
);

location.assign(url); // "stop.html"
```

Assignment Operators

An **assignment operator** assigns a value to its left operand based on the value of its right operand.

Overview

The basic 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`. The other assignment operators are usually shorthand for standard operations, as shown in the following definitions and examples.

Assignment

Simple assignment operator which assigns a value to a variable. Chaining the assignment operator is possible in order to assign a single value to multiple variables. See the example.

Syntax

```
Operator: x = y
```

Examples

```
// Assuming the following variables
//  x = 5
//  y = 10
//  z = 25

x = y     // x is 10
x = y = z // x, y and z are all 25
```

Addition assignment

The addition assignment operator **adds** the value of the right operand to a variable and assigns the result to the variable. The types of the two operands determine the behavior of the addition assignment operator. Addition or concatenation is possible. See the **addition operator** for more details.

Syntax

```
Operator: x += y
Meaning:  x  = x + y
```

Examples

```
// Assuming the following variables
//  foo = "foo"
//  bar = 5
//  baz = true


// Number + Number -> addition
bar += 2 // 7

// Boolean + Number -> addition
baz += 1 // 2

// Boolean + Boolean -> addition
```

```
baz += false // 1

// Number + String -> concatenation
bar += "foo" // "5foo"

// String + Boolean -> concatenation
foo += false // "foofalse"

// String + String -> concatenation
foo += "bar" // "foobar"
```

Subtraction assignment

The subtraction assignment operator **subtracts** the value of the right operand from a variable and assigns the result to the variable. See the **subtraction operator** for more details.

Syntax

```
Operator: x -= y
Meaning:  x  = x - y
```

Examples

```
// Assuming the following variable
//  bar = 5

bar -= 2     // 3
bar -= "foo" // NaN
```

Multiplication assignment

The multiplication assignment operator **multiplies** a variable by the value of the right operand and assigns the result to the variable. See the **multiplication operator** for more details.

Syntax

```
Operator: x *= y
Meaning:  x  = x * y
```

Examples

```
// Assuming the following variable
//  bar = 5

bar *= 2     // 10
bar *= "foo" // NaN
```

Division assignment

The division assignment operator **divides** a variable by the value of the right operand and assigns the result to the variable. See the **division operator** for more details.

Syntax

```
Operator: x /= y
Meaning:  x  = x / y
```

Examples

```
// Assuming the following variable
//  bar = 5

bar /= 2     // 2.5
bar /= "foo" // NaN
bar /= 0     // Infinity
```

Remainder assignment

The remainder assignment operator **divides** a variable by the value of the right operand and assigns the **remainder** to the variable. See the **remainder operator** for more details.

Syntax

```
Operator: x %= y
Meaning:  x  = x % y
```

Examples

```
// Assuming the following variable
//  bar = 5

bar %= 2     // 1
bar %= "foo" // NaN
bar %= 0     // NaN
```

Exponentiation assignment

The exponentiation assignment operator returns the result of raising first operand to the **power** second operand. See the **exponentiation operator** for more details.

Syntax

```
Operator: x **= y
Meaning:  x  = x ** y
```

Examples

```
// Assuming the following variable
//  bar = 5

bar **= 2     // 25
bar **= "foo" // NaN
```

Left shift assignment

The left shift assignment operator moves the specified amount of bits to the left and assigns the result to the variable. See the **left shift operator** for more details.

Syntax

```
Operator: x <<= y
Meaning:  x   = x << y
```

Examples

```
var bar = 5; //  (00000000000000000000000000000101)
bar <<= 2; // 20 (00000000000000000000000000010100)
```

Right shift assignment

The right shift assignment operator moves the specified amount of bits to the right and assigns the result to the variable. See the **right shift operator** for more details.

Syntax

```
Operator: x >>= y
Meaning:  x   = x >> y
```

Examples

```
var bar = 5; //   (00000000000000000000000000000101)
bar >>= 2;   // 1 (00000000000000000000000000000001)

var bar -5; //    (-00000000000000000000000000000101)
bar >>= 2;  // -2 (-00000000000000000000000000000010)
```

Unsigned right shift assignment

The unsigned right shift assignment operator moves the specified amount of bits to the right and assigns the result to the variable. See the **unsigned right shift operator** for more details.

Syntax

```
Operator: x >>>= y
Meaning:  x    = x >>> y
```

Examples

```
var bar = 5; //   (00000000000000000000000000000101)
bar >>>= 2;  // 1 (00000000000000000000000000000001)

var bar = -5; // (-00000000000000000000000000000101)
```

```
bar >>>= 2; // 1073741822 (00111111111111111111111111111110)
```

Bitwise AND assignment

The bitwise AND assignment operator uses the binary representation of both operands, does a bitwise AND operation on them and assigns the result to the variable. See the **bitwise AND operator** for more details.

Syntax

```
Operator: x &= y
Meaning:  x  = x & y
```

Example

```
var bar = 5;
// 5:     00000000000000000000000000000101
// 2:     00000000000000000000000000000010
bar &= 2; // 0
```

Bitwise XOR assignment

The bitwise XOR assignment operator uses the binary representation of both operands, does a bitwise XOR operation on them and assigns the result to the variable. See the **bitwise XOR operator** for more details.

Syntax

```
Operator: x ^= y
Meaning:  x  = x ^ y
```

Example

```
var bar = 5;
bar ^= 2; // 7
// 5: 00000000000000000000000000000101
// 2: 00000000000000000000000000000010
```

```
// ----------------------------------
// 7: 00000000000000000000000000000111
```

Bitwise OR assignment

The bitwise OR assignment operator uses the binary representation of both operands, does a bitwise OR operation on them and assigns the result to the variable. See the **bitwise OR operator** for more details.

Syntax

```
Operator: x |= y
Meaning:  x  = x | y
```

Example

```
var bar = 5;
bar |= 2; // 7
// 5: 00000000000000000000000000000101
// 2: 00000000000000000000000000000010
// ----------------------------------
// 7: 00000000000000000000000000000111
```

Examples

Left operand with another assignment operator

In unusual situations, the assignment operator (e.g. `x += y`) is not identical to the meaning expression (here `x = x + y`). When the left operand of an assignment operator itself contains an assignment operator, the left operand is evaluated only once. For example:

```
a[i++] += 5          // i is evaluated only once
a[i++] = a[i++] + 5 // i is evaluated twice
```

Comma Operator

The **comma operator** evaluates each of its operands (from left to right) and returns the value of the last operand.

Syntax

expr1, expr2, expr3...

Parameters

`expr1`, `expr2, expr3...`

Any expressions.

Description

You can use the comma operator when you want to include multiple expressions in a location that requires a single expression. The most common usage of this operator is to supply multiple parameters in a `for` loop.

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. Note that the comma in the `var` statement is ***not*** the comma operator, because it doesn't exist within an expression. Rather, it is a special character in `var` statements to combine multiple of them into one. Practically, that comma behaves almost the same as the comma operator, though. The code prints the values of the diagonal elements in the array:

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

Processing and then returning

Another example that one could make with comma operator is processing before returning. As stated, only the last element will be returned but all others are going to be evaluated as well. So, one could do:

```
function myFunc () {
  var x = 0;

  return (x += 1, x); // the same as return ++x;
}
```

Thanks for read

WhatsApp : +56 9 3711 3110